VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

PERIFERIE PROCESORU RISC-V

RISC-V PROCESSOR PERIPHERALS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. TOMÁŠ VAVRO**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MARTÍNEK, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Vavro Tomáš, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Počítačové a vestavěné systémy
Název: **Periferie procesoru RISC-V**
RISC-V Processor Peripherals
Kategorie: Počítačová architektura
Zadání:

1. Seznamte se s architekturou procesoru RISC-V a způsoby připojení základních periferií k tomuto procesoru (např. UART, I2C, GPIO, TIMER, PLIC).
2. Seznamte se s jazyky pro popis hardware VHDL/Verilog a z technikami verifikace hardwarových obvodů.
3. Proveďte návrh a implementaci vybraných periferií s využitím jazyka VHDL nebo Verilog.
4. Vytvořte verifikační prostředí k těmto komponentám a jejich funkčnost ověřte.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího pokračování projektu.

Literatura:

- Dle pokynů vedoucího práce.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Martínek Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

Abstrakt

Platforma RISC-V je jedným z lídrov v odvetví počítačových a vstavaných systémov. Pri čoraz väčšej miere využívania takýchto systémov rastie dopyt po dostupných perifériách pre implementácie tejto platformy. Táto práca sa zaoberá procesorom FU540-C000 od spoločnosti SiFive, ktorý je jednou z implementácií architektúry RISC-V, a jeho základnými perifériami. Na základe analýzy bol spomedzi periférií tohoto procesoru zvolený obvod UART slúžiaci pre asynchrónnu sériovú komunikáciu. Cieľom tejto diplomovej práce je danú perifériu navrhnúť a implementovať v niektorom z jazykov pre popis číslicových obvodov, a následne vytvoriť verifikačné prostredie, prostredníctvom ktorého bude overená funkčnosť implementácie.

Abstract

The RISC-V platform is one of the leaders in the computer and embedded systems industry. With the increasing use of these systems, the demand for available peripherals for the implementations of this platform is growing. This thesis deals with the FU540-C000 processor from SiFive company, which is one of the implementations of the RISC-V architecture, and its basic peripherals. Based on the analysis, an UART circuit for asynchronous serial communication was selected from the peripherals of this processor. The aim of this master thesis is to design and implement the peripheral in one of the languages for the description of digital circuits, and then create a verification environment, through which the functionality of the implementation will be verified.

Kľúčové slová

RISC-V, procesor, UART, sériová komunikácia, VHDL, návrh číslicových systémov, SystemVerilog, UVM, Universal Verification Methodology, funkčná verifikácia číslicových systémov

Keywords

RISC-V, processor, UART, serial communication, VHDL, design of digital systems, SystemVerilog, UVM, Universal Verification Methodology, functional verification of digital systems

Citácia

VAVRO, Tomáš. *Periferie procesoru RISC-V*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Martínek, Ph.D.

Periferie procesoru RISC-V

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Tomáša Martínka, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

. . . .
Tomáš Vavro
18. mája 2021

Poďakovanie

Rád by som sa srdečne poďakoval vedúcemu mojej diplomovej práce pánovi Ing. Tomášovi Martínkovi, Ph.D. za jeho ochotu, čas a odborné rady, ktoré mi venoval.

Obsah

1 Úvod **5**

2 Teoretická časť **6**
2.1 RISC-V . . . 6
2.1.1 Režimy procesoru RISC-V . . . 6
2.1.2 ISA . . . 7
2.1.3 Procesor SiFive FU540-C000 . . . 7
2.2 Sériová komunikácia . . . 8
2.2.1 Universal Asynchronous Receiver-Transmitter . . . 10
2.2.2 Periféria UART pre procesor SiFive FU540-C000 . . . 12
2.3 Návrh číslicových obvodov . . . 15
2.3.1 Jazyk VHDL . . . 16
2.4 Funkčná verifikácia číslicových obvodov . . . 18
2.4.1 Samokontrolné mechanizmy . . . 18
2.4.2 Funkčná verifikácia riadená pokrytím . . . 19
2.4.3 Pseudonáhodné testy . . . 20
2.4.4 Jazyk SystemVerilog . . . 20
2.4.5 Universal Verification Methodology . . . 21

3 Praktická časť **27**
3.1 Zhodnotenie a kritika súčasného stavu . . . 27
3.1.1 Analýza existujúceho ovládaču . . . 27
3.1.2 Analýza existujúcich implementácií . . . 28
3.1.3 Záver hodnotenia . . . 30
3.2 Návrh implementácie . . . 30
3.2.1 Rozhranie obvodu UART . . . 30
3.2.2 Zapojenie riadiacich registrov s rozhraním . . . 32
3.2.3 FIFO . . . 32
3.2.4 Vysielacia časť . . . 33
3.2.5 Prijímacia časť . . . 34
3.2.6 Prerušenia . . . 34
3.3 Implementácia . . . 35
3.3.1 FIFO . . . 35
3.3.2 Výstupná komponenta vysielacej časti . . . 36
3.3.3 Vstupná komponenta prijímacej časti . . . 37
3.3.4 UART . . . 38
3.4 Verifikácia . . . 39
3.4.1 Verifikačné prostredie . . . 40

3.4.2 Verifikačné scenáre 41

4 Záver **45**

Literatúra **46**

Prílohy **49**
Zoznam príloh 50

A Obsah priloženého pamäťového média **51**

B Schéma výsledného obvodu **52**

Zoznam obrázkov

2.1 Schéma procesoru SiFive FU540-C000 na najvyššej úrovni [28] 8
2.2 Diagram paralelnej a sériovej komunikácie [13] 9
2.3 Grafické znázornenie komunikácie v technológii UART [7] 10
2.4 Ukážkový dátový rámec sériového protokolu UART [7] 11
2.5 Vzorkovanie technikou 16 × oversampling with 2/3 majority voting per bit 11
2.6 Schéma zapojenia verifikačného prostredia a verifikovaného obvodu 18
2.7 Kontrola odozvy verifikovaného obvodu pomocou referenčného modelu . . . 19
2.8 Proces úpravy verifikačných testov na základe analýzy metrík pokrytia [33] 19
2.9 Vývoj verifikačných metodík naprieč časom [33] 21
2.10 Oddelenie testov od verifikačného prostredia v UVM [9] 22
2.11 Hierarchická triedna štruktúra verifikačnej metodiky UVM [8] 23
2.12 Architektúra verifikačného prostredia podľa metodiky UVM [11] 23
2.13 Jednoduchá ukážka transakcie v UVM 24
2.14 Schéma verifikačnej komponenty v UVM [8] 25
2.15 Ukážka prepojenia analytických komponent s monitorom v UVM [10] . . . 25

3.1 Schéma rozhrania obvodu UART 30
3.2 Časový diagram zápisovej transakcie 31
3.3 Časový diagram čítacej transakcie 31
3.4 Schéma zapojenia riadiacich registrov k rozhraniu 32
3.5 Schéma bufferu typu FIFO 33
3.6 Schéma vysielacej časti obvodu UART 33
3.7 Schéma prijímacej časti obvodu UART 34
3.8 Schéma časti obvodu zaisťujúcej vyvolanie a propagáciu prerušení 35
3.9 Stavový automat využívaný výstupnou komponentou vysielacej časti obvodu UART 36
3.10 Stavový automat využívaný vstupnou komponentou prijímacej časti obvodu UART 38
3.11 Schéma verifikačného prostredia na najvyššej úrovni 40
3.12 Formát transakcie používanej v implementovanom verifikačnom prostredí . 40
3.13 Schéma komponenty *enviroment* používanej verifikačným prostredím 41

Zoznam tabuliek

2.1 Povolené kombinácie privilegovaných režimov 6
2.2 Parametre periférií UART procesoru SiFive FU540-C000 12
2.3 Adresový priestor riadiacich registrov periférie UART 12
2.4 Register Transmit Data 13
2.5 Register Receive Data 13
2.6 Register Transmit Control 13
2.7 Register Receive Control 14
2.8 Register Interrupt Pending 14
2.9 Register Interrupt Enable 14
2.10 Register Baud Rate Divisor 15
2.11 Často využívané prenosové rýchlosti a hodnoty pre ich generovanie 15

3.1 Základné výstupy zo syntézy obvodu UART 39
3.2 Hodnoty pokrytia kódu a funkčného pokrytia dosiahnuté pri verifikácii . . . 44

Kapitola 1

Úvod

Všetky odvetvia informačných technológií v dnešnej dobe zažívajú dramatický rozmach. Spolu s nimi tiež narastá miera využívania počítačových a vstavaných systémov. Jedného z lídrov tejto tejto oblasti predstavuje platforma RISC-V. Táto platforma poskytuje voľne dostupnú inštrukčnú sadu, cielenú na široké spektrum aplikácií. S rastúcou mierou používania implementácií tejto platformy vzniká čím ďalej tým väčší dopyt po dostupných perifériách pre tieto implementácie.

Posun nastal taktiež v procese vývoja takýchto systémov. V súčasnosti sa používajú rôzne jazyky pre popis integrovaných obvodov, ako sú VHDL alebo Verilog. Popis daného obvodu je následne vyrobený ako ASIC, prípadne nahraný do programovateľného čipu FPGA. Pri vývoji takýchto systémov bežne nastáva jav, kedy je do implementácie systému vnesená chyba a systém teda nepracuje podľa špecifikácie. Odbor zaoberajúci sa odhaľovaním takýchto chýb sa nazýva verifikácia a v dnešnej dobe už existuje viacero komplexných verifikačných metodík.

Táto práca sa zaoberá základnými perifériami konkrétnej implementácie procesoru architektúry RISC-V. Jedná sa o procesor FU540-C000 od spoločnosti SiFive a spomedzi jeho dostupných periférií bol zvolený obvod UART, ktorý slúži ako periféria pre asynchrónnu sériovú komunikáciu. Cieľom tejto diplomovej práce je analyzovať špecifikáciu tohoto obvodu a jeho existujúceho ovládaču. Na základe tejto analýzy bude následne vypracovaný návrh daného obvodu a tiež jeho implementácia v jazyku VHDL. Pre overenie funkčnosti implementácie bude použitá verifikačná metodika UVM.

Práca je rozdelená medzi dve väčšie kapitoly. Kapitola 2 sa venuje teoretickému základu tejto práce. V úvode popisuje základné parametre a vlastnosti procesorovej architektúry RISC-V, pričom sa venuje aj jej konkrétnej implementácií, procesoru FU540-C000. V ďalšej časti nasledujú princípy sériovej komunikácie, pričom je kladený dôraz na asynchrónnu komunikáciu a technológiu UART. Nasledujúca časť pojednáva o disciplíne návrhu číslicových obvodov a uvádza prehľad základných techník v programovacom jazyku VHDL. V záverečnej časti je popísaný proces verifikácie číslicových obvodov a verifikačná metodika UVM. V kapitole 3 je popísaná realizácia praktickej časti tejto diplomovej práce. Úvodná časť obsahuje prehľad existujúcich implementácií obvodu UART a skúma ich vhodnosť pre pripojenie k systému FU540-C000 z hľadiska špecifikácie a existujúceho ovládaču. Témou nasledujúcej časti je návrh vlastného riešenia obvodu UART. Ďalšia časť popisuje významné aspekty implementácie navrhnutého obvodu a v záverečnej časti je uvedená realizácia verifikačného prostredia, a použitých verifikačných scenárov.

Kapitola 2

Teoretická časť

Táto kapitola predstavuje teoretický základ pre túto prácu. V prvej časti tejto kapitoly budú popísané základné vlastnosti platformy RISC-V a jej konkrétnej implementácie FU540-C000 od spoločnosti SiFive. Nasledovať bude časť o princípoch sériovej komunikácie a technológii UART, na ktorú táto práca cieli. Ďalšia podkapitola popisuje problematiku návrhu číslicových obvodov a záverečná časť sa venuje verifikácii takýchto obvodov.

2.1 RISC-V

RISC-V je voľne dostupná inštrukčná sada, ďalej len **ISA** (Instruction Set Architecture), ktorá vznikla pôvodne za účelom výuky a výskumu počítačových architektúr v roku 2010 v Spojených štátoch amerických na univerzite University of California, Berkeley. V súčasnosti už RISC-V predstavuje priemyselný štandard a stojí za ním asociácia RISC-V Foundation založená v roku 2015. Primárnym cieľom a výnimočnosťou tohoto štandardu je poskytnúť voľne dostupnú ISA, ktorá sa nešpecializuje len na jednu cieľovú technológiu alebo mikroarchitektúru, ale na čo najviac rôznych platforiem, od malých vstavaných systémov až po zložité serverové systémy, čím zaisťuje väčšiu kompatibilitu programov medzi čipmi od rôznych výrobcov, viď [12]. Základ tejto ISA je postavený na princípoch inštrukčnej sady RISC (Reduced Instruction Set Computing) a ponúka možnosť vysokej úrovne paralelizácie.

2.1.1 Režimy procesoru RISC-V

Režim daného procesoru definuje množinu operácií, ktoré môže nejaký proces bežiaci na tomto procesore vykonávať.

Počet režimov	**Podporované režimy**	**Použitie**
1	M	Jednoduché vstavané systémy
2	M, U	Zabezpečené vstavané systémy
3	M, U, S	Systémy s podporou unixových operačných systémov

Tabuľka 2.1: Povolené kombinácie privilegovaných režimov

Potreba existencie režimov vyplýva z možnosti, že vykonávaný kód môže byť škodlivý. Štandard RISC-V definuje celkom tri privilegované režimy procesoru, pomocou ktorých je možné vytvoriť ľubovoľný systém:

- Strojový režim **M** (Machine) - režim s najväčšími právomocami, v ktorom nie je vykonávanie inštrukcií obmedzené žiadnym spôsobom, pretože tieto inštrukcie pristupujú k procesoru na nízkej úrovni, čím sú považované za dôveryhodné. Tento režim je základný a preto musí byť zahrnutý v každej implementácii.
- Užívateľský režim **U** (User) - tento režim je určený pre používanie bežných užívateľských aplikácií a operačných systémov, kde sa potenciálne môže vyskytnúť škodlivý zdrojový kód. Vykonávanie inštrukcií v tomto režime je preto oproti strojovému režimu obmedzené.
- Dozorujúci režim **S** (Supervisor) - jedná sa o podobný režim ako užívateľský, s rozšírením o podporu virtualizácie, vďaka čomu mohol byť odstránený virtualizačný režim **H** (Hypervisor).

Uvedené privilegované režimy je možné v implementáciách RISC-V kombinovať tak, ako je ukázané v tabuľke 2.1.

2.1.2 ISA

Základná ISA je tvorená množinou inštrukcií pre celočíselné operácie, ktoré sú potrebné na fungovanie operačného systému, prekladaču apod. Tento základ musí byť zahrnutý v každej implementácii RISC-V a je možné ho doplniť veľkým množstvom rozšírení, ktoré budú uvedené v nasledujúcej časti tejto podkapitoly. V súčasnosti sú dostupné tri varianty základnej ISA, dedikované na rôzne veľkosti registrov v adresovom priestore architektúry. Jedná sa o 32 bitovú verziu **RV32I**, 64 bitovú verziu **RV64I** a 128 bitovú verziu **RV128I**, kde **I** značí celočíselnú aritmetiku (Integer). Tieto sady poskytujú inštrukcie pre základné celočíselné (sčítanie, odčítanie, logické operácie apod.), pamäťové (čítanie, zápis) a riadiace (skoky) operácie [12].

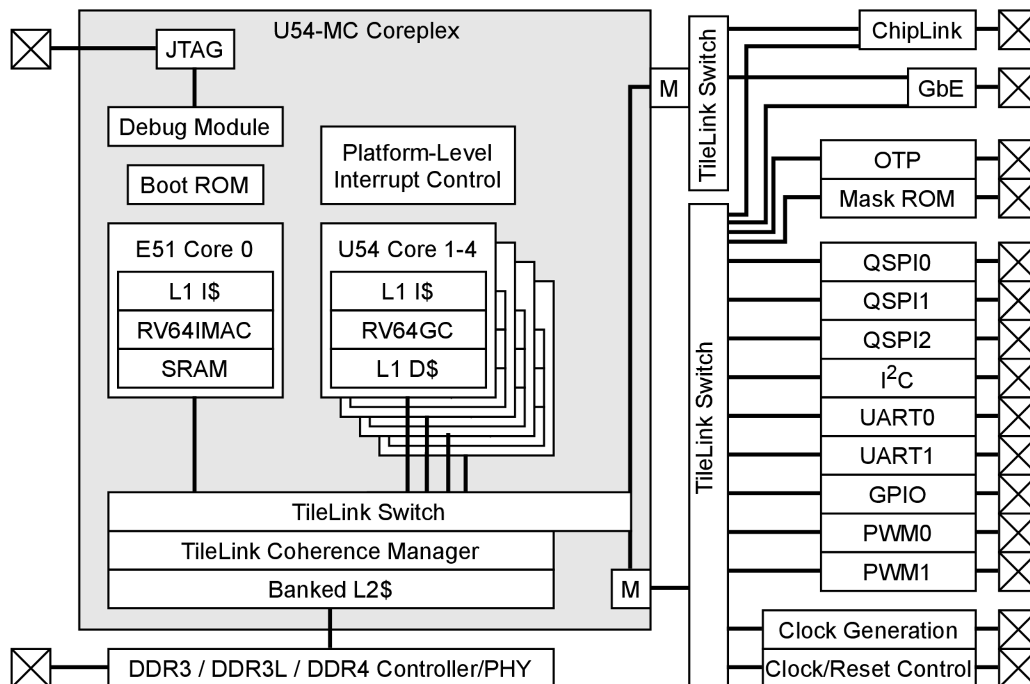
Štandardné rozšírenia

Sú to všeobecne použiteľné rozšírenia, ktoré neprichádzajú do konfliktu so základnou ISA a ani medzi sebou navzájom. Oproti tomu je možné využiť aj neštandardné rozšírenia, ktoré sú určené pre vysoko špecializované úlohy. Nie je pri nich však zaručené, že nebudú kolidovať s inými rozšíreniami.

Rozšírenie **M** (Multiplication) poskytuje inštrukcie pre operácie násobenia, delenia a delenia so zvyškom. Rozšírenia **F** (Float) a **D** (Double) obsahujú inštrukcie, a registre pre prácu s číslami, ktoré majú pohyblivú rádovú čiarku, pričom rešpektujú jednoduchú, respektíve dvojitú presnosť. Štandardné rozšírenie **A** (Atomic) pridáva skupinu inštrukcií vykonávajúcich atomický zápis, modifikáciu a čítanie z pamäti, ktoré je možné využiť pri synchronizácii medzi procesormi. Pokiaľ sú použité všetky uvedené rozšírenia, a teda **IMAFD**, takáto konfigurácia sa označuje **G** (General-purpose) [12]. Pre používanie komprimovaných inštrukcií s menšou dĺžkou je k dispozícii rozšírenie označené **C** (Compressed). Existujú aj ďalšie štandardné rozšírenia ISA, ktoré prinášajú možnosti napríklad pre vektorizáciu, bitové operácie, prácu s číslami s pohyblivou rádovou čiarkou v štvoritej presnosti apod.

2.1.3 Procesor SiFive FU540-C000

Procesor **FU540-C000** od spoločnosti **SiFive** [29] predstavuje 64 bitovú implementáciu architektúry RISC-V, ktorá ponúka podporu plnohodnotným operačným systémom, akým je



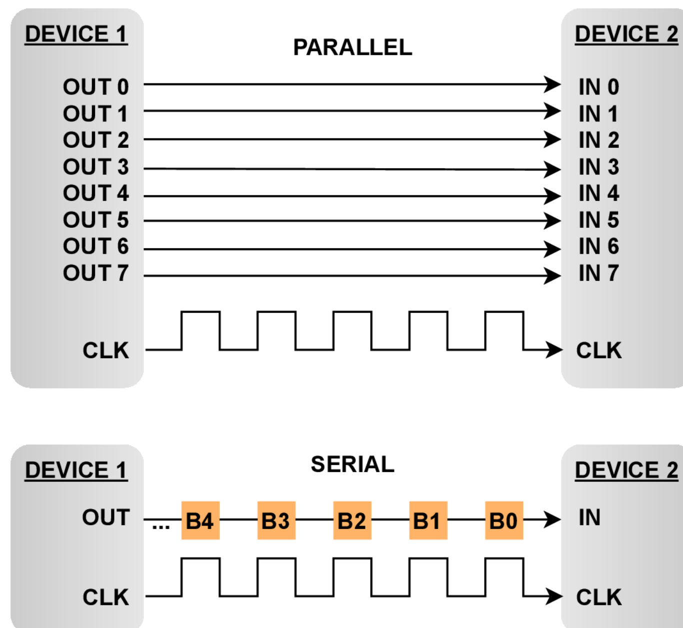
Obr. 2.1: Schéma procesoru SiFive FU540-C000 na najvyššej úrovni [28]

napríklad Linux. Je základom pre platformu HiFive Unleashed Development Platform z rodiny Freedom U500. Model FU540-C000 je kompatibilný so všetkými príslušnými RISC-V štandardami, pričom využíva zbernicovú technológiu **TileLink**, je postavený okolo jadra U54-MC Core Complex a vyrobený prostredníctvom procesu TSMC 28HPC 28 nm. Bloková schéma tohoto procesoru je zobrazená na obrázku 2.1

Procesor obsahuje 64 bitové jadro architektúry RISC-V E51, ktoré disponuje vysokovýkonnou zreťazenou linkou. Táto linka vykonáva inštrukcie v poradí, pričom je schopná udržať výkon jednej inštrukcie za periódu hodinového signálu. Toto jadro podporuje strojový a užívateľský privilegovaný režim, a využíva štandardné rozšírenia M, A a C. Procesor ďalej disponuje štyrmi 64 bitovými jadrami RISC-V U54, ktoré využívajú rovnako výkonnú zreťazenú linku ako uvedené E51. Tieto jadrá podporujú všetky tri privilegované režimy a oproti jadru E51 využívajú navyše štandardné rozšírenia F a D. Funkcionalitu prerušenia zaisťuje štandardný RISC-V radič Platform Level Interrupt Controller, ktorý podporuje 53 rôznych globálnych prerušení so siedmimi úrovňami priority. Softvérové prerušenia a prerušenia od časovaču v strojovom režime zaisťuje radič Core Local Interuptor. Pre asynchrónnu sériovú komunikáciu sú k dispozícii dva moduly typu UART a pre synchrónnu tri moduly SPI, a jeden modul I^2C. Procesor taktiež obsahuje určité množstvo GPIO pinov, ktoré môžu byť použité aj pre generovanie PWM (pulzne šírkovej modulácie). Priemyselný štandard JTAG zaisťuje podporu debuggovania. Dozvedieť sa o ďalších vlastnostiach, prvkoch a perifériách tohoto procesoru je možné na [28].

2.2 Sériová komunikácia

Elektronické systémy, ako napríklad procesory, zvyknú pozostávať z dielčích komponent, ktoré medzi sebou komunikujú prostredníctvom výmeny určitých dát. Aby im bola táto



Obr. 2.2: Diagram paralelnej a sériovej komunikácie [13]

komunikácia umožnená, musia zdieľať nejaký komunikačný protokol, ktorý definuje pravidlá pre vysielajúcu a prijímaciu stranu, viď [13].

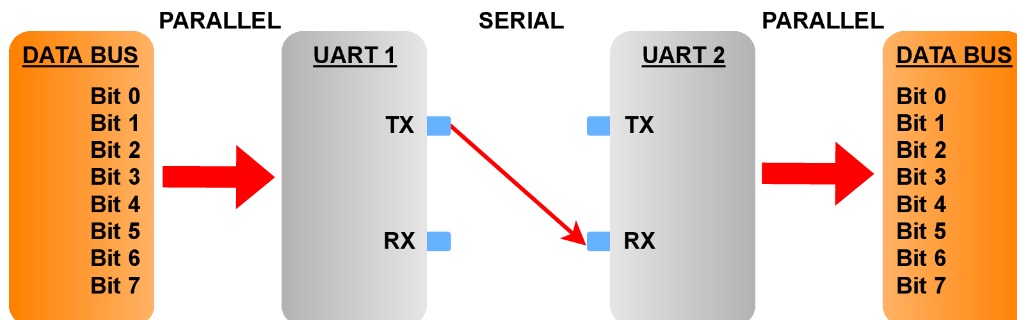
Takýchto protokolov v súčasnosti existuje pomerne veľké množstvo a v zásade je možné ich rozdeliť na dve základné skupiny, a síce protokoly pre **sériovú** a paralelnú komunikáciu. Princíp paralelnej komunikácie spočíva v prenose viacerých bitov v jednom okamihu, na čo sa typicky využívajú zbernice a dosahuje sa tým vysoká rýchlosť prenosu. Oproti tomu je základnou myšlienkou sériovej komunikácie čo najnižší počet vodičov potrebných pre komunikáciu, čo je docielené postupným prenášaním jednotlivých bitov po jedinom vodiči, ako na obrázku 2.2. Sériová komunikácia sa ďalej delí na synchrónnu a **asynchrónnu**.

Pri synchrónnom prenose je spolu s dátami prenášaný aj hodinový signál slúžiaci pre synchronizáciu, pre ktorý musí byť v komunikačnom rozhraní vyčlenený ďalší vodič, viac na [3]. V takejto komunikácii vystupujú účastníci dvoch typov, a síce master a slave. V jednej takejto konfigurácii môže byť iba jedno zariadenie typu master a k nemu pripojené jedno alebo viac zariadení typu slave. Počas komunikácie zariadenie master generuje pre zariadenia slave hodinový signál, ktorého hrany určujú kedy je možné na dátový vodič vystaviť, respektíve snímať z dátového vodiča ďalší bit. Medzi najznámejšie synchrónne sériové rozhrania patria napríklad SPI alebo I^2C.

V asynchrónnej sériovej komunikácii sa už hodinový signál neprenáša. Prijímacie zariadenie si ho preto musí generovať samo s dostatočnou presnosťou a je potrebné, aby pred prenosom dátového rámca došlo k zosynchronizovaniu s vysielacím zariadením. Pre tento dej sa využíva detekcia zmeny logickej hodnoty na dátovom vodiči, ktorá je vopred definovaná v danom komunikačnom protokole. Po úspešnom detekovaní začiatku prenosu dát prijímacia strana vzorkuje dátový vodič v závislosti na danej prenosovej rýchlosti, zatiaľ čo vysielacia strana posiela jednotlivé dátové bity po dátovom vodiči, kde každý bit vystaví na dobu korešpondujúcu s prenosovou rýchlosťou. Po odoslaní celého dátového rámca vysielacie zariadenie nastaví hodnotu dátového vodiču späť na predvolenú hodnotu [31].

2.2.1 Universal Asynchronous Receiver-Transmitter

Univerzálny asynchrónny prijímač-vysielač, ďalej len **UART**, je jedným z najpoužívanejších protokolov pre asynchrónnu sériovú komunikáciu, a teda jeho hlavným účelom je odosielanie a prijímanie dát. Jednu z najväčších výhod tejto technológie predstavuje skutočnosť, že pre komunikáciu medzi dvomi zariadeniami, ktorá môže byť pri správnej konfigurácii **plne duplexná**, postačujú dva vodiče.



Obr. 2.3: Grafické znázornenie komunikácie v technológii UART [7]

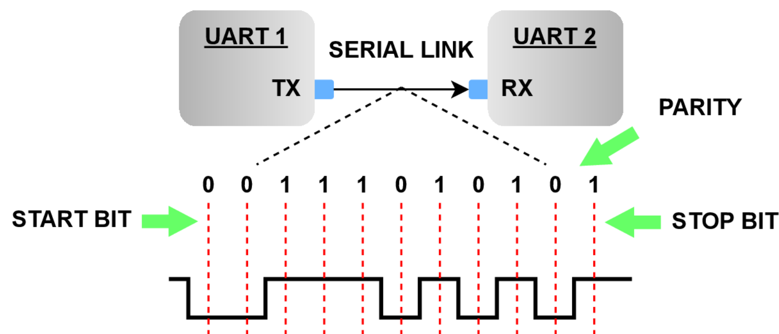
Typickým príkladom použitia je komunikácia medzi dvomi obvodmi typu UART, kedy vysielací obvod konvertuje paralelné dáta, dodané riadiacim zariadením, napríklad procesorom po zbernici, do sériovej podoby a po jednotlivých bitoch ich posiela obvodu druhému, ktorý ich následne prekonvertuje späť do paralelnej podoby a preposiela ich po zbernici prijímajúcemu zariadeniu, viď [6]. Odosielanie a prijímanie dát zabezpečujú moduly vysielač, a prijímač, ktoré sú typicky označované ako **TX**, respektíve **RX**, viď obrázok 2.3. Tieto dva moduly sú na sebe úplne nezávislé, čo umožňuje už spomínanú plne duplexnú komunikáciu.

Keďže sa jedná o asynchrónny typ komunikácie, čo znamená neprítomnosť hodinového signálu, ktorý by synchronizoval komunikujúce obvody, je potrebné, aby boli vyriešené viaceré aspekty synchronizácie medzi obomi komunikujúcimi stranami.

Prvý takýto aspekt predstavuje potreba detekcie začiatku prenosu dát v prijímači. Tento problém je riešený prechodom signálu z kľudovej úrovne, ktorá býva väčšinou definovaná ako logická 1, do úrovne opačnej, a teda logickej 0. V praxi to znamená, že pokiaľ sa práve neprenášajú dáta, tak je na vodiči sériovej linky nastavená logická 1. Akonáhle sa chystá vysielač odoslať nejaké dáta, nastaví hodnotu signálu do logickej 0 na presne určenú dobu, respektíve pridá pred dátové bity synchronizačný bit, ktorý sa nazýva aj **štart bit**. V momente, kedy prijímač detekuje túto udalosť na svojom vstupe, môže začať vzorkovať prichádzajúce dátové bity.

Ďalším, podobným problémom, ktorý sa týka synchronizácie, je nutnosť detekcie ukončenia prenosu dát v prijímači. Riešením je opäť manipulácia s hodnotou signálu na sériovom vodiči. Po odoslaní posledného dátového bitu vysielač nastaví signál na presne určenú dobu do logickej 1, čím pridáva k odoslaným dátovým bitom jeden alebo viac takzvaných **stop bitov**. Spravidla sa zvyknú používať jeden alebo dva stop bity. Tieto synchronizačné bity slúžia pre prijímač nielen ako oddeľovače prenášaných dátových slov, ale poskytujú mu tiež nástroj pre zosynchronizovanie hodinového signálu s vysielacou stranou, viac v [3].

Dátový rámec používaný perifériou UART teda väčšinou pozostáva z jedného štart bitu, skupiny dátových bitov, vo väčšine prípadov vo veľkosti jedného bytu, a jedného alebo dvoch stop bitov, ako ukazuje obrázok 2.4. Rámec tiež môže obsahovať **paritný bit**, ktorý je voliteľný. Tento bit sa vkladá medzi dátové bity a stop bity, a slúži pre prijímač ako



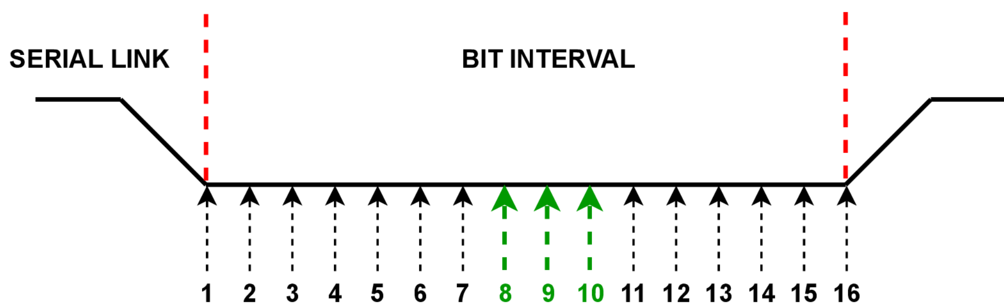
Obr. 2.4: Ukážkový dátový rámec sériového protokolu UART [7]

indikátor poškodenia prijatých dát v dôsledku elektromagnetického žiarenia, nekorektne nastavenej prenosovej rýchlosti apod.

Pre správne fungovanie uvedených synchronizačných techník je potrebné, aby bola pre vysielač aj prijímač korektne nakonfigurovaná prenosová rýchlosť, nazývaná tiež **baud rate**. Vysielač podľa nej odosiela jednotlivé bity a prijímač v závislosti na nej vzorkuje bity na svojom vstupe. Táto rýchlosť sa v kontexte asynchrónnej sériovej komunikácie udáva v bitoch za sekundu.

Pre udržovanie prijatých dát alebo dát určených na odoslanie v technológii UART sa väčšinou využívajú dva rôzne prístupy. Prvý predstavuje jednoduché úložisko o veľkosti jedného dátového slova. Nadradená komponenta môže pre získanie informácie o uvoľnení odosielacieho bufferu, respektíve naplnení prijímacieho bufferu, využívať techniku, ktorá sa nazýva polling. Predstavuje konštantnú slučku, z ktorej program vyskočí, až keď požadovaná udalosť nastane. Miesto pollingu sa môžu používať aj prerušenia. Druhý prístup využíva ako úložisko dátovú štruktúru typu fronta, ďalej len **FIFO**, o určitej veľkosti. V tomto prípade je možné pripraviť viacero dátových slov na odoslanie, respektíve prijať viacero dátových slov a spracovať ich zároveň. V kombinácii s FIFO sa využívajú tzv. programovateľné **watermark** prerušenia, ktoré sú vyvolané, ak je v danej FIFO počet záznamov väčší, respektíve menší, než zadaný prah, ktorý je možné konfigurovať.

Prijímacia časť spravidla využíva nejakú techniku vzorkovania sériovej linky za účelom odstránenia určitej miery šumu zo vstupného signálu. Príkladom takejto techniky je **16 × oversampling with 2/3 majority voting per bit**, viď [16]. Pri použití tejto techniky je každý bit za dobu danú prenosovou rýchlosťou navzorkovaný šestnásťkrát a jeho výslednú hodnotu určuje väčšinová hodnota spomedzi ôsmej, deviatej a desiatej vzorky. Táto technika je graficky znázornená na obrázku 2.5.



Obr. 2.5: Vzorkovanie technikou 16 × oversampling with 2/3 majority voting per bit

Medzi výhody tejto perférie patrí predovšetkým jej široké zastúpenie v rôznych technológiách a z toho prameniaca kvalitná dokumentácia, nízky počet vodičov potrebných pre komunikáciu a taktiež absencia hodinového synchronizačného signálu. Nevýhodu predstavuje predovšetkým nízka priepustnosť sériovej linky. Ďalšou nevýhodou je chýbajúca podpora pre takzvané multiple slave a multiple master systémy [7].

2.2.2 Periféria UART pre procesor SiFive FU540-C000

Procesor FU540-C000 od spoločnosti SiFive obsahuje dve inštancie periférie UART, ktorých parametre sú uvedené v tabuľke 2.2. Táto podkapitola čerpá z [28].

Instance number	Address	div_width	div_init	TX FIFO DEPTH	RX FIFO DEPTH
0	0x10010000	16	4339 (6509)	8	8
1	0x10011000	16	4339 (6509)	8	8

Tabuľka 2.2: Parametre periférií UART procesoru SiFive FU540-C000

Táto periféria podporuje formáty **8-N-1**, respektíve **8-N-2**, čo predstavuje dátový rámec v podobe ôsmich dátových bitov, absenciu paritného bitu a jeden alebo dva stop bity. Vysielač aj prijímač využívajú pre neodoslané, respektíve nespracované dáta buffer typu FIFO, ktorý podporuje programovateľné watermark prerušenia. Obe tieto fronty majú veľkosť ôsmich záznamov. Prijímacia časť využíva pre vzorkovanie sériovej linky techniku 16 × oversampling with 2/3 majority voting per bit. Modul UART sa niekedy využíva aj pre riadenie toku v hardvéri alebo ako podpora pre riadiace signály v modemoch, táto periféria však dané funkcie nepodporuje.

Offset	Name	Description
0x00	txdata	Transmit data register
0x04	rxdata	Receive data register
0x08	txctrl	Transmit control register
0x0C	rxctrl	Receive control register
0x10	ie	UART interrupt enable
0x14	ip	UART interrupt pending
0x18	div	Baud rate divisor

Tabuľka 2.3: Adresový priestor riadiacich registrov periférie UART

Adresový priestor riadiacich 32 bitových registrov tejto periférie je zobrazený v tabuľke 2.3 a ich funkcia bude popísaná v ďalších častiach tejto podkapitoly.

Transmit Data Register

Pri zápise do tohoto registru sa daný byte uložený v políčku *data* zafrontuje do vysielacej FIFO, pokiaľ je FIFO schopná prijať ďalší záznam. Návratová hodnota po prečítaní tohoto registru obsahuje aktuálnu hodnotu príznaku *full* a samé nuly v políčku *data*, pričom príznak *full* definuje schopnosť FIFO prijímať ďalšie záznamy. Pokiaľ je nastavený, tak sú všetky zápisy ignorované. Register je zobrazený v tabuľke 2.4.

Transmit Data Register (txdata)				
Register Offset	0x0			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	data	RW	X	Transmit data
[30:8]	Reserved			
31	full	RO	X	Transmit FIFO full

Tabuľka 2.4: Register Transmit Data

Receive Data Register

Čítanie z tohoto registru spôsobí odstránenie záznamu zo začiatku FIFO a jeho vrátenie v políčku *data*, viď tabuľku 2.5. Príznak *empty* určuje, či bola FIFO pred čítaním prázdna. Ak je nastavený, tak políčko *data* neobsahuje validné informácie. Zápisy do tohoto registra sú ignorované.

Receive Data Register (rxdata)				
Register Offset	0x4			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[7:0]	data	RO	X	Receive data
[30:8]	Reserved			
31	empty	RO	X	Receive FIFO full

Tabuľka 2.5: Register Receive Data

Transmit Control Register

Tento register, zobrazený v tabuľke 2.6, riadi operácie vysielacieho kanálu a pri resetovaní sa kompletne nuluje. Ak je príznak *txen* nastavený, znamená to, že kanál je aktívny. Pokiaľ nie je nastavený, tak je vysielacia činnosť pozastavená a na vodič sériovej linky je privedená hodnota logickej jednotky. Políčko *nstop* definuje počet stop bitov, konkrétne 0 pre jeden stop bit a 1 pre 2 stop bity. Počet záznamov vo FIFO spúšťajúci prerušenie sa ukladá do políčka *txcnt*.

Transmit Control Register (txctrl)				
Register Offset	0x8			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	txen	RW	0x0	Transmit enable
1	nstop	RW	0x0	Number of stop bits
[15:2]	Reserved			
[18:16]	txcnt	RW	0x0	Transmit watermark level
[31:19]	Reserved			

Tabuľka 2.6: Register Transmit Control

Receive Control Register

Úlohou tohoto registru je riadenie operácií prijímacieho kanálu a pri resetovaní sa kompletne vynuluje. Nastavený príznak *rxen* definuje, že je prijímací kanál aktívny. Pokiaľ nie je nastavený, vzorkovanie sériovej linky a ukladanie spracovaných dát do FIFO je pozastavené. Políčko *rxcnt* určuje počet záznamov vo FIFO potrebný pre vyvolanie prerušenia. Register je zobrazený v tabuľke 2.7.

Receive Control Register (rxctrl)				
Register Offset	0xC			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	rxen	RW	0x0	Receive enable
[15:1]	Reserved			
[18:16]	rxcnt	RW	0x0	Receive watermark level
[31:19]	Reserved			

Tabuľka 2.7: Register Receive Control

Registre Interrupt Enable a Interrupt Pending

Register Interrupt Pending, zobrazený v tabuľke 2.8, slúži len na čítanie a indikuje, či je vo vysielacej alebo prijímacej FIFO splnená podmienka, ktorá vyvoláva prerušenie. Register Interrupt Enable určuje, či sú prerušenia vo vysielači, respektíve prijímači povolené a pri resetovaní je vynulovaný, viď tabuľku 2.9. Pri povolenom prerušení sa príznak *txwmp* v registri Interrupt Pending nastaví v momente, kedy je počet záznamov vo vysielacej FIFO ostro menší než počet daný políčkom *txcnt* v registri Transmit Control. Obdobne funguje príznak *rxwmp* s tým rozdielom, že počet záznamov v prijímacej FIFO musí byť ostro väčší ako hodnota políčka *rxcnt* v registri Receive Control.

UART Interrupt Pending Register (ip)				
Register Offset	0x14			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	txwmp	RO	X	Transmit watermark interrupt pending
1	rxwmp	RO	X	Receive watermark interrupt pending
[31:2]	Reserved			

Tabuľka 2.8: Register Interrupt Pending

UART Interrupt Enable Register (ie)				
Register Offset	0x10			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
0	txwme	RW	0x0	Transmit watermark interrupt enable
1	rxwme	RW	0x0	Receive watermark interrupt enable
[31:2]	Reserved			

Tabuľka 2.9: Register Interrupt Enable

Baud Rate Divisor Register (div)				
Register Offset	0x18			
Bits	**Field Name**	**Attr.**	**Rst.**	**Description**
[15:0]	div	RW	X	Baud rate divisor is div_width bits wide, and the reset value is div_init.
[31:16]	Reserved			

Tabuľka 2.10: Register Baud Rate Divisor

Baud Rate Divisor Register

Hodnota políčka *div* v tomto registri, vid tabuľku 2.10, špecifikuje akou hodnotou sa má deliť frekvencia hodinového signálu tejto periférie. Tento proces slúži na generovanie prenosovej rýchlosti pre vysielací aj prijímací kanál. Vzťah medzi frekvenciou hodinového signálu f_{in} a výslednou prenosovou rýchlosťou f_{baud} je uvedený v rovnici 2.1.

$$f_{baud} = \frac{f_{in}}{div + 1} \tag{2.1}$$

Tabuľka 2.11 ukazuje prehľad hodnôt, ktoré sa používajú pre generovanie zavedených prenosových rýchlostí. Uvedené deliace pomery rešpektujú predchádzajúcu rovnicu, takže sú o jedna väčšie než hodnoty uložené v políčku *div*.

tlclk (MHz)	Target Baud (Hz)	Divisor	Actual Baud (Hz)	Error (%)
500	31250	16000	31250	0
500	115200	4340	115207	0.0064
500	250000	2000	250000	0
500	1843200	271	1845018	0.099
750	31250	24000	31250	0
750	115200	6510	115207	0.0064
750	250000	3000	250000	0
750	1843200	407	1842751	0.024

Tabuľka 2.11: Často využívané prenosové rýchlosti a hodnoty pre ich generovanie

2.3 Návrh číslicových obvodov

V minulosti prebiehal návrh číslicových obvodov rýdzo manuálne, kedy návrhár vypracoval podľa špecifikácie schému daného obvodu, viď [25]. Podľa spomenutej schémy bol potom navrhnutý obvod fyzicky vyrobený a jeho správna funkčnosť vzhľadom k špecifikácii bola otestovaná až následne. Pokiaľ bola do návrhu pri jeho vypracovávaní z ľubovoľného dôvodu zanesená chyba, bolo potrebné návrh prepracovať, prípadne vytvoriť úplne nový a vyrobený kus hardvéru zostal nepoužiteľný.

Čím viac narastala hustota integrácie informačných technológií, a teda aj číslicových obvodov, tým viac pociťovali spoločnosti zaoberajúce sa výrobou číslicových obvodov potrebu znižovať výrobné náklady a urýchľovať produkciu. To im však značne znemožňovali vyššie uvedené problémy. Táto situácia nakoniec vyústila ku vzniku takzvaných **HDL** jazykov (Hardware Description Language), ktoré slúžia na modelovanie navrhovaných obvodov,

konkrétne popisom ich funkcie namiesto schémy. S využitím niektorého jazyka z rodiny HDL si môže návrhár vytvoriť návrh obvodu vo svojom počítači, následne v ňom odsimulovať jeho činnosť a až po odladení všetkých chýb odoslať navrhnutý obvod do výroby. Zároveň pribudla návrhárom možnosť dekomponovať zložitý systém na jednoduchšie podsystémy a venovať sa ich návrhu jednotlivo, prípadne rozdeliť prácu medzi viac návrhárov. Táto dekompozícia môže siahať v podstate až na úroveň tranzistorov.

V súčasnosti prebieha návrh číslicových obvodov nasledovne. Na začiatku procesu dodáva zákazník zoznam požiadaviek, respektíve špecifikáciu cieľového obvodu. Návrhár na základe dôkladnej analýzy tejto špecifikácie vypracuje návrh, v ktorom zvolí cieľovú technológiu obvodu, programovací jazyk z rodiny HDL, pomocou ktorého prebehne implementácia modelu obvodu, a taktiež návrh samotnej implementácie. Po dokončení implementácie nasleduje proces verifikácie obvodu, ktorý bude detailnejšie popísaný na ďalších stranách tejto kapitoly. Počas verifikácie sa overuje správnosť fungovania daného obvodu vzhľadom k jeho špecifikácii a v prípade odhalenia nejakej nekonzistencie sa návrhár vracia k niektorej z predchádzajúcich fáz, a reflektuje svoje zistenia prepracovaním návrhu, respektíve špecifikácie. Po úspešnom ukončení verifikačného procesu nasleduje syntéza obvodu, ktorá predstavuje transformáciu popisu obvodu v niektorom z HDL jazykov do zoznamu komponent cieľovej technológie, ich vzájomného prepojenia a detailného geometrického rozloženia, viac o syntéze na [15]. Ďalšími výstupmi zo syntézy sú rôzne parametre cieľového obvodu ako časovanie, oneskorenie, veľkosť plochy na čipe, ktorú obvod zaberá apod. Následne môže byť obvod fyzicky vyrobený a nasadený do prevádzky.

Medzi najpoužívanejšie cieľové technológie sa radia predovšetkým ASIC (Application Specific Integrated Circuit) a **FPGA** (Field Programmable Gate Array). Technológia ASIC predstavuje integrovaný obvod, ktorý je špeciálne navrhnutý pre vykonávanie konkrétnej funkcie. Táto skutočnosť umožňuje obvod veľmi dôkladne optimalizovať z hľadiska rýchlosti, veľkosti plochy na čipe, spotreby energie a ďalších parametrov. Obvody typu ASIC sú však relatívne drahé a ich ďalším problémom je, že akonáhle je takýto obvod vyrobený a vyskytne sa požiadavka na rozšírenie alebo zmenu jeho funkcionality, je potrebné ho vyrobiť odznova. Oproti tomu, technológia FPGA ponúka možnosť rekonfigurácie obvodu, jeho parametre však zvyknú byť spravidla horšie ako pri obvodoch typu ASIC. Čipy FPGA pozostávajú z konfigurovateľných logických blokov, ktorých funkciu modifikuje návrhár, vstavaných blokov ako napríklad pamäte typu Block RAM, ktoré môže návrhár využiť, a z prepojovacej siete, ktorá zaberá väčšinu plochy na čipe. Po syntéze popisu obvodu v niektorom jazyku z rodiny HDL je do FPGA čipu nahraný takzvaný konfiguračný reťazec, ktorý definuje funkciu jednotlivých využitých blokov, využitie vstavaných blokov a ich vzájomné prepojenie pomocou prepojovacej siete [14].

Pre popis návrhov číslicových obvodov sa v sučasnosti používajú rôzne jazyky z rodiny HDL. Medzi najpoužívanejšie patria jazyky **VHDL**, Verilog, **SystemVerilog** a SystemC. Keďže jazyk VHDL bol zvolený pre implementáciu návrhu cieľového obvodu tejto práce, jeho vlastnosti budú uvedené nižšie. Pre následnú verifikáciu daného obvodu bude použitý jazyk SystemVerilog, ktorý bude popísaný v ďalšej časti tejto kapitoly.

2.3.1 Jazyk VHDL

VHSIC Hardware Description Language, kde skratka VHISC znamená Very High Speed Integrated Circuit, je programovací jazyk určený pre popis hadrvérových obvodov. Vznikol v Spojených štátoch amerických, kde na začiatku slúžil pre vojenské účely a po prvý raz bol štandardizovaný v roku 1987 inštitúciou IEEE, kedy šlo o verziu IEEE 1076-1987,

a vychádzal vtedy z jazyka ADA. Prostredníctvom tohoto jazyku je možné vytvárať návrhy obvodov vo veľmi širokom intervale zložitosti, od obvodov s len niekoľkými hradlami až po veľmi komplexné viacúrovňové systémy. Je taktiež vo výraznej miere podporovaný rôznymi simulačnými a syntéznymi nástrojmi. Ponúka možnosť modelovať súbežné deje pomocou komponent, ktoré paralelne vykonávajú svoju funkciu. Ďalej umožňuje vytvárať špecifické, častokrát aj skutočne komplexné dátové typy a zahŕňa tiež koncept času, a schopnosť modelovať niektoré elektrické vlastnosti. Návrh číslicového obvodu v jazyku VHDL je väčšinou realizovaný na úrovni medziregistrových prenosov. Pre popis obvodu alebo jeho časti sú využívané takzvané komponenty, ktoré môžu byť vzájomne poprepájané a komunikovať medzi sebou, prípadne s okolím. Takéto komponenty sú definované návrhovými jednotkami jazyka VHDL, z ktorých najdôležitejšie budú popísané nižšie v tejto podkapitole. Pre viac informácií o programovacom jazyku VHDL viď [17].

Entita je primárnou návrhovou jednotkou, ktorá je určená k deklarácii rozhrania medzi komponentou a jej okolím. Nepopisuje však chovanie danej komponenty. Entita môže obsahovať viacero druhov deklarácii, kde medzi najdôležitejšie patria generické parametre a porty. Generické parametre slúžia pre parametrizáciu komponenty, kedy je možné napríklad určiť veľkosť pamäte, ktorú komponenta obsahuje, až pri inštanciácii danej komponenty. Porty sa používajú pre definíciu samotného rozhrania komponenty. Jedná sa o signály, ktoré môžu byť v rôznom režime smeru dát, ktoré cez ne prúdia. Najpoužívanejšie režimy portových signálov sú vstupný, výstupný a vstupno-výstupný, ktorý je typický pre zbernicové operácie. Tieto deklarácie sú nepovinné, takže entita prakticky nemusí mať žiadne rozhranie a jedná sa potom o uzavretý systém. Entita môže tiež obsahovať ďalšie deklarácie pre dátové typy, konštanty, funkcie, procedúry apod.

Ďalšou návrhovou jednotkou jazyku VHDL je **architektúra**. Používa sa pre popis funkcie komponenty, respektíve jej chovania alebo štruktúry. Ide o sekundárnu návrhovú jednotku, čo znamená, že musí byť zviazaná s nejakou entitou, pričom entita môže byť zviazaná s ľubovoľným počtom architektúr. Architektúra pozostáva z deklaračnej a operačnej časti, kedy deklaračná časť môže obsahovať deklarácie dátových typov a tiež ďalších programových konštrukcií jazyka VHDL, podobne ako pri entite. Operačná časť potom obsahuje popis chovania komponenty, pričom toto chovanie je možné definovať tromi rôznymi spôsobmi, a síce **behaviorálnym** popisom, **dataflow** popisom a **štrukturálnym** popisom. Behaviorálny popis definuje chovanie komponenty programovou štruktúrou nazývanou **proces**. Takýto proces určuje hodnoty výstupných signálov komponenty na základe zmien hodnôt jej vstupných signálov. Jedným procesom je možné definovať celú komponentu, prípadne rozdeliť jej logicky súvisiace celky medzi viacero procesov, ktorých činnosť prebieha paralelne, zatiaľ čo príkazy v ich tele sú vykonávané sekvenčne. Popis typu dataflow definuje komponentu modelovaním dátových závislosti v jej vnútri, ktoré popisuje paralelne sa vykonávajúcimi príkazmi. Napokon štrukturálny popis využíva pre definovanie chovania komponenty inštanciáciu jej subkomponent a ich prepojenie pre vytvorenie hierarchického systému. Všetky uvedené typy popisu fungovania komponenty je možné medzi sebou navzájom ľubovoľne kombinovať.

Jazyk VHDL poskytuje taktiež istú formu testovania navrhovaných obvodov. Takzvaný **testbench** je možné vytvoriť ako prázdnu entitu s architektúrou, v ktorej operačnej časti je inštanciovaná daná komponenta a generované testovacie vstupy, ktoré sú privádzané na rozhranie tejto komponenty s časovaním podľa potreby. Priebeh takejto simulácie je možné sledovať v niektorom zo simulačných nástrojov.

2.4 Funkčná verifikácia číslicových obvodov

Pri každom vyvíjanom systéme je tou najdôležitejšou požiadavkou, aby svoju funkciu vykonával bezchybne a presne podľa špecifikácie. Inak tomu nie je ani pri číslicových obvodoch. Ako už bolo uvedené, odhalenie chýb v návrhu obvodu je kritické ešte pred jeho fyzickým vyrobením a technika, ktorá sa pre toto odhaľovanie používa, sa nazýva verifikácia. Táto technika môže mať rôzne prístupy, v tejto práci však bude kladený dôraz predovšetkým na verifikáciu **funkčnú**, viď [22].

Funkčná verifikácia sa radí k dynamickým verifikačným metódam, ktoré pre overovanie funkcie obvodu využívajú simuláciu v niektorom zo simulačných nástrojov. Jej základným princípom je vytvorenie verifikačného prostredia, ktoré simuluje prostredie budúceho nasadenia verifikovaného obvodu. V tomto prostredí sú potom počas simulácie generované testovacie stimuly, ktoré sú privádzané na vstupy verifikovaného obvodu, viď obrázok 2.6. Odozva obvodu na tieto vstupy je zachytávaná a kontrolovaná. Výhodou takéhoto druhu verifikácie je hlavne jednoduchosť a široká podpora zo strany simulačných nástrojov. Simulácia je však výpočtovo náročná úloha, ktorej zložitosť rastie s komplexnosťou verifikovaného obvodu. Funkčná verifikácia preto používa v kombinácii so simuláciou aj rôzne prídavné techniky pre zlepšenie jej efektivity, ktoré budú popísané v ďalších častiach tejto podkapitoly.

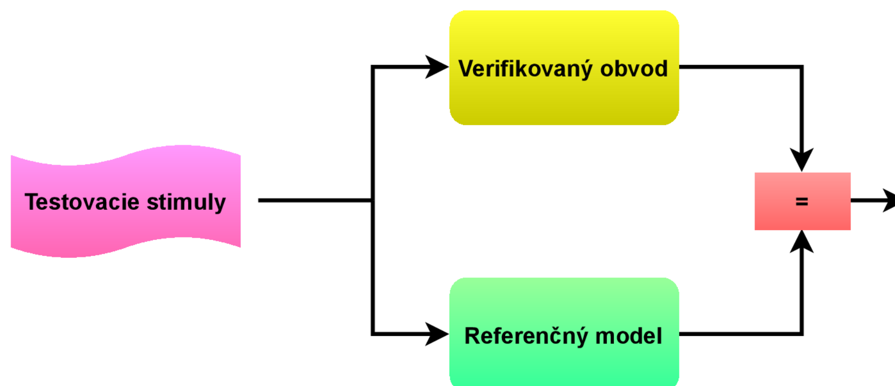
Obr. 2.6: Schéma zapojenia verifikačného prostredia a verifikovaného obvodu

2.4.1 Samokontrolné mechanizmy

Vďaka tejto technike sa stáva kontrola odozvy verifikovaného obvodu na testovacie stimuly automatizovanou. V praxi sa spravidla používajú dva základné druhy samokontrolných mechanizmov [34].

Prvým sú referenčné vektory, ktoré predstavujú zoznam očakávaných hodnôt na výstupe obvodu v závislosti na hodnotách vstupných stimulov. Tieto vektory musia byť vytvorené ešte pred spustením simulácie, aby počas nej mohli byť porovnávané s výstupmi verifikovaného obvodu. Vytváranie takýchto vektorov je zdĺhavý a náročný proces, a pokiaľ dochádza k častým zmenám v špecifikácii alebo návrhu obvodu, je potrebné ich vytvárať odznova, čo výrazne znižuje efektivitu takéhoto prístupu.

Ďalším druhom samokontrolného mechanizmu je referenčný model, označovaný aj golden model, ktorý implementuje chovanie verifikovaného obvodu podľa tej istej špecifikácie. Proces automatickej kontroly výstupov verifikovaného obvodu potom prebieha tak, že sú dané testovacie vstupy privedené zároveň na vstup obvodu a referenčného modelu, a ich výstupy sú následne porovnané, ako na obrázku 2.7. Výhodou tohto prístupu oproti refe-

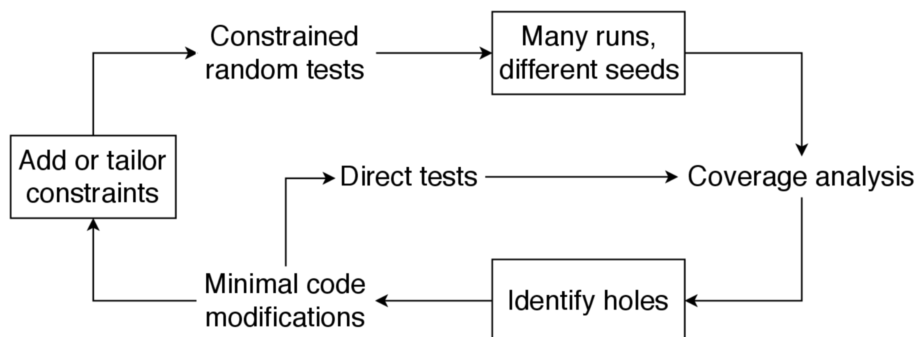


Obr. 2.7: Kontrola odozvy verifikovaného obvodu pomocou referenčného modelu

renčným vektorom je jednoduchšia prispôsobiteľnosť, celková implementácia však môže byť pomerne náročná.

2.4.2 Funkčná verifikácia riadená pokrytím

Jednou z najvýraznejších techník funkčnej verifikácie je verifikácia riadená **pokrytím**, poskytujúca rôzne metriky. Táto technika za behu simulácie obvodu poskytuje spätnú väzbu, či verifikácia preverila všetky vlastnosti, respektíve stavy verifikovaného systému v definovanom pokrytí, alebo je potrebné vo verifikácii pokračovať. Analýza výsledného pokrytia následne slúži na úpravu, prípadne vytvorenie nových verifikačných testov, ktoré pomôžu dosiahnuť vyššie pokrytie, viď obrázok 2.8.



Obr. 2.8: Proces úpravy verifikačných testov na základe analýzy metrík pokrytia [33]

Jedným z typov pokrytia je **funkčné** pokrytie, ktoré je zamerané na funkcionalitu a chovanie systému [33]. Verifikačné testy v tomto prípade typicky využívajú pseudonáhodné generovanie stimulov, ale pre niektoré oblasti systémov sú vhodnejšie testy priame. Jedná sa o zameranie na hodnoty jednotlivých signálov na rozhraní verifikovaného obvodu, ale predovšetkým sa zvyknú pokrývať komplexnejšie vlastnosti, ako napríklad kombinácie hodnôt viacerých signálov, či postupnosť hodnôt jednotlivých signálov apod. Jazyk SystemVerilog poskytuje pre definovanie modelu pokrytia konštrukciu *covergroup*, ktorá pozostáva z bodov pokrytia, respektíve *coverpointov* [30]. Tieto body obsahujú jeden alebo viac *binov*, ktoré definujú hodnoty, množiny hodnôt, alebo sekvencie hodnôt na daných signáloch, ktoré majú byť počas funkčnej verifikácie pokryté. Takto definované pokrytie, respektíve jeho body je

následne možné sledovať v simulačnom nástroji, ktorí poskytuje informácie, či a koľkokrát boli jednotlivé body počas simulácie navštívené.

Ďalším typom pokrytia je pokrytie **kódu**. Poskytuje ďalšiu metriku na posúdenie úplnosti overenia verifikovaného systému prostredníctvom merania podielu kódu vykonaného počas simulácie. Zahŕňa to niekoľko metrík, ako napríklad koľko riadkov kódu sa vykonalo, pokrytie ciest, ktoré cesty boli prostredníctvom kódu preverené, pokrytie výrazov v kóde, pokrytie stavov konečného automatu atď. Množstvo simulačných nástrojov v súčasnosti poskytuje prostriedky pre zbieranie a sledovanie štatistík tohto typu pokrytia, takže odpadáva nutnosť vytvárania ďalšieho zdrojového kódu za týmto účelom.

Existujú aj ďalšie typy pokrytia, v tejto práci bude však ako primárna spätná väzba použité pokrytie funkčné v kombinácii s overovaním pokrytia kódu. Pokrytie bude dôležité pre vyhodnocovanie efektivity navrhnutého verifikačného prostredia, respektíve použitých testov.

2.4.3 Pseudonáhodné testy

Pri tvorbe verifikačných testov sa verifikační inžinieri predovšetkým orientujú na špecifikáciu obvodu a navrhujú testy tak, aby zacielili na jednotlivé zložky funkcionality obvodu. Takýmto testom sa hovorí testy **priame** a majú dve hlavné nevýhody. S rastúcou komplexnosťou verifikovaného systému je potrebné vytvárať viac takýchto testov pre dosiahnutie požadovanej úrovne pokrytia. Druhou nevýhodou je možnosť prehliadnutia chyby, ktorá nastane kvôli nepredvídanej kombinácii vstupných stimulov, na ktorú verifikační inžinieri svoje priame testy necielili.

Postupom času teda prestalo byť používanie priamych testov pre verifikáciu postačujúce a to viedlo k vzniku takzvaných **pseudonáhodných testov**, viď [22]. Úlohou týchto testov je generovanie pseudonáhodných vstupných stimulov, na ktoré sú uplatňované rôzne obmedzovacie podmienky, ako napríklad iba určitý rozsah adries na vstupe hodnoty adresy do adresového dekodéru apod. Takéto testy sú rýchlejšie na implementáciu a vďaka nim odpadá potreba implementácie relatívne veľkého množstva priamych testov, pretože pseudonáhodné testy rýchlejšie pokrývajú stavový priestor verifikovaného obvodu. Prinášajú však problém redundancie vstupných stimulov, ktorá odďaľuje dosiahnutie požadovaného pokrytia, preto sa pre verifikáciu používajú väčšinou pseudonáhodné testy v kombinácii s priamymi testami zacielenými na okrajové stavy.

2.4.4 Jazyk SystemVerilog

Tento programovací jazyk sa používa pre návrh číslicových obvodov, ponúka však tiež rôzne prostriedky a programové konštrukcie, ktoré môžu byť využité na verifikáciu týchto obvodov. Patrí teda aj do rodiny jazykov **HVL** (Hardware Verification Language).

V minulosti dospel návrh hardvéru do takého bodu, kedy už pre verifikáciu navrhovaných obvodov vtedajšie prostriedky prestali postačovať. Ďalší problém pre verifikačných inžinierov predstavoval fakt, že aj keď boli používané prostriedky relatívne dostačujúce, tak boli buď platené alebo ich bolo možné použiť iba na verifikáciu. Verifikační inžinieri si teda museli osvojovať viacero programovacích jazykov zároveň. Následne ako odpoveď na uvedené problémy vznikol jazyk SystemVerilog, ktorý sa medzitým stal už priemyselným štandardom.

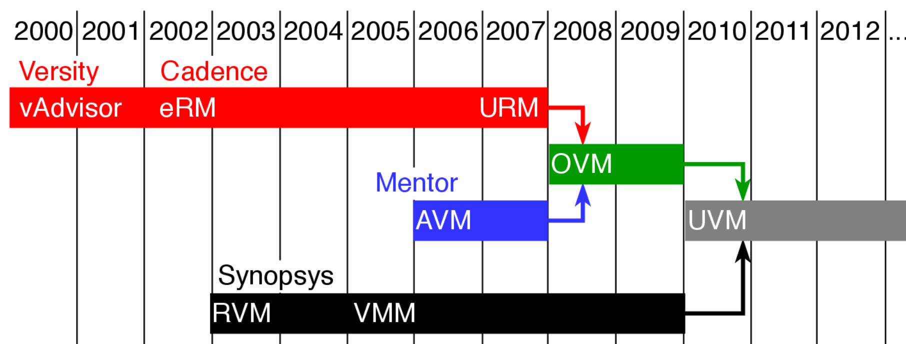
Programovací jazyk SystemVerilog predstavuje nadstavbu jazyka Verilog, do ktorého boli pridané rôzne rozšírenia v podobe abstraktných jazykových konštrukcií, zjednodušujúcich verifikačný proces. Jedným z najvýznamnejších rozšírení sú triedy, vďaka ktorým je

umožnené v jazyku SystemVerilog vyžívať princípy objektovo orientovaného programovania. Tieto triedy predstavujú šablóny, na základe ktorých sa vytvárajú objekty. Systém správy pamäte využíva techniku garbage collector, kedy sú, ako pri iných programovacích objektovo orientovaných jazykoch, objekty odstránené v momente, keď už na ne neexistuje žiadna referencia [21]. Každá trieda definuje premenné, prostredníctvom ktorých si jej objekty udržujú vnútorný stav, a metódy, ktoré objekty využívajú pre manipuláciu nad dátami alebo rôzne iné akcie. Metódy sa ďalej delia na tasky a funkcie, pričom tasky je možné parametrizovať dĺžkou ich trvania v simulačnom čase, zatiaľ čo funkcie sa vykonávaju v nulovom simulačnom čase. Špeciálnym prípadom objektu v jazyku SystemVerilog je **modul**, ktorý predstavuje komponentu na najvyššej úrovni verifikačného prostredia. Je oproti objektom vytvorených z tried statický, pretože počas simulácie nevzniká ani nezaniká. Musí byť prítomný ešte pred začiatkom simulácie a inštanciácie všetkých ostatných objektov sú invokované práve z neho. Okrem toho je možné v prípade, že prostriedky jazyka SystemVerilog nie sú dostačujúce, využiť rozhranie DPI (Direct Programming Interface) a prostredníctvom neho volať funkcie implementované v iných jazykoch. Viac informácií o programovacom jazyku SystemVerilog sa nachádza na [30].

Možnosť využívať jeden a ten istý programovací jazyk pre návrh aj verifikáciu hardvérových obvodov, prítomnosť tried a objektovo orientovaného prístupu sú najvýraznejšie prednosti jazyka SystemVerilog. Vďaka týmto vlastnostiam sa postupom času prostredníctvom tohoto jazyka začali vytvárať komplexné, jednoducho prenositeľné a rozšíriteľné verifikačné prostredia. Vznikli tiež celé verifikačné metodiky, z ktorých jedna bude použitá aj v tejto práci, a preto je popísaná v ďalšom texte tejto podkapitoly.

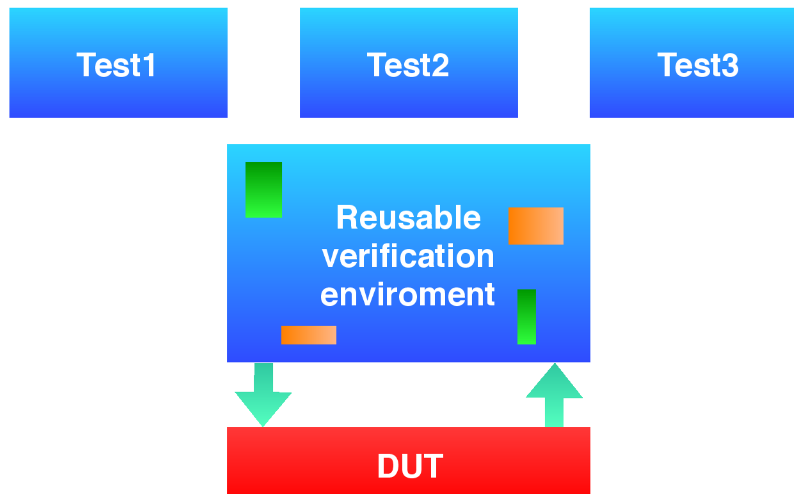
2.4.5 Universal Verification Methodology

Verifikačné metodiky priniesli a stále prinášajú výrazné vylepšenia do procesu funkčnej verifikácie predovšetkým z hľadiska znovupoužiteľnosti. Ich hlavným cieľom je poskytovanie knižníc základných a rozšírených tried pre tvorbu znovupoužiteľných a jednoducho rozšíriteľných verifikačných prostredí pre funkčnú verifikáciu. Vývoj takýchto metodík naprieč časom je zobrazený na obrázku 2.9. V tejto podkapitole bude podrobne popísaná iba metodika **Universal Verification Methodology**, ďalej označovaná len ako **UVM**.



Obr. 2.9: Vývoj verifikačných metodík naprieč časom [33]

Verifikačná metodika UVM bola štandardizovaná spoločnosťou Accellera [2]. Je to mechanizmus, ktorý umožňuje vytváranie verifikačných prostredí pre návrhy obvodov naprogramovaných v jazykoch SystemVerilog, Verilog, VHDL alebo SystemC. Samotné UVM predstavuje knižnicu základných tried jazyka SystemVerilog a je to prvý štandard, ktorý začal



Obr. 2.10: Oddelenie testov od verifikačného prostredia v UVM [9]

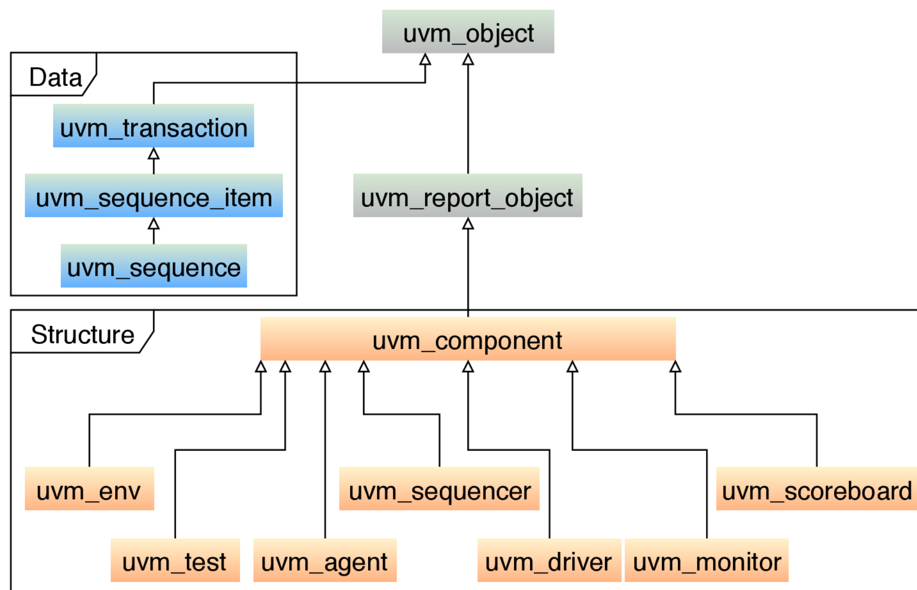
poskytovať dokumentáciu spolu so zdrojovými kódmi, voľne dostupnými prostredníctvom licencie Apache [9]. Táto metodika je navyše takmer kompletne spätne kompatibilná so staršou verifikačnou metodikou OVM.

Medzi základné aspekty metodiky UVM patrí pseudonáhodné generovanie vstupných stimulov verifikovaného obvodu, na ktoré je možné aplikovať obmedzujúce podmienky. Tieto podmienky zaisťujú, že vstupný stimul je vzhľadom na verifikovaný obvod legálny a pseudonáhodnosť stimulov môže pomôcť odhaliť aj neočakávané chyby obvodu. Verifikačné prostredia vytvárané prostredníctvom UVM sú široko konfigurovateľné a flexibilné. Dané verifikačné prostredie je ďalej oddelené od verifikačných testov, čo uľahčuje znovupoužiteľnosť konkrétneho prostredia a aj jeho komponent, viď obrázok 2.10. Komunikácia medzi komponentami prebieha na úrovni transakcií, teda na vysokom stupni abstrakcie, ktorý odstraňuje určité detaily sťažujúce možnosť znovupoužiteľnosti daných komponent [21]. Vstupný stimulus má sekvenčný charakter, a teda predstavuje sekvenciu transakcií. UVM však poskytuje hierarchický, vrstvený model týchto sekvencií, čím umožňuje vytvárať sekvencie sekvencií, respektíve virtuálne sekvencie a tým vytvárať skutočne komplexné a flexibilné verifikačné scenáre. Samotná verifikácia založená na UVM je riadená funkčným pokrytím.

Architektúra verifikačného prostredia podľa metodiky UVM

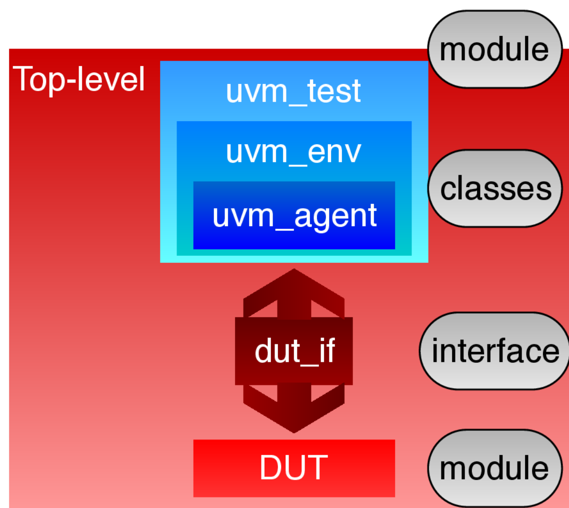
Ako už bolo spomenuté, metodika UVM poskytuje pre vytváranie verifikačných prostredí knižnicu základných a rozšírených tried jazyka SystemVerilog. Koreňovou triedou v hierarchii tejto knižnice je trieda *uvm_object* a z tejto triedy dedia, respektíve sú odvodené ďalšie, ktoré sú rozdelené do dvoch skupín podľa toho, ktoré vlastnosti verifikácie a verifikačného prostredia ovplyvňujú, viď obrázok 2.11.

Prvá skupina tried sa nazýva **data** a jej koreňová trieda má názov *uvm_transaction*, ktorá predstavuje už spomenutú transakciu. Počas simulácie, respektíve verifikácie obvodu vzniká a zaniká skutočne mnoho inštancií objektov z týchto tried, pretože akonáhle sa dáta aplikujú na verifikovaný obvod, už typicky nie sú potrebné. Druhou skupinou sú triedy typu **structure** s koreňovou triedou *uvm_component*. Triedy tohto typu zaisťujú konštrukciu a prepojenie jednotlivých komponent celej hierarchie verifikačného prostredia.



Obr. 2.11: Hierarchická triedna štruktúra verifikačnej metodiky UVM [8]

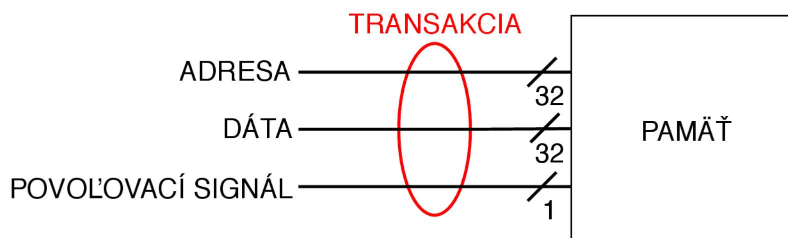
Architektúra verifikačného prostredia podľa verifikačnej metodiky UVM má svoju hierarchickú štruktúru, viď obrázok 2.12. Verifikovaný obvod je reprezentovaný ako modul a komunikácia s ním prebieha prostredníctvom štandardného rozhrania programovacieho jazyka SystemVerilog, teda **interface-u**, ktoré obsahuje všetky signály potrebné pre komunikáciu s obvodom. Tieto dva prvky architektúry sú štruktúrneho charakteru.



Obr. 2.12: Architektúra verifikačného prostredia podľa metodiky UVM [11]

Ďalším prvkom, ktorý je založený na triedach jazyka SystemVerilog, je samotné verifikačné prostredie UVM. Toto prostredie pozostáva z variabilnej časti a statickej časti [11]. Pevnú časť predstavuje **prostredie** samotné, ktoré je odvodené od triedy *uvm_env*. Toto prostredie zahŕňa všetky komponenty potrebné pre komunikáciu s verifikovaným obvodom a združuje ich do hierarchickej štruktúry. Variabilná časť sa nazýva **test** a je odvodený od triedy *uvm_test*. Úlohou testu je vytvorenie inštancie prostredia a jej nastavenie do poža-

dovanej konfigurácie, a taktiež definícia konkrétneho verifikačného scenára, ktorý daný test overuje, a teda aké sekvencie, a v akom poradí sa budú vykonávať, aké transakcie budú obsahovať, aký model pokrytia bude počas simulácie použitý apod. Prakticky sa pre verifikáciu využíva celá sada takýchto testov. Inštancie uvedených prvkov architektúry UVM sú zapúzdrené v module, ktorý sa nazýva **top-level** a sú z neho spúšťané jednotlivé testy [21].

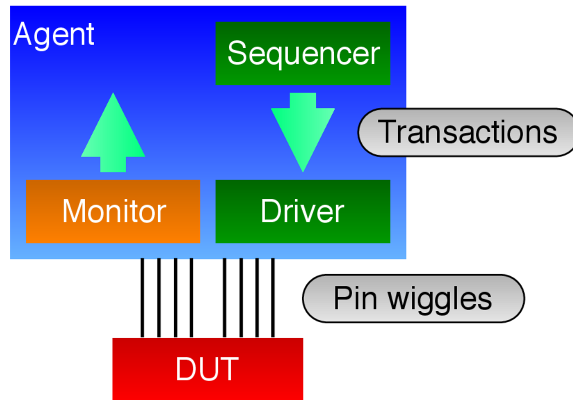


Obr. 2.13: Jednoduchá ukážka transakcie v UVM

Ako už bolo vyššie uvedené, komunikácia medzi komponentami prebieha na úrovni **transakcií**. Transakcia v kontexte metodiky UVM predstavuje všetky signály potrebné pre namodelovanie komunikačnej jednotky protokolu konkrétneho rozhrania daného verifikovaného obvodu [8]. Ako príklad môže poslúžiť transakcia asynchrónneho zápisu do pamäte, ktorá obsahuje povoľovací signál, adresu zápisu a zapisované dáta, prípadne ďalšie signály, viď obrázok 2.13.

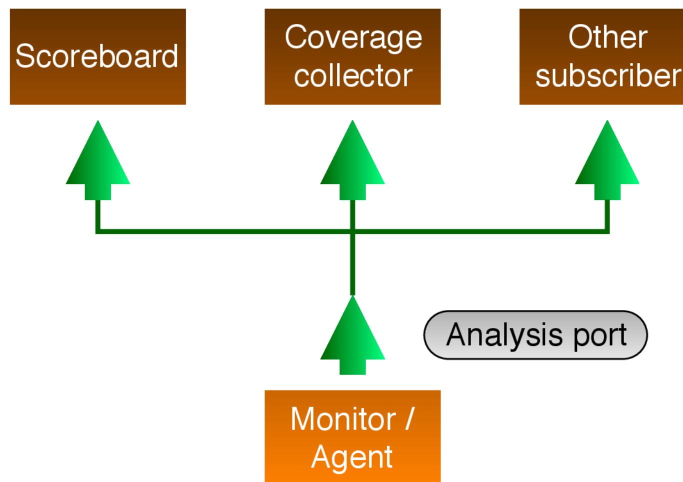
UVM transakcia teda obsahuje rôzne dátové položky a atribúty v závislosti na protokole. V transakcii je taktiež možné definovať sadu obmedzujúcich podmienok pre zaistenie toho, aby pseudonáhodne generované hodnoty jednotlivých položiek boli legálne. V konečnom dôsledku transakcia v UVM nie je odvodená od triedy *uvm_transaction*, ale od triedy *uvm_sequence_item*, pretože tá obsahuje určitú dodatočnú infraštruktúru, ktorá umožňuje preposielať dané transakcie medzi komponentami sequencer a driver, ktorých význam a úloha budú vysvetlené v nižšie. Jednotlivé transakcie sú generované objektom, ktorý je odvodený z triedy *uvm_sequence*, a teda UVM **sekvenciou**. Tento objekt obsahuje predpis pre generovanie obmedzených pseudonáhodných transakcií [1]. Využitie sekvencií je skutočne flexibilné, verifikačný test môže pozostávať z jednej sekvencie generujúcej transakcie kompletného verifikačného scenáru daného testu, alebo z veľkého množstva sekvencií, kde každá generuje iba jednu transakciu, typicky sa ale používa varianta medzi predošlými dvomi extrémami. Sekvencie môžu fungovať dokonca hierarchicky, v tomto prípade nadradená, virtuálna sekvencia slúži na generovanie a koordinovanie ďalších sekvencií. Vyššie popísaná UVM komponenta prostredie obsahuje ďalšie rôzne komponenty, z ktorých najdôležitejšie pre vysvetlenie sú komponenty verifikačné a analytické.

Verifikačná komponenta UVM sa nazýva **agent** a je odvodená od triedy *uvm_agent*. Jedná sa o hierarchickú komponentu zoskupujúcu ďalšie komponenty potrebné pre verifikáciu, ktoré komunikujú prostredníctvom rozhrania s verifikovaným obvodom, viď [1]. Spravidla sa jedná o tri špecifické komponenty, možné vidieť aj na obrázku 2.14. Prvou je komponenta **driver** odvodená od triedy *uvm_driver*, ktorá prijíma jednotlivé transakcie zo sekvencií a preposiela ich na jednotlivé piny rozhrania obvodu, ale už na signálovej úrovni, čím dosahuje určitý stupeň abstrakcie. Ďalšia komponenta sa nazýva **monitor** a je odvodená od triedy *uvm_monitor*. Táto komponenta vzorkuje signálovú aktivitu na jednotlivých pinoch rozhrania obvodu, zachytené informácie prevádza do podoby transakcií a tie posiela prostredníctvom analytického portu do analytických komponent. Poslednou komponentou je komponenta **sequencer**, ktorá je odvodená od triedy *uvm_sequencer*. Táto komponenta



Obr. 2.14: Schéma verifikačnej komponenty v UVM [8]

slúži ako arbiter alebo koordinátor toku transakcií, ktoré sú generované sekvenciami stimulov, respektíve UVM sekvenciami. Každá takáto transakcia je vygenerovaná na vyžiadanie komponenty driver a preposlaná do driveru prostredníctvom komponenty sequencer. Keďže verifikované obvody môžu disponovať viacerými rozhraniami, kedy každé používa svoj vlastný komunikačný protokol, komponenta prostredie typicky zoskupuje viacero verifikačných komponent agent, kde každý agent komunikuje s jedným rozhraním obvodu a je nakonfigurovaný podľa definície jeho protokolu. Ako je zrejmé, komponenta agent musí zároveň fungovať v aktívnom aj pasívnom režime, aby bola schopná posielať vstupné stimuly na rozhranie verifikovaného obvodu a súčasne toto rozhranie monitorovať.



Obr. 2.15: Ukážka prepojenia analytických komponent s monitorom v UVM [10]

Komponenta prostredie obsahuje okrem verifikačných komponent tiež komponenty analytické, ktoré sú pripojené na analytický port komponenty monitor, od ktorej cez tento port získavajú zachytené transakcie, ako na obrázku 2.15. Tieto komponenty sú odvodené od triedy *uvm_subscriber* [10]. Existujú rôzne druhy implementácie analytických komponent, no zrejme najvýznamnejšie sú konkrétne dve. Prvou je komponenta s názvom **scoreboard**, ktorej hlavnou funkciou je kontrola chovania verifikovaného obvodu pri daných vstupných transakciách a jeho porovnanie s chovaním referenčného modelu daného obvodu, známeho tiež ako **predictor**, pri rovnakých transakciách. Druhou implementáciou je komponenta,

ktorá sa nazýva **coverage collector**. Komponenty tohto druhu si udržujú zdieľanú databázu parametrov už vykonaných transakcií, prostredníctvom ktorej v kombinácii s definovaným modelom a stratégiou funkčného pokrytia ukazujú aktuálny stav pokroku vo verifikácii daného obvodu.

Kapitola 3

Praktická časť

Táto kapitola popisuje proces realizácie praktickej časti tejto práce. Úvodná časť opisuje existujúce implementácie periférie UART a prejednáva vhodnosť ich použitia v systéme FU540-C000. V nasledujúcej časti je uvedený návrh cieľového obvodu UART. Ďalšia časť opisuje dôležité prvky implementácie navrhnutého obvodu a obsahom záverečnej časti je popis tvorby verifikačného prostredia, a samotnej verifikácie implementovaného obvodu.

3.1 Zhodnotenie a kritika súčasného stavu

Táto podkapitola obsahuje analýzu existujúceho ovládaču pre periférie UART procesoru SiFive FU540-C000 a tiež analýzu už existujúcich návrhov periférie UART vytvorených v niektorom z jazykov HDL, ktoré by mohli byť vhodnými kandidátmi aj pre procesor FU540-C000. Tento prieskum bude zameraný na jednotlivé vlastnosti uvedené v špecifikácii tejto periférie a jej ovládaču, na spôsob, akým bol daný návrh verifikovaný, a taktiež na kvalitu dokumentácie.

3.1.1 Analýza existujúceho ovládaču

Kompletný zdrojový kód linuxového ovládaču pre túto perifériu je k dispozícii na [32]. Základ tohoto ovládaču tvoria funkcie, ktoré slúžia na čítanie, respektíve zápis danej hodnoty do registru určeného adresou. Pomocou týchto funkcií a ich kombinácií sú implementované ďalšie funkcie, prostredníctvom ktorých je možné zapisovať, respektíve čítať z registrov uvedených v kapitole 2 a tým ovládať prenos dát po sériovej linke, a jeho parametre. Väčšina prístupov do registrov poskytnutých ovládačom je buď bezpečná, alebo je možné ich softvérovo ošetriť inými funkciami implementovanými ovládačom. Jedinou výnimkou je aktualizácia prenosovej rýchlosti, kedy ovládač nekontroluje, či je vysielač alebo prijímač aktívny. Pokiaľ by k takejto aktualizácii došlo počas prebiehajúceho prenosu, mohlo by to viesť k nedefinovanému chovaniu, preto táto funkcionalita vyžaduje hardvérové ošetrenie. Daný ovládač teda požaduje od obvodu podporu nasledujúcej funkcionality:

- adresový priestor a sada registrov, ktoré sú uvedené v tabuľke 2.3,
- počiatočná prenosová rýchlosť 115200 baudov,
- podpora režimu FIFO s veľkosťou ôsmich záznamov pre prijímací aj vysielací kanál,
- formát dátového rámca 8-N-1, respektíve 8-N-2,

- použitie techniky 16 × oversampling with 2/3 majority voting per bit pre vzorkovanie prijímacieho kanálu,
- podpora watermark prerušení, ktoré sú propagované aj mimo samotnej periférie
- aktivácia a deaktivácia vysielacieho kanálu zápisom do políčka *txen* registru Transmit Control,
- nastavenie počtu stop bitov pre vysielací kanál zápisom do políčka *nstop* registru Transmit Control,
- nastavenie podmienky pre vyvolanie prerušenia od vysielacieho kanálu zápisom do políčka *txcnt* registru Transmit Control,
- odoslanie bytu, respektíve niekoľkých bytov, zápisom do vysielacej FIFO prostredníctvom políčka *data* registru Transmit Data,
- overenie plnosti vysielacej FIFO prečítaním príznaku *full* z registru Transmit Data,
- aktivácia a deaktivácia prijímacieho kanálu zápisom políčka *rxen* registru Receive Control,
- nastavenie podmienky pre vyvolanie prerušenia od prijímacieho kanálu zápisom do políčka *rxcnt* registru Receive Control,
- prijatie bytu, respektíve niekoľkých bytov, čítaním z prijímacej FIFO prostredníctvom políčka *data* registru Receive Data,
- overenie platnosti prijatých dát čítaním príznaku *empty* z registru Receive Data,
- povolenie a zakázanie prerušení od vysielaču, respektíve prijímaču, zápisom do políčok *txwme* a *rxwme* registru Interrupt Enable,
- overenie existencie nespracovaných prerušení od vysielaču, respektíve prijímaču, čítaním z políčok *txwmp* a *rxwmp* registru Interrupt Pending,
- aktualizácia prenosovej rýchlosti zápisom do políčka *div* registru Baud Rate Divisor.

3.1.2 Analýza existujúcich implementácií

Pre túto analýzu bolo zvolených deväť konkrétnych implementácií, ktorých prienik s požadovanou špecifikáciou je uvedený nižšie.

VHDL 16550 UART Core

Tento návrh je k dispozícii v jazyku VHDL spolu s pomerne obsiahlou dokumentáciou [18]. Rozhranie komponenty na najvyššej úrovni je možné pripojiť na poskytnuté prevodníky do zbernicových protokolov AMBA a Wishbone, prevodník do protokolu TileLink však absentuje. Funkcionalita tejto komponenty presahuje špecifikáciu periférie pre procesor SiFive FU540-C000. V niektorých parametroch, ako napríklad adresovom priestore alebo riadení prerušení, sa však rozchádza. K návrhu nie sú dostupné žiadne zdroje ani informácie o verifikácii.

MiniUART a UART16550

Pre implementáciu komponenty MiniUART [26] bol zvolený jazyk VHDL a komponenta UART16550 [23] je k dispozícii v jazykoch VHDL aj Verilog. Obe dokumentácie sú dostatočne obšírne a obsahujú všetky náležitosti. Rozhranie je navrhnuté špeciálne pre zbernicový protokol Wishbone. Adresový priestor ani sada použitých registrov sa nezhoduje s požiadavkami vyplývajúcimi zo špecifikácie. Komponente chýba tiež podpora pre FIFO a konfiguráciu dátového rámca. K návrhu je dodaných aj pár jednoduchých testov pre simuláciu a dokonca tiež výstup zo syntézy.

Simple UART

Jedná sa o veľmi jednoduchú implementáciu v jazyku Verilog, ktorú nie je možné pripojiť k žiadnemu zbernicovému protokolu ani prevodníku, viď [4]. Chýba možnosť konfigurácie dátového rámca a prenosovej rýchlosti. Použitie FIFO a prerušení nie je podporované. Priložená dokumentácia nie je kompletná a nie sú k dispozícii žiadne verifikačné testy.

Simple UART For FPGA

Táto komponenta je naimplementovaná v jazyku VHDL a je k nej priložená stručná dokumentácia obsahujúca informáciu, že daný návrh bol otestovaný aj v hardvéri, viac na [5]. Pre pripojenie rozhrania k zbernicovému protokolu je potrebný prevodník, ktorý absentuje. Podobne ako pri predchádzajúcej, komponente nie je implementovaná funkcionalita pre konfiguráciu dátového rámca a prenosovej rýchlosti, ani pre podporu prerušení alebo FIFO. K návrhu je priložených niekoľko jednoduchých testov pre simuláciu.

SSP UART

Jedná sa o komplexnú implementáciu systému UART v jazyku Verilog, ktorá prevyšuje požadovanú špecifikáciu [24]. Priložená dokumentácia obsahuje všetky dôležité informácie o komponente. Je tiež k dispozícii väčšie množstvo testov pre simuláciu. Sada používaných registrov a ich adresový priestor sa však nezhodujú so špecifikáciou.

UART To Bus

Tento návrh je k dispozícii hneď v dvoch jazykoch, a síce VHDL a Verilog. V dokumentácii sú zmienené všetky dôležité aspekty, viď [19]. Komponente však oproti špecifikácii chýba podpora pre FIFO a prerušenia. Návrh zahŕňa pre obe verzie implementácie zdroje na simuláciu v podobe niekoľkých testov a tiež výstupy zo syntézy.

Uart Wishbone Slave

Rozsiahla a zložitá implementácia periférie UART v jazyku VHDL doplnená o obšírnu dokumentáciu a pomerne veľké množstvo testov pre každú súčasť systému [27]. Nepodporuje však FIFO, prerušenia ani konfiguráciu dátového rámca. Taktiež adresový priestor a sada použitých registrov nie sú kompatibilné s požiadavkami špecifikácie.

MiniUART Core

Ide o jednoduchú implementáciu v jazyku VHDL, ktorej jedinú dokumentáciu predstavujú komentáre v zdrojových kódoch, viac na [20]. Je navrhnutá pre konkrétny typ procesoru

a neposkytuje podporu pre FIFO. K tejto komponente je priložených niekoľko jednoduchých testov.

3.1.3 Záver hodnotenia

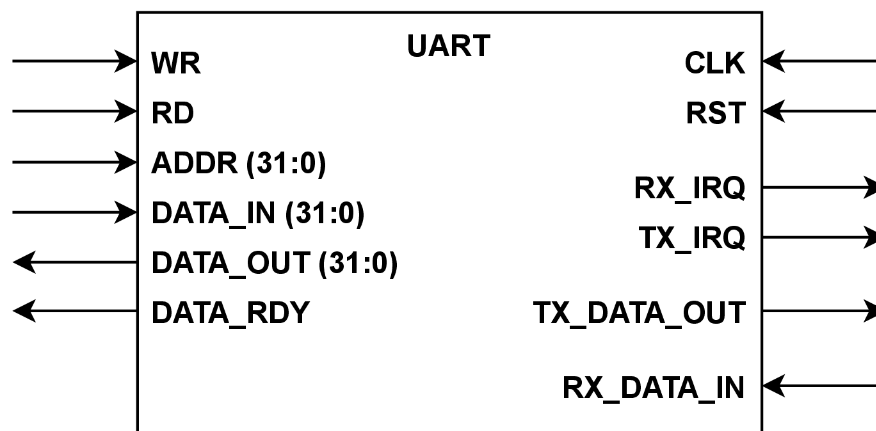
Vyššie uvedené implementácie periférie UART buď nie sú verifikované prostredníctvom prepracovaného verifikačného prostredia, vytvoreného podľa niektorej verifikačnej metodiky, alebo nie sú verifikované vôbec. Predovšetkým však nie sú kompatibilné so špecifikáciou pre procesor SiFive FU540-C000, každá v individuálnej miere. Preto bude cieľom praktickej časti tejto práce komponentu spĺňajúcu definované požiadavky navrhnúť, implementovať a patričným spôsobom aj verifikovať.

3.2 Návrh implementácie

Popis návrhu obvodu UART je vyhotovený prístupom zhora dole, a teda od komponenty na najvyššej úrovni smerom k podkomponentám.

3.2.1 Rozhranie obvodu UART

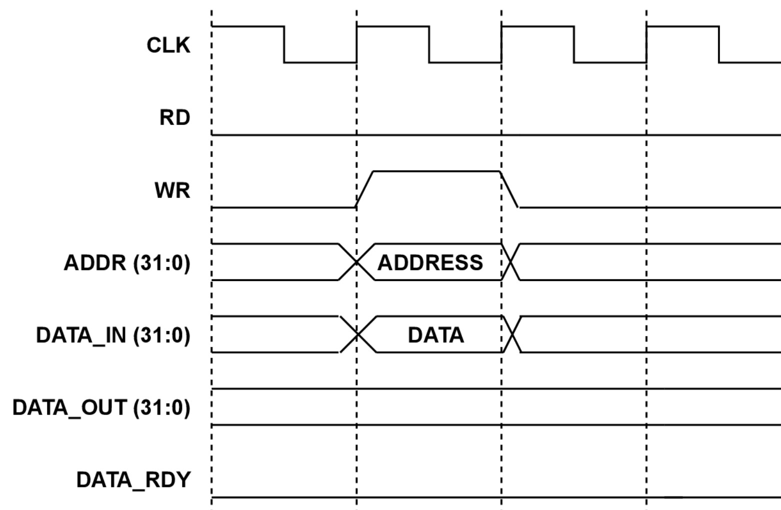
Navrhnuté rozhranie obvodu UART je rozdelené na päť častí. Bloková schéma tohoto rozhrania je zobrazená na obrázku 3.1.



Obr. 3.1: Schéma rozhrania obvodu UART

Prvá časť slúži ako vstupné rozhranie pre riadiace signály, a síce vstupný hodinový signál *clk* a vstupný signál *rst* určený pre asynchrónne resetovanie obvodu.

Druhá časť je určená pre pripojenie obvodu k prevodníku na konkrétny zbernicový protokol, respektíve pre komunikáciu s nadradenou komponentou. Táto časť obsahuje vodiče *wr*, *r*, *addr*, *data_in*, *data_out* a *data_rdy*. Vstupné príznaky *wr* a *r* slúžia ako povoľovacie signály pri zápisových, respektíve čítacích zbernicových transakciách. Vstupný 32 bitový signál *addr* určuje, z ktorého registru komponenty UART sa má čítať, alebo do ktorého sa má zapisovať. Po 32 bitovom vstupnom vodiči *data_in* prichádzajú dáta určené pre zápis do niektorého z registrov komponenty. Časový diagram zápisovej transakcie je zobrazený na obrázku 3.2. Na výstupný 32 bitový signál *data_out* sa po úspešnom spracovaní čítacej transakcie vystavia dáta z príslušného registru. Výstupný príznak *data_rdy*, určuje validitu dát na výstupnom vodiči *data_out*, ako je uvedené na obrázku 3.3.

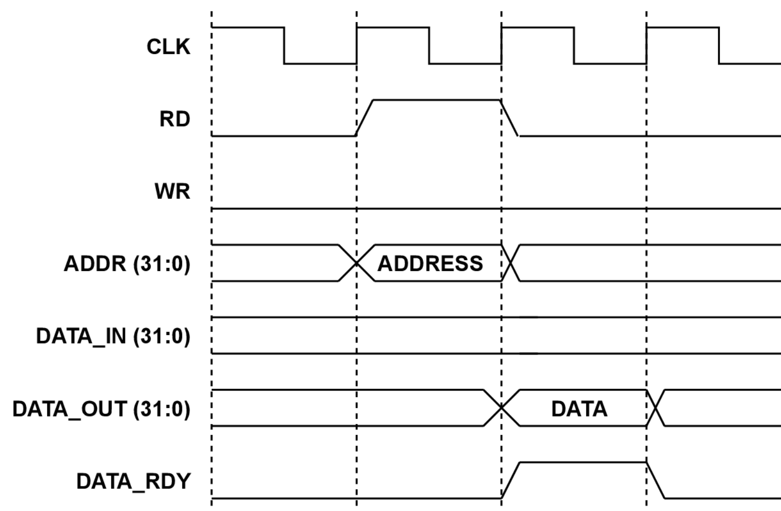


Obr. 3.2: Časový diagram zápisovej transakcie

Úlohou tretej časti je poskytnutie výstupného rozhrania pre prerušenia, aby mohli byť propagované do radiču PLIC procesoru FU540-C000. Výstupný príznak *tx_irq* definuje prítomnosť vyvolaného a nespracovaného prerušenia so zdrojom vo vysielacej časti, a príznak *rx_irq* zase v časti prijímacej.

Štvrtá časť slúži ako výstupné rozhranie vysielacieho kanálu, kedy vysielacia časť pri aktívnom prenose po vodiči *tx_data_out* posiela jednotlivé bity dát určených na odoslanie pripojenému zariadeniu.

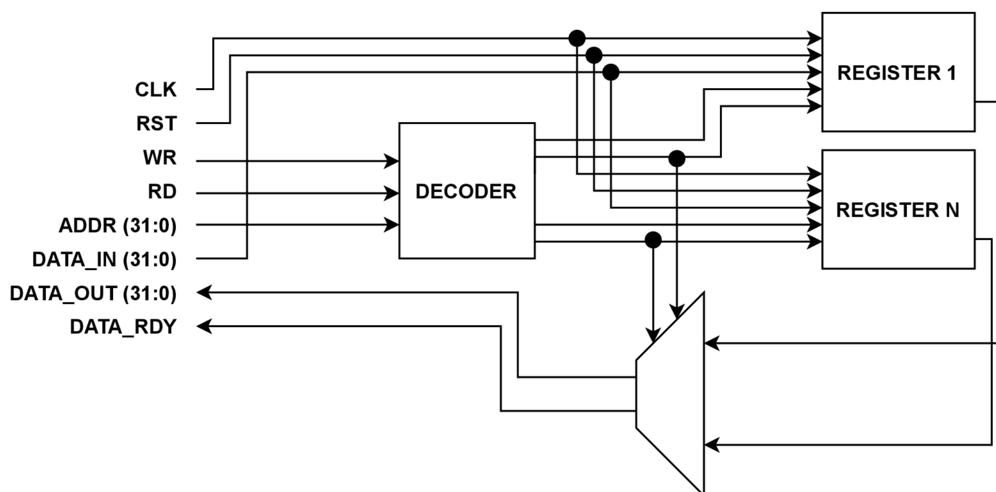
Účelom poslednej časti je poskytnutie vstupného rozhrania pre prijímací kanál. Prijímacia časť vzorkuje signál *rx_data_in* a pri aktívnom prenose získané dáta posiela ďalej na ďalšie spracovanie.



Obr. 3.3: Časový diagram čítacej transakcie

3.2.2 Zapojenie riadiacich registrov s rozhraním

Riadiace registre obvodu UART uvedené v tabuľke 2.3 sú k vyššie popísanému rozhraniu pripojené nasledujúcim spôsobom. Do každého registru sú privedené signály *clk* a *rst*. Ku každému z registrov, do ktorých je možné zapisovať, je pripojený vstupný dátový vodič *data_in*. Vodiče *wr*, *rd* a *addr* sú privedené na vstup adresového dekodéru. Tento dekodér na základe vstupnej adresy určuje, ktorý register bude aktívny pri spracovávaní aktuálnej transakcie, nastala zhoda s niektorou z adries. Po rozhodnutí nastaví povoľovací signál pre čítanie, respektíve zápis, ktorý je pripojený k príslušnému registru. Pokiaľ šlo o zápisovú transakciu, aktívny register prečíta dáta z vodiča *data_in* a zapíše ich do svojho úložiska. Do niektorých registrov z ich definície v špecifikácii nejde zapisovať a takéto transakcie sú ignorované. V prípade, že sa jedná o čítaciu transakciu, vyzvaný register vystaví dáta na svoj výstupný vodič, ktorý je privedený na vstup výstupného multiplexoru. Tento multiplexor propaguje obdržané dáta na výstupný vodič *data_out*, pričom nastavuje príznak *data_rdy* do logickej jednotky. Pre výber výstupných dát multiplexor využíva povoľovacie signály pre čítanie jednotlivých registrov. Schéma popísaného zapojenia je uvedená na obrázku 3.4.

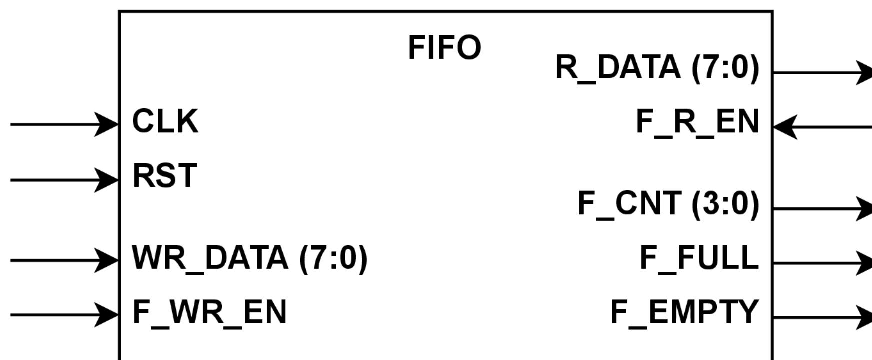


Obr. 3.4: Schéma zapojenia riadiacich registrov k rozhraniu

Registre Transmit Control, Receive Control, Interrupt Enable a Interrupt Pending sú prepojené ešte s ďalšími vodičmi rozhrania obvodu UART, toto zapojenie bude však popísané nižšie v podkapitole o prerušeniach.

3.2.3 FIFO

Periféria UART využíva podľa špecifikácie buffer typu FIFO s veľkosťou ôsmich bytov. Na vstup tejto časti obvodu sú privedené riadiace signály *clk* a *rst*. Pokiaľ je nastavený vstupný povoľovací signál *f_wr_en* a FIFO nie je plná, sú dáta vystavené na vodiči *wr_data* uložené na koniec FIFO. Výstupný signál *r_data* reflektuje aktuálny záznam uložený na začiatku FIFO. Ak je vstupný signál *f_r_en* nastavený do logickej jednotky a zároveň FIFO nie je prázdna, tak je na *r_data* vystavený nasledujúci záznam. Výstupný signál *f_cnt* udáva aktuálny počet záznamov uložených v FIFO. Príznaky *f_full* a *f_empty* reflektujú, či je FIFO plná, respektíve prázdna, viď obrázok 3.5. Takto navrhnutú FIFO je možné pri korektnom zapojení využiť pre vysielač a zároveň aj pre prijímač. Takéto zapojenia budú popísané v nasledujúcich častiach tejto podkapitoly.



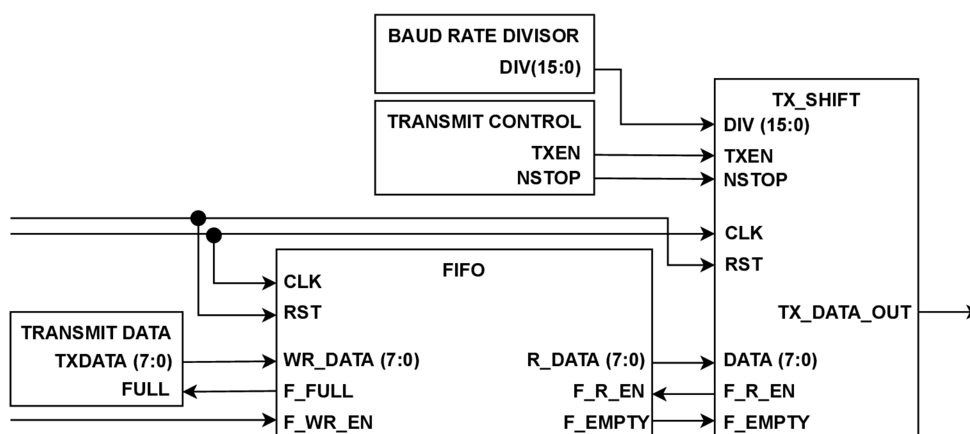
Obr. 3.5: Schéma bufferu typu FIFO

3.2.4 Vysielacia časť

Vysielacia časť obvodu UART pozostáva z obvodov, ktoré zaisťujú chovanie relevantných riadiacich registrov, vysielacej FIFO a výstupnej komponenty, ako je zobrazené na obrázku 3.6.

Zápisy dát do vysielacej FIFO vykonáva obvod, ktorý zahŕňa funkcionalitu registru Transmit Data. Pri zápise vystaví dáta určené pre zápis na signál *txdata* a kontroluje príznak *full* na jeho vstupe. Pokiaľ je nulový, nastaví príznak *f_wr_en* do logickej jednotky a následne sú dané dáta uložené na koniec vysielacej FIFO.

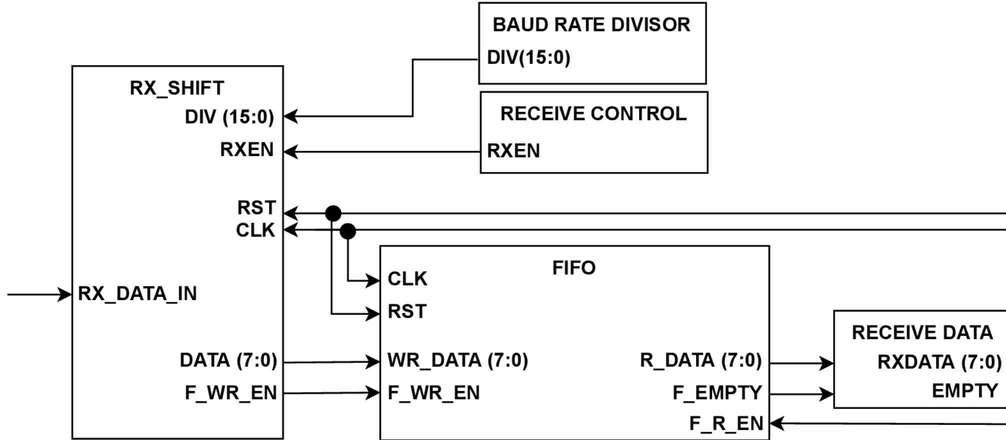
Čítania dát z vysielacej FIFO vykonáva výstupná komponenta vysielacej časti *tx_shift*. Tento obvod je aktívny len v prípade, že je jeho vstupný príznak *txen*, definujúci aktívny vysielací kanál, z registru Transmit Control nastavený do logickej jednotky. V prípade aktívneho vysielacieho kanálu a neprázdnej FIFO komponenta prečíta záznam uložený na začiatku FIFO a nastavením príznaku *f_r_en* ju informuje, že môže na svoj začiatok presunúť nasledujúci záznam. Obdržané dáta potom tento obvod vysiela po jednotlivých bitoch, prostredníctvom výstupného signálu rozhrania obvodu UART *tx_data_out*. Prenosovú rýchlosť komponenta odvodzuje zo vstupného signálu *div* od registru Baud Rate Divisor a počet stop bitov v dátovom rámci je daný hodnotou signálu *nstop* z registru Transmit Control.



Obr. 3.6: Schéma vysielacej časti obvodu UART

3.2.5 Prijímacia časť

Prijímacia časť obvodu UART obsahuje komponenty, ktoré zaisťujú chovanie relevantných riadiacich registrov, prijímaciu FIFO a vstupnú komponentu *rx_shift*. Táto časť je uvedená na obrázku 3.7.



Obr. 3.7: Schéma prijímacej časti obvodu UART

Zápisy dát do prijímacej FIFO vykonáva vstupná komponenta prijímacej časti. Daná komponenta je aktívna len v prípade, že je príznak *rxen* od riadiaceho registru Receive Control na jej vstupe, definujúci aktívny prijímací kanál, nastavený do logickej jednotky. V prípade, že je prijímací kanál aktívny, komponenta vzorkuje vstupný signál rozhrania obvodu UART *rx_data_in*, pričom frekvenciu vzorkovania odvodzuje z hodnoty vstupného signálu *div* od registru Baud Rate Divisor. Po spracovaní celého bytu komponenta tento byte vystaví na výstupný dátový vodič a zároveň nastaví príznak *f_wr_en* do logickej jednotky, čo následne spôsobí zápis daného bytu na koniec prijímacej FIFO.

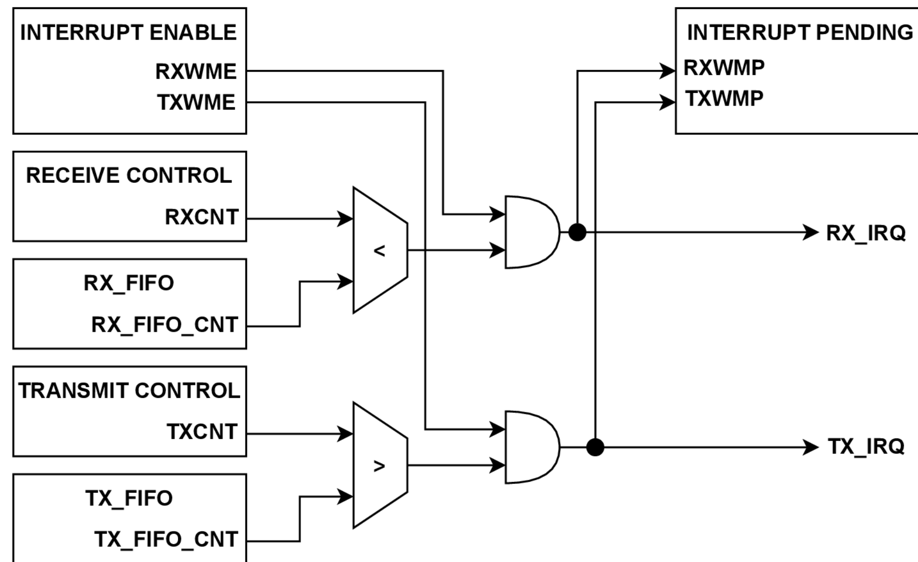
Čítania dát z prijímacej FIFO sú zaistené obvodom, ktorý zahŕňa funkcionalitu registru Receive Data. Po prečítaní pošle záznam zo začiatku FIFO spolu s príznakom *empty*, ktorý definuje validitu obdržaných dát, na ďalšie spracovanie a zároveň nastavením príznaku *f_r_en* do logickej jednotky informuje FIFO, že môže na svoj začiatok presunúť nasledujúci záznam.

3.2.6 Prerušenia

Funkcionalita prerušení je navrhnutá nasledujúcim spôsobom. Hodnota vodiča *rx_f_cnt* udáva počet záznamov v prijímacej FIFO. Signál *rxcnt* z registru Receive Control definuje prah počtu záznamov v prijímacej FIFO, ktorý vyvoláva prerušenie, viď obrázok 3.8. V prípade, kedy je počet záznamov ostro väčší než definovaný prah a zároveň je prerušenie od prijímacej časti povolené signálom *rxwme* z registru Interrupt Enable nastaveným do logickej jednotky, jedná sa o nespracované vyvolané prerušenie. V prípade neplatnosti predchádzajúcich podmienok prerušenie nenastalo. Informácia o existencii prerušenia z prijímacej časti sa ukladá do registru Interrupt Pending do políčka *rxwmp* a taktiež sa propaguje cez výstupný signál rozhrania obvodu UART *rx_irq*.

Prerušenia z vysielacej časti sú navrhnuté obdobným spôsobom s tým rozdielom, že podmienkou pre vyvolanie prerušenia je počet záznamov vo vysielacej FIFO ostro menší než je definovaný prah. Povolenie prerušenia udáva hodnota príznaku *txwme* z registru Iterrupt

Enable a informácia o existencii prerušenia sa ukladá do registru Interrupt Pending na políčko *txwmp*, a propaguje sa cez výstupný signál *tx_irq*.



Obr. 3.8: Schéma časti obvodu zaisťujúcej vyvolanie a propagáciu prerušení

3.3 Implementácia

Pre implementáciu navrhnutého obvodu UART bol zvolený programovací jazyk VHDL. Celá implementácia je rozdelená medzi niekoľko komponent, ktoré dohromady tvoria výsledný systém.

3.3.1 FIFO

Táto komponenta implementuje buffer typu FIFO, ktorý využíva vysielacia aj prijímajúca časť obvodu UART procesoru FU540-C000. Uvedená FIFO je takzvane kruhová.

Entita tejto komponenty je totožná s rozhraním tohoto obvodu uvedenom v podkapitole o návrhu, zobrazenom na obrázku 3.5, pričom je rozšírená o dva generické parametre. Prvý parameter udáva bitovú šírku jedného záznamu a druhý definuje maximálny počet záznamov v danej FIFO. Vďaka týmto parametrom je možné túto implementáciu prispôsobiť a využiť aj v iných projektoch. Táto implementácia je nakonfigurovaná na osem záznamov o šírke ôsmich bitov.

Architektúra, respektíve definícia chovania, tejto komponenty je implementovaná prostredníctvom kombinácie behaviorálneho a dataflow popisu.

Behaviorálnu časť predstavuje proces, ktorý je citlivý na vstupný hodinový signál a resetovací signál. Tento proces zaisťuje funkcionalitu čítaču počtu záznamov vo FIFO a registrov, ktoré zaznamenávajú uložené záznamy, hodnotu indexu, na ktorý sa má zapisovať a hodnotu indexu, ktorý udáva pozíciu záznamu na začiatku FIFO. Pri aktívnej hodnote resetu sú uvedené hodnoty vynulované. Pri nástupných hranách hodinového signálu sú snímané hodnoty povoľovacích signálov pre zapisovanie a čítanie. Pokiaľ FIFO nie je plná a je iniciovaný zápis, tak je na aktuálny zápisový index úložiska pridaný záznam získaný zo vstupného dátového signálu. Následne sú inkrementované hodnoty zápisového indexu

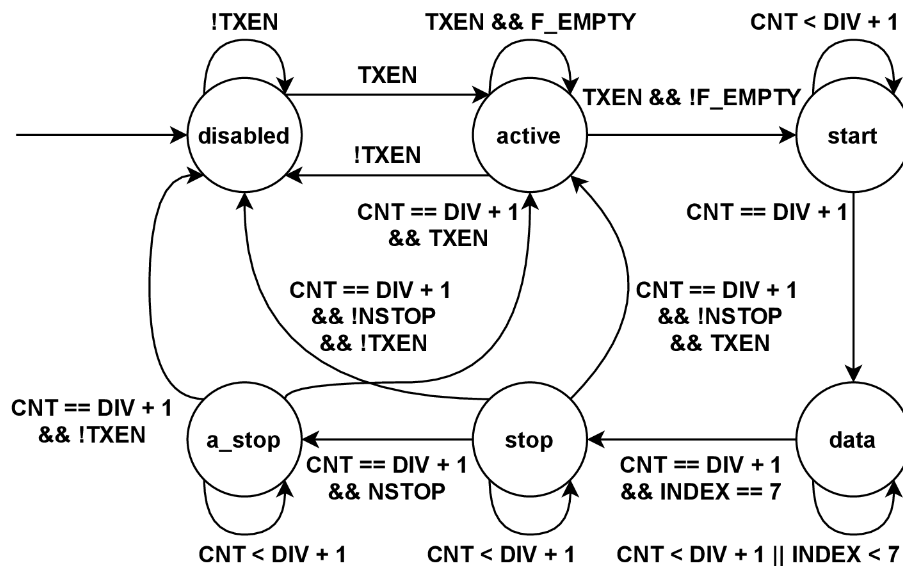
a interného čítaču počtu záznamov. V prípade neprázdnosti FIFO a inicovaného čítania sú dekrementované hodnoty čítacieho indexu a interného čítaču počtu záznamov.

V dataflow časti je propagovaná hodnota čítaču počtu záznamov na výstupné rozhranie komponenty. Výstupné príznaky definujúce plnosť FIFO, respektíve jej prázdnosť, sú odvodené z hodnoty spomenutého čítaču. Hodnotu výstupného dátového signálu určuje záznam v internom úložisku umiestnený na indexe odpovedajúcom aktuálnej hodnote čítacieho indexu.

3.3.2 Výstupná komponenta vysielacej časti

Táto komponenta slúži k posielaniu jednotlivých bitov určených dát po výstupnej sériovej linke obvodu UART.

Entita tejto komponenty vychádza z rozhrania uvedeného na obrázku 3.9. Je doplnená o dva generické parametre, kde prvý určuje bitovú šírku dátového slova, ktoré register bude vysielať, a druhý definuje bitovú šírku signálu, ktorý obsahuje hodnotu deliaceho pomeru pre odvodenie požadovanej prenosovej rýchlosti. Vďaka týmto parametrom je možné túto komponentu parametrizovať a využiť aj pre iné špecifikácie obvodu UART. Konfigurácia tejto implementácie predstavuje 8 bitové dátové slovo a 16 bitový signál *div*.



Obr. 3.9: Stavový automat využívaný výstupnou komponentou vysielacej časti obvodu UART

Definícia chovania komponenty, alebo jej architektúra, využíva behaviorálny popis, kde je celá funkcionalita popísaná jedným procesom, ktorý je citlivý na vstupný hodinový signál a resetovací signál. Tento proces pozostáva z registrov, ktoré udržujú dáta určené na odoslanie a index aktuálne odosielaného bitu. Proces tiež obsahuje čítač hodinových cyklov. Zvyšnú časť procesu tvorí stavový automat, ktorý riadi spravovanie uvedených podkomponent a odosielanie dát. Diagram tohoto automatu je uvedený na obrázku 3.9.

Počiatočným stavom tohoto automatu je stav *disabled*, počas ktorého je na výstupnom vodiči nastavená kľudová úroveň, a teda logická jednotka. Do tohto stavu sa automat dostáva aj po resetovaní. Automat zostáva v tomto stave, až kým nie je nastavený vstupný signál *txen* do logickej jednotky, čo spôsobí aktiváciu vysielacieho kanálu a prechod auto-

matu do stavu *active*, počas ktorého je na výstupnom vodiči nastavená kľudová úroveň. Pokiaľ v tomto stave spadne hodnota signálu *txen* do logickej nuly, automat sa vracia do stavu *disabled*. V opačnom prípade je snímaný vstupný príznak *f_empty*, ktorý určuje prázdnosť vysielacej FIFO. Pokiaľ FIFO nie je prázdná, komponenta si uloží dáta zo vstupného dátového vodiča do interného úložiska, informuje FIFO, že môže vystaviť na výstup nasledujúci záznam, vynuluje interný čítač hodinových cyklov a automat prejde do stavu *start*. Pokiaľ bude v niektorom z nasledujúcich stavov vysielací kanál deaktivovaný, komponenta najskôr dokončí prenos obdržaných dát a následne prejde do stavu *disabled*.

Počas stavu *start* je na výstupnom dátovom vodiči nastavená hodnota logickej nuly, ktorá definuje štart bit. Komponenta inkrementuje svoj čítač hodinových cyklov, až kým nenadobudne hodnotu rovnú hodnote vstupného signálu *div* inkrementovanej o jedna. V tomto momente už bol štart bit na výstupnom vodiči po dobu danú prenosovou rýchlosťou a komponenta nuluje čítač hodinových cyklov, bitový index vysielaných dát a automat prechádza do stavu *data*.

V tomto stave komponenta vystavuje na dobu danú prenosovou rýchlosťou jednotlivé bity dát obdržaných z vysielacej FIFO na výstupný vodič, pričom využíva čítač hodinových cyklov rovnako, ako v predchádzajúcom stave. Pozícia vysielaného bitu je daná hodnotou bitového indexu, ktorý sa vždy uplynutí počtu hodinových cyklov pre jeden bit inkrementuje. Keď sú odoslané všetky bity, automat prechádza do stavu *stop*.

Počas tohto stavu je po dobu danú prenosovou rýchlosťou na výstupnom vodiči nastavená hodnota logickej jednotky definujúca stop bit. Následne, pokiaľ je nastavený príznak *nstop*, prechádza automat do stavu *a_stop*, v ktorom komponenta odošle druhý stop bit. Ak tento príznak nie je nastavený, automat prechádza do stavu *active*, respektíve *disabled*, v závislosti na hodnote signálu *txen*. Aktualizácia stavu prebieha rovnako aj po odoslaní druhého stop bitu.

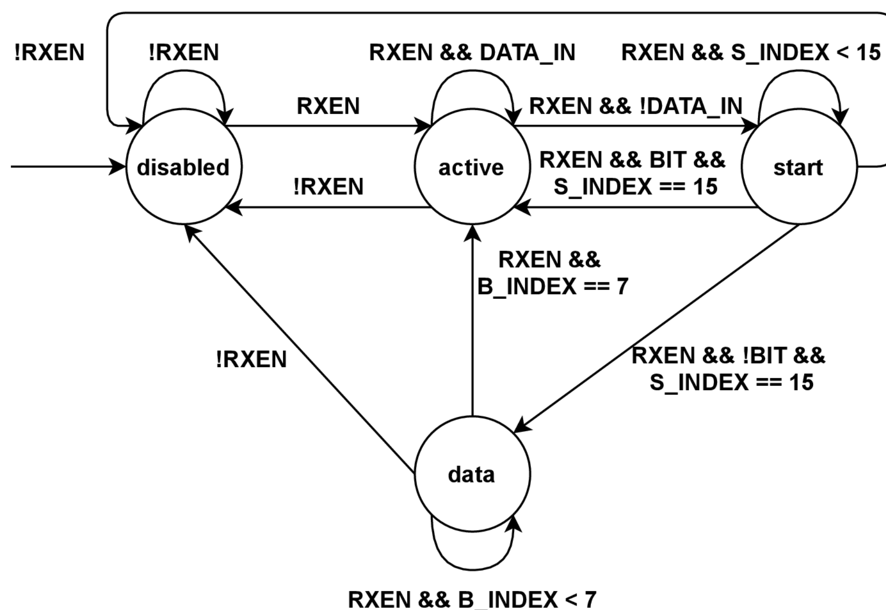
3.3.3 Vstupná komponenta prijímacej časti

Účelom tejto komponenty je vzorkovanie vstupnej sériovej linky obvodu UART a zapisovanie spracovaných dát do prijímacej FIFO.

Entita tejto komponenty vychádza z rozhrania uvedeného v návrhu, viď obrázok 3.7, je však rozšírená o sedem generických parametrov. Prvé dva parametre predstavujú bitové šírky dátového slova, respektíve signálu obsahujúceho hodnotu deliaceho pomeru pre odvodenie požadovanej prenosovej rýchlosti. Tretí udáva počet vzoriek, ktorý pripadá na jeden prijímaný bit. Štvrtý udáva hodnotu, ktorou je potrebné deliť vstupný signál *div* pre odvodenie vzorkovacej frekvencie. Posledné tri parametre slúžia ako indexy pre výber vzoriek, z ktorých bude odvodená výsledná hodnota prijatého bitu. Tieto parametre umožňujú konfigurovať komponentu pre použitie v obvode UART podľa odlišnej špecifikácie. Táto implementácia je nakonfigurovaná na šírku dátového slova osem bitov, 16 bitový signál *DIV*, šestnásť vzoriek získaných vzorkovaním šestnásťkrát väčšou frekvenciou než má prenosová rýchlosť a výslednú hodnotu prijatého bitu určenú ôsmou, deviatou a desiatou vzorkou.

Pre implementáciu architektúry tejto komponenty bol použitý behaviorálny popis. Konkrétne sa jedná o proces citlivý na vstupný hodinový signál a resetovací signál. Tento proces je pozostáva z niekoľkých registrov. Dané registre v sebe udržujú vzorky získané zo vstupnej sériovej linky obvodu UART, hodnotu indexu aktuálne získavanej vzorky, deliaci pomer pre odvodenie požadovanej vzorkovacej frekvencie, už spracované dáta a index aktuálne prijímaného bitu. Proces tiež obsahuje čítač hodinových cyklov. Zvyšok procesu tvorí sta-

vový automat, ktorý spravuje činnosť uvedených podkomponent a prijímanie dát. Diagram tohoto automatu je zobrazený na obrázku 3.10.



Obr. 3.10: Stavový automat využívaný vstupnou komponentou prijímacej časti obvodu UART

Počiatočným stavom daného automatu je stav *disabled* a automat do tohoto stavu prechádza aj po resetovaní, alebo zhodení vstupného signálu *rxen* do logickej nuly. Automat zostáva v tomto stave, až kým nie je nastavený signál *rxen* do logickej jednotky, čo spôsobí aktiváciu prijímacieho kanálu, prechod automatu do stavu *active* a nulovanie úložiska vzoriek, vzorkového indexu a čítaču hodinových cyklov.

V stave *active* komponenta sníma hodnotu vstupnej sériovej linky obvodu UART. V momente, keď detekuje logickú nulu, uloží ju do vzorkovacieho úložiska, inkrementuje vzorkovací index a čítač hodinových cyklov, a prejde do stavu *start*.

Počas stavu *start* komponenta naďalej sníma hodnotu vstupnej sériovej linky podľa vzorkovacej frekvencie, pričom inkrementuje čítač hodinových cyklov. Získané vzorky ukladá do úložiska a pritom inkrementuje vzorkový index, a nuluje čítač hodinových cyklov. Po získaní šestnástich vzoriek je hodnota výsledného bitu daná väčšinovou hodnotou spomedzi jednotlivých vybraných vzoriek a vzorkové úložisko spolu s vzorkovým indexom sú vynulované. Pokiaľ šlo o bit s hodnotou nula, bol detekovaný štart bit a automat prechádza do stavu *data*, a nuluje bitový index. V opačnom prípade sa vracia do stavu *active*.

V stave *data* komponenta vzorkuje jednotlivé bity rovnakým spôsobom ako v predchádzajúcom stave, pričom ich postupne reflektuje na výstupný dátový vodič a zároveň inkrementuje bitový index. Po prijatí ôsmich bitov je nastavený príznak *f_wr_en* a do vysielacej FIFO je zapísaný nový záznam. Automat následne prechádza do stavu *active*.

3.3.4 UART

Ide o komponentu na najvyššej úrovni hierarchie, ktorá v sebe zapúzdruje vyššie uvedené podkomponenty.

Entita tejto komponenty vychádza z navrhnutého rozhrania, viď obrázok 3.1. Je však rozšírená o šestnásť generických parametrov. Prvý parameter určuje iniciálnu hodnotu deliaceho pomeru pre odvodenie požadovanej prenosovej rýchlosti. Nasledujúcich sedem parametrov definuje adresy jednotlivých riadiacich registrov, ktoré sú uvedené v tabuľke 2.3. Zvyšné parametre slúžia k parametrizácii podkomponent a sú popísané v predchádzajúcich častiach tejto podkapitoly. Vďaka týmto parametrom je možné túto komponentu integrovať aj do systému s inou špecifikáciou.

Architektúra komponenty je implementovaná prostredníctvom kombinácie štrukturálneho, behaviorálneho a dataflow popisu.

V štrukturálnej časti sú inštanciované a parametrizované komponenty vysielacej, a prijímacej FIFO, výstupného registru vysielacej časti a vstupného registru prijímacej časti. Tieto komponenty sú prepojené navzájom medzi sebou a taktiež so signálmi rozhrania, a riadiacimi registrami presne takým spôsobom, aký je uvedený v návrhu.

Behaviorálnu časť tvorí skupina procesov, ktoré zaisťujú funkcionalitu riadiacich registrov obvodu UART, ktoré slúžia pre riadenie prenosu a kontrolu toku dát. Zvyšok tejto časti predstavujú procesy, vykonávajúce činnosť adresového dekodéru a výstupného multiplexoru tak, ako bolo uvedené v podkapitole o návrhu. Každý register je implementovaný ako proces, ktorý je citlivý na vstupný hodinový signál a resetovací signál. Register Baud Rate Divisor je v implementácii rozšírený o ďalší register, ktorý slúži ako záložný. Pri pokuse o zápis do registru Baud Rate Divisor v čase, kedy je vysielací kanál aktívny, je nová hodnota uložená do záložného registru a prenos prebieha naďalej v závislosti na starej hodnote. Po deaktivácii vysielacieho kanálu je hodnota zo záložného registru prepísaná do hlavného registru.

V časti dataflow prebieha odvodzovanie existencie prerušení so zdrojom vo vysielacej alebo prijímacej časti. Výsledné príznaky sú propagované na výstupné vodiče tejto komponenty a zároveň do registru Interrupt Pending.

Syntéza

Výsledný obvod je plne syntetizovateľný hardvéri. Pre syntézu bol použitý nástroj **Xilinx ISE 13.1**. Ako cieľová technológia bol zvolený čip **XC3S50** z rodiny **Spartan-3**. V tabuľke 3.1 sú uvedené základné výstupy zo syntézy.

Frekvencia [MHz]	136.466
LUTs	438
Flip Flops	329
IOBs	92

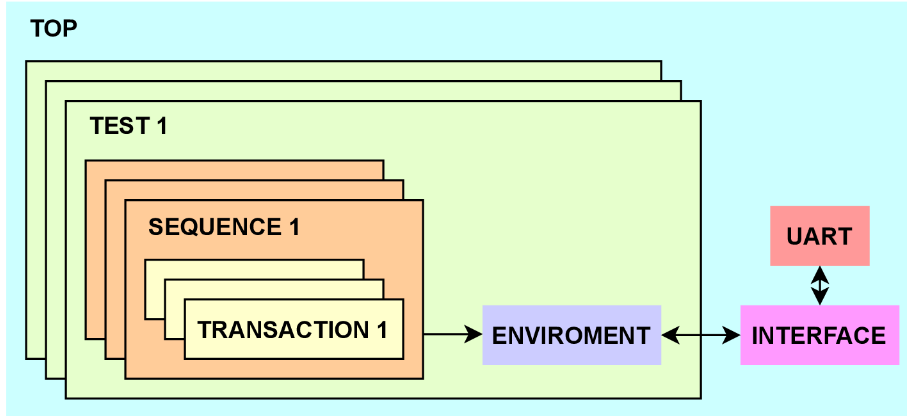
Tabuľka 3.1: Základné výstupy zo syntézy obvodu UART

3.4 Verifikácia

Táto podkapitola obsahuje popis procesu verifikácie obvodu UART pozostávajúci zo spôsobu implementácie verifikačného prostredia, prehľadu použitých verifikačných scenárov a zhodnotenia dosiahnutých výsledkov.

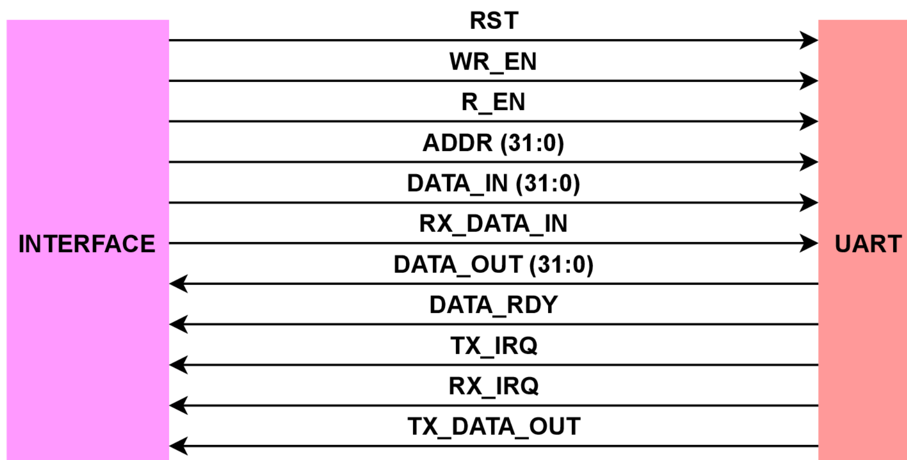
3.4.1 Verifikačné prostredie

Pre verifikáciu funkcionality implementovaného obvodu bolo použité verifikačné prostredie vytvorené podľa verifikačnej metodiky UVM. Toto prostredie je implementované pomocou objektov vytvorených z tried programovacieho jazyka SystemVerilog.



Obr. 3.11: Schéma verifikačného prostredia na najvyššej úrovni

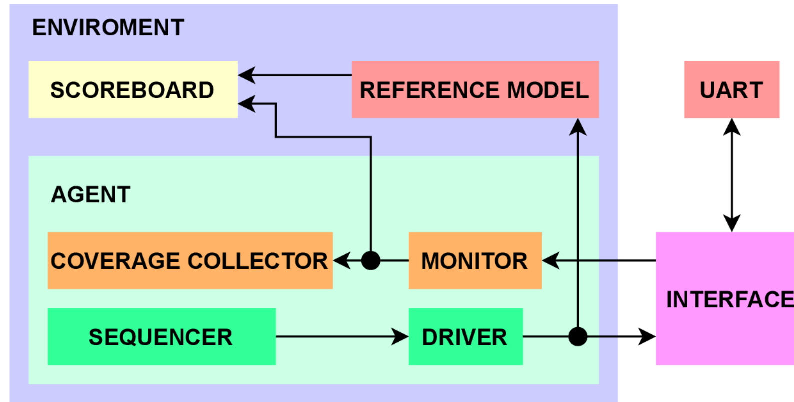
Najvyššiu úroveň tohoto prostredia predstavuje modul *top*, viď obrázok 3.11. Tento modul pred začiatkom verifikácie inštanciuje obvod UART a počas priebehu verifikácie zaisťuje generovanie hodinového signálu, a spúšťanie jednotlivých testov. Dané testy predstavujú objekty odvodené z triedy *uvm_test*. V každom z týchto testov sú vytvárané sekvencie odvodené z triedy *uvm_sequence*, ktoré definujú scenár konkrétneho testu a sú privádzané do komponenty *enviroment* z triedy *uvm_env*. Tieto sekvencie sú tvorené postupnosťou transakcií z triedy *uvm_sequence_item*, ktoré popisujú priebeh danej sekvencie. Formát použitých transakcií je znázornený na obrázku 3.12, pričom je odvodený od rozhrania obvodu UART uvedeného v podkapitole o návrhu.



Obr. 3.12: Formát transakcie používanej v implementovanom verifikačnom prostredí

Komponenta *interface* slúži ako rozhranie pre komunikáciu s verifikovaným obvodom na úrovni jednotlivých transakcií. Účelom komponenty *enviroment*, ktorá je zobrazená na obrázku 3.13, je monitorovanie miery dosiahnutého pokrytia, transformácia vstupujúcich

sekvencií na jednotlivé transakcie, ich následné prenesenie na vstup verifikovaného obvodu a jeho referenčného modelu, a porovnanie ich odozvy na dané vstupy. Porovnávanie výstupov verifikovaného obvodu a referenčného modelu je činnosťou komponenty *scoreboard*, ktorá je inštanciovaná z triedy *uvm_scoreboard*. Referenčný model implementuje funkcionalitu obvodu UART podľa špecifikácie, pričom je odvodený z triedy *uvm_subscriber*.



Obr. 3.13: Schéma komponenty *enviroment* používanej verifikačným prostredím

Komponenta *agent* odvodená z triedy *uvm_agent* v sebe zapúzdruje dve aktívne a dve pasívne podkomponenty. Do aktívnej komponenty *sequencer*, implementovanej ako objekt z triedy *uvm_sequencer* sú privádzané jednotlivé sekvencie z aktuálne prebiehajúceho testu. Každá sekvencia je následne prenesená do objektu *driver* z triedy *uvm_driver*. Tento objekt slúži ako aktívna komponenta, ktorá rozdelí sekvenciu na jednotlivé transakcie a tie posiela ďalej na rozhranie, a tiež do referenčného modelu. Pasívna komponenta *monitor* inštanciovaná z triedy *uvm_monitor* sleduje hodnoty signálov na rozhraní, pričom vstupné hodnoty posiela do pasívnej komponenty *coverage collector*, ktorá je odvodená z triedy *uvm_subscriber* a monitoruje mieru dosiahnutého definovaného pokrytia. Výstupné hodnoty signálov sú posielané do komponenty *scoreboard*.

3.4.2 Verifikačné scenáre

V tejto podkapitole je uvedený stručný prehľad použitých verifikačných testov. Bola použitá kombinácia priamych a pseudonáhodných testov, ktoré sú rozdelené do skupín pre testovanie registrového rozhrania, činnosti vysielacej časti obvodu, činnosti prijímacej časti obvodu a funkcionality prerušení.

Registre

- Resetovanie obvodu a postupné čítanie z jednotlivých registrov pre overenie ich iniciálnych hodnôt (priamy test).
- Zapisovanie náhodných hodnôt do registrov, do ktorých je z definície možné zapisovať, a následné čítanie z nich pre overenie správnej odozvy na zápis (náhodný test).
- Zapisovanie do registrov, do ktorých z definície nie je možné zapisovať (priamy test).
- Zapisovanie a čítanie z adries, ktoré neodpovedajú adresám registrov obvodu (náhodný test).

- Aktualizácia políčka *div* v registri Baud Rate Divisor pri aktívnom vysielacom kanáli (priamy test).
- Aktualizácia políčka *div* v registri Baud Rate Divisor pri aktívnom prijímacom kanáli (priamy test).
- Aktualizácia políčka *div* v registri Baud Rate Divisor pri aktívnom vysielacom aj prijímacom kanáli (priamy test).
- Aktualizácia políčka *div* v registri Baud Rate Divisor pri oboch kanáloch neaktívnych (priamy test).

Vysielacia časť

- Zápis do vysielacej FIFO, aktualizácia deliaceho pomeru pre odvodenie prenosovej rýchlosti, aktivácia vysielacieho kanálu a následné odoslanie bytu, a sledovanie výstupnej sériovej linky (priamy test).
- Spustenie odosielania bytu, načasovanie ďalšieho zápisu do vysielacej FIFO presne na moment, kedy z nej bude čítať výstupná komponenta. Sledovanie výstupnej sériovej linky pre overenie správneho prečítania pri súčasnom zápise (priamy test).
- Náhodná aktivácia a deaktivácia vysielacieho kanálu (náhodný test).
- Odoslanie bytu s dvoma stop bitmi (priamy test).
- Zmena počtu stop bitov z dvoch na jeden pri odosielaní prvého stop bitu (priamy test).
- Naplnenie vysielacej FIFO, nastavenie deliaceho pomeru, aktivácia vysielacieho kanálu, pokus o zápis do plnej FIFO a sledovanie odosielaných dát (priamy test).
- Resetovanie obvodu pri odosielaní bytu (náhodný test).
- Deaktivácia vysielacieho kanálu v momente, kedy je výstupná komponenta v jednom zo stavov, kedy už prenos dokončí. Následné overenie prechodu do deaktivovaného stavu (priamy test).

Prijímacia časť

- Pokus o čítanie z prázdnej prijímacej FIFO (priamy test).
- Aktualizácia políčka *div* v registri Baud Rate Divisor, aktivácia prijímacieho kanálu, nastavenie štart bitu a následné prijatie bytu, a jeho prečítanie z prijímacej FIFO (priamy test).
- Načasovanie čítania z prijímacej FIFO presne na moment, kedy do nej zapisuje vstupná komponenta. Overenie správnosti prečítaných dát pri súčasnom zápise a čítaní (priamy test).
- Náhodná aktivácia a deaktivácia prijímacieho kanálu (náhodný test).
- Naplnenie prijímacej FIFO, prijatie ďalšieho bytu. Overenie kruhovosti FIFO a správnosti prijatých dát (priamy test).

- Nastavenie vstupnej sériovej linky do logickej nuly, aby vstupná komponenta prešla do stavu, kedy vzorkovaním testuje prítomnosť štart bitu. Následné nastavenie vstupnej sériovej linky do logickej jednotky a overenie odozvy vstupnej komponenty (priamy test).
- Resetovanie obvodu pri prijímaní bytu (náhodný test).
- Deaktivácia prijímacieho kanálu pri rôznych stavoch vstupnej komponenty (priamy test).
- Odoslanie štart bitu do vstupnej komponenty, nastavenie vstupnej sériovej linky na signál s vysokou frekvenciou a následné overenie správnosti prijatých dát (priamy test).

Prerušenia

- Povolenie prerušenia z vysielacej časti a odoslanie počtu bytov, ktorý spôsobí prerušenie. Následné sledovanie výstupného príznaku prerušenia z vysielacej časti a čítanie z registru Interrupt Pending (priamy test).
- Povolenie prerušenia z prijímacej časti a prijatie počtu bytov, ktorý spôsobí prerušenie. Následné sledovanie výstupného príznaku prerušenia z prijímacej časti a čítanie z registru Interrupt Pending (priamy test).
- Zmena prahu pre vyvolanie prerušenia vo vysielacej časti pri už existujúcom, alebo neexistujúcom prerušení a sledovanie odozvy (priamy test).
- Zmena prahu pre vyvolanie prerušenia vo prijímacej časti pri už existujúcom, alebo neexistujúcom prerušení a sledovanie odozvy (priamy test).
- Zakázanie prerušenia vo vysielacej časti pri existujúcom prerušení a sledovanie odozvy (priamy test).
- Zakázanie prerušenia v prijímacej časti pri existujúcom prerušení a sledovanie odozvy (priamy test).
- Povolenie prerušenia vo vysielacej časti v situácii, kedy by už bolo pri skoršom povolení vyvolané (priamy test).
- (priamy test) Povolenie prerušenia v prijímacej časti v situácii, kedy by už bolo pri skoršom povolení vyvolané (priamy test).
- Zhodenie príznaku prerušenia z vysielacej časti naplnením FIFO dostatočným počtom bytov (priamy test).
- Zhodenie príznaku prerušenia z prijímacej časti prečítaním dostatočného množstva bytov z FIFO (priamy test).

Zhodnotenie dosiahnutých výsledkov

Implementovaný obvod UART bol verifikovaný prostredníctvom kombinácie priamych a pseudonáhodných testov. Odozva obvodu bola pri všetkých verifikačných scenároch totožná s výstupmi použitého referenčného modelu. Pre spúšťanie verifikačných testov bol použitý

simulačný nástroj **QuestaSim 2019.4**. V tabuľke 3.2 sú uvedené dosiahnuté hodnoty pokrytia. Metrika pokrytia kódu nie je naplnená v celej miere z dvoch dôvodov. Prvým sú predvolené vetvy konštrukcie *case*, ktoré sú v jazyku VHDL povinné aj keď ich telo bude prázdne. Druhým dôvodom je používanie tej istej implementácie FIFO pre vysielaciu aj prijímaciu časť. Zápisy do plnej vysielacej FIFO obvod ignoruje a výstupná komponenta z nej nečíta, pokiaľ je prázdna. Oproti tomu pri prijímacej FIFO sú takéto akcie povolené.

Pokrytie kódu	**Funkčné pokrytie**
97.79%	100%

Tabuľka 3.2: Hodnoty pokrytia kódu a funkčného pokrytia dosiahnuté pri verifikácii

Kapitola 4

Záver

Táto práca bola zameraná na návrh, implementáciu a verifikáciu základnej periférie procesoru architektúry RISC-V podľa jeho špecifikácie. Ako reprezentant platformy RISC-V bol zvolený procesor FU540-C000 od spoločnosti SiFive a z jeho dostupných periférií bol vybraný obvod UART, slúžiaci pre asynchrónnu sériovú komunikáciu. Pre implementáciu bol zvolený programovací jazyk VHDL.

Pri analýze už existujúcich voľne dostupných implementácií obvodu UART vyšlo najavo, že ich integrácia do systému SiFive FU540-C000 by bola náročnejšia než vytvorenie vlastnej implementácie. Jednalo sa buď o veľmi jednoduché obvody, ktoré neposkytovali požadovanú funkcionalitu, alebo o skutočne komplexné a zverifikované obvody, ktoré sa však rozchádzali so špecifikáciou v rôznych aspektoch.

Výsledná implementácia obvodu UART je zverifikovaná pomocou verifikačnej metodiky UVM. Je tiež plne syntetizovateľná v hardvéri, splňuje požiadavky zo špecifikácie procesoru FU540-C000 a je kompatibilná s jeho linuxovým ovládačom.

Napriek tomu má implementovaný obvod stále priestor na zlepšovanie. Jednou z možností je rozšírenie jeho funkcionality tak, aby bol použiteľný aj v inej implementácii procesorovej architektúry RISC-V. Ďalším možným pokračovaním tejto práce by mohlo byť doplnenie obvodu o prevodník z implementovaného rozhrania do jedného alebo viacerých zbernicových protokolov, napríklad TileLink, Wishbone apod. V neposlednom rade stojí tiež za uváženie možnosť rozšírenia obvodu o ďalšiu perifériu, napríklad SPI.

Literatúra

[1] AccelleraTeam. *Universal Verification Methodology (UVM) User's Guide*. 1.2. Október 2015.

[2] AccelleraTeam. *Home* [online]. 2021. Aktualizované 2021 [cit. 5. mája 2021]. Dostupné z: https://www.accellera.org/.

[3] Bidlo, M. *Principy sériové komunikace, sériová komunikační rozhraní* [online]. November 2020. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIMP-IT%2Flectures%2F04-IMP-seriova_kom.pdf&cid=12166.

[4] Bourdeauducq, S. *Simple RS232 UART* [online]. 2010. Aktualizované 2010 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/mmuart.

[5] Cabal, J. *Simple UART for FPGA* [online]. 2019. Aktualizované 2019 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/simple_uart_for_fpga.

[6] Campbell, S. *Basics of UART Communication* [online]. 2020. Dostupné z: https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html#.

[7] Eric Peňa, M. G. L. *UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter* [online]. 2016. Dostupné z: https://www.circuitbasics.com/basics-uart-communication/.

[8] Fitzpatrick, T. *Basic UVM - Introducing Transactions* [online]. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-introducing-transactions.

[9] Fitzpatrick, T. *Basic UVM - Introduction to UVM* [online]. 2013. Dostupné z: https://verificationacademy.com/sessions/introduction-uvm.

[10] Fitzpatrick, T. *Basic UVM - Monitors and Subscribers* [online]. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-monitors-and-subscribers.

[11] Fitzpatrick, T. *Basic UVM - UVM "Hello World"* [online]. 2013. Dostupné z: https://verificationacademy.com/sessions/uvm-hello-world.

[12] International, R.-V. *Specifications - RISC-V International* [online]. 2021. Aktualizované 2021 [cit. 5. mája 2021]. Dostupné z: https://riscv.org/technical/specifications/.

[13] Jimblom. *Serial Communication* [online]. 2021. Dostupné z: https://learn.sparkfun.com/tutorials/serial-communication/all.

[14] Kořenek, J. *Introduction* [online]. September 2020. Dostupné z:
https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FPCS-IT%2Flectures%2Fintroduction.pdf&cid=13434.

[15] Kořenek, J. *Syntéza obvodů* [online]. Október 2020. Dostupné z:
https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FPCS-IT%2Flectures%2Fsynteza-obvodu.pdf&cid=13434.

[16] L., B. *UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter* [online]. 2018. Dostupné z:
https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2018/09/26/uart_-_majority_vote-EkEW.

[17] Lameres, B. J. *Introduction to Logic Circuits & Logic Design with VHDL*. 1. vyd. Cham: Springer International Publishing, 2017. ISBN 9783319341941.

[18] LeFevre, H. A. *A VHDL 16550 UART core* [online]. 2010. Aktualizované 2010 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/a_vhd_16550_uart.

[19] Litochevski, M. *UART to Bus* [online]. 2018. Aktualizované 2018 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/uart2bus.

[20] Lupas, O. *Serial UART* [online]. 2010. Aktualizované 2010 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/uart.

[21] MentorVerificationMethodologyTeam. *UVM Cookbook* [online]. 2013. Dostupné z: https://verificationacademy.com/cookbook/uvm.

[22] Meyer, A. A. S. *Principles of functional verification*. 1. vyd. Amsterdam : Boston: Elsevier ; Newnes, 2003. ISBN 0-7506-7617-5.

[23] Mohor, I. *UART 16550 core* [online]. 2018. Aktualizované 2018 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/uart16550.

[24] Morris, M. A. *SSP_UART* [online]. 2014. Aktualizované 2014 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/ssp_uart.

[25] Mrázek, V. *Úvod do problematiky návrhu hardware* [online]. 2021. Dostupné z:
https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FIVH-IT%2Flectures%2F01-uvod.pdf&cid=10356.

[26] Philippe. *Serial UART* [online]. 2010. Aktualizované 2010 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/miniuart2.

[27] Santos, L. A. dos. *Uart block* [online]. 2012. Aktualizované 2012 [cit. 6. mája 2021]. Dostupné z: https://opencores.org/projects/uart_block.

[28] SiFive, I. *SiFive FU540-C000 Manual*. V1p0. Apríl 2016. Dostupné z:
https://static.dev.sifive.com/FU540-C000-v1.0.pdf.

[29] SiFive, I. *Home - SiFive* [online]. 2021. Aktualizované 2021 [cit. 5. mája 2021]. Dostupné z: https://www.sifive.com/.

[30] Spear, C. *SystemVerilog for verification : a guide to learning the testbench language features*. 2nd ed. New York: Springer, 2008. ISBN 978-0-387-76529-7.

[31] Swaroop. *What is Serial Communication and How it works?* [online]. 2020. Dostupné z: https://www.codrey.com/embedded-systems/serial-communication-basics/.

[32] Walmsley, P. *Driver for the SiFive UART*. 2019. Dostupné z: https://patchwork.kernel.org/project/linux-riscv/patch/20190413020111.23400-3-paul.walmsley@sifive.com/.

[33] Zachariášová, M. *Verifikace číslicových obvodů* [online]. November 2020. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FPCS-IT%2Flectures%2Fverifikace_2020.pptx.

[34] Zachariášová, M. *Samo-kontrolní mechanizmy* [online]. Marec 2021. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cwk.php.cs?title=Main_Page&src=Lecture5_2021.pdf&ns=FVS&action=download&csid=721967&id=13245.

Prílohy

Zoznam príloh

A Obsah priloženého pamäťového média **51**

B Schéma výsledného obvodu **52**

Príloha A

Obsah priloženého pamäťového média

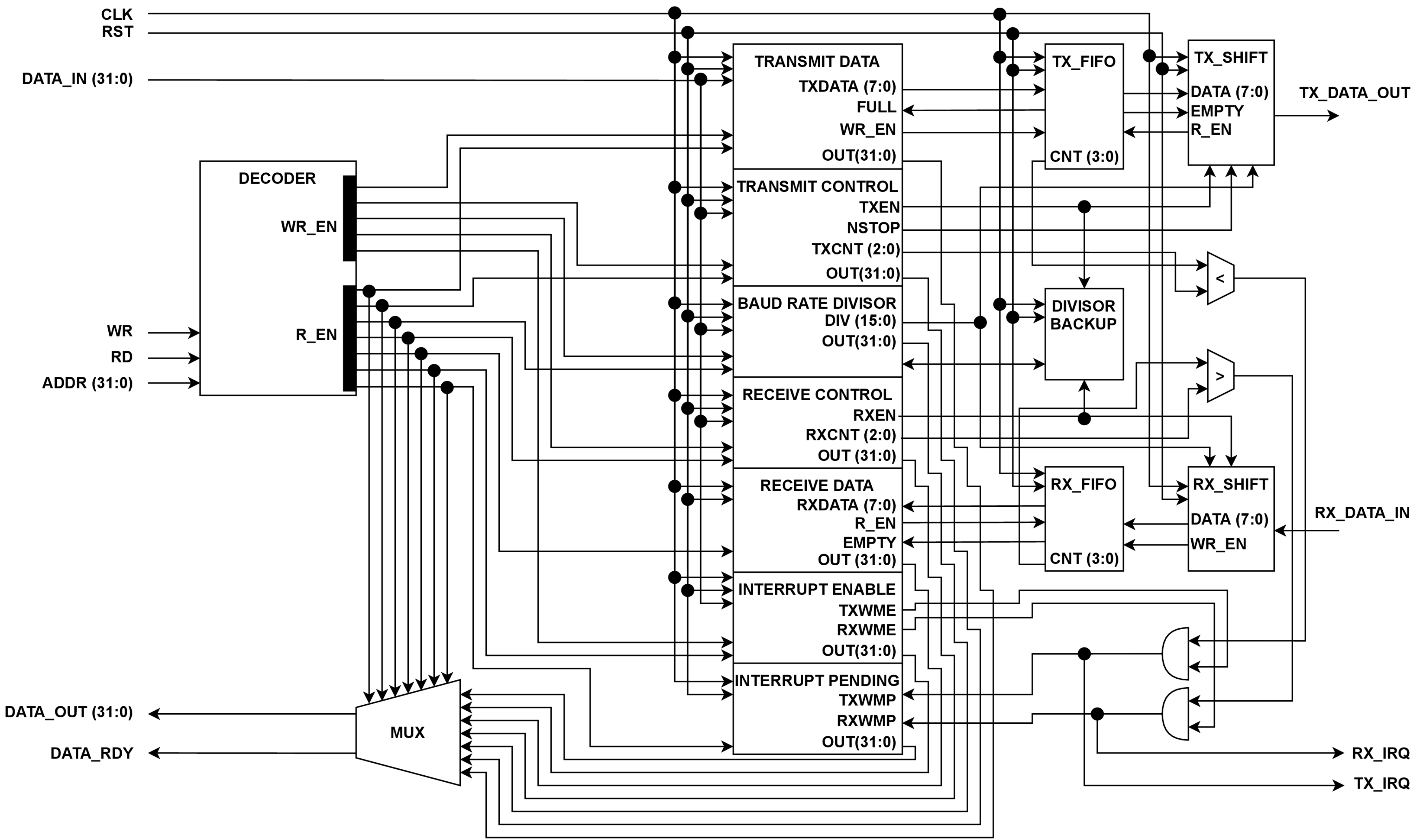
Priložené pamäťové médium obsahuje nasledujúce adresáre a súbory:

- adresár `src` – obsahuje všetky súbory a zdrojové kódy implementácie obvodu, a verifikačného prostredia, a tiež súbor `README.pdf`, ktorý obsahuje stručný návod pre používanie verifikačného prostredia,
- adresár `thesis` – obsahuje zdrojové súbory potrebné pre preklad a zostavenie technickej správy v LaTeXe,
- súbor `master_thesis.pdf` – technická správa vo formáte PDF.

Príloha B

Schéma výsledného obvodu

Bloková schéma celkového výsledného obvodu.


CLK
RST
DATA_IN (31:0)
DECODER
WR_EN
R_EN
WR
RD
ADDR (31:0)
TRANSMIT DATA
TXDATA (7:0)
FULL
WR_EN
OUT(31:0)
TRANSMIT CONTROL
TXEN
NSTOP
TXCNT (2:0)
OUT(31:0)
BAUD RATE DIVISOR
DIV (15:0)
OUT(31:0)
RECEIVE CONTROL
RXEN
RXCNT (2:0)
OUT (31:0)
RECEIVE DATA
RXDATA (7:0)
R_EN
EMPTY
OUT (31:0)
INTERRUPT ENABLE
TXWME
RXWME
OUT(31:0)
INTERRUPT PENDING
TXWMP
RXWMP
OUT(31:0)
TX_FIFO
CNT (3:0)
TX_SHIFT
DATA (7:0)
EMPTY
R_EN
TX_DATA_OUT
DIVISOR
BACKUP
RX_FIFO
CNT (3:0)
RX_SHIFT
DATA (7:0)
WR_EN
RX_DATA_IN
MUX
DATA_OUT (31:0)
DATA_RDY
RX_IRQ
TX_IRQ