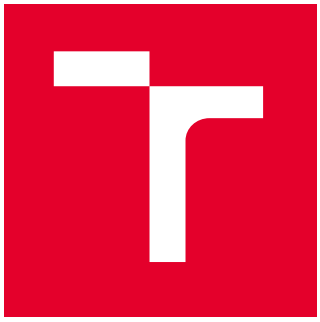
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2022 Bc. Lukáš Prusák



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

AUTOMATIZOVANÝ TESTBED PRO SIL/PIL TESTOVÁNÍ FIRMWARE POMOCÍ FPGA

AUTOMATED TESTBED FOR SIL/PIL TESTING OF EMBEDDED APPLICATION USING FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Lukáš Prusák**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Jakub Arm, Ph.D.**
SUPERVISOR

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Kybernetika, automatizace a měření**

Ústav automatizace a měřicí techniky

Student: Bc. Lukáš Prusák ***ID:*** 203328

Ročník: 2 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Automatizovaný testbed pro SIL/PIL testování firmware pomocí FPGA

POKYNY PRO VYPRACOVÁNÍ:

Úkolem je vytvořit testbed (na bázi test-bench) pro SIL/PIL simulaci embedded aplikace v prostředí FPGA. Testovací nástroj automatizovaně provede simulovaný běh aplikace dle zadaného předpisu, přičemž vytvoří report z chodu (profilace programu a zaznamenání sledu RTOS funkcí).

1) Proveďte rešerši vhodných soft-core.
2) Navrhněte architekturu testbedu a jeho komponentů.
3) Implementujte testbed se soft-core a vyhodnocovacími moduly.
4) Definujte předpis testovacího scénáře a vytvořte testovací dataset.
5) Proveďte testování a vyhodnoťte vlastnosti testbedu.

DOPORUČENÁ LITERATURA:

Broekman, Bart. Testing Embedded Software. Addison-Wesley, 2003. ISBN: 978-0321159861

Termín zadání: 7.2.2022 ***Termín odevzdání:*** 18.5.2022

Vedoucí práce: Ing. Jakub Arm, Ph.D.

doc. Ing. Petr Fiedler, Ph.D.
předseda rady studijního programu

ABSTRAKT

Diplomová práca sa zaoberá návrhom testbench na vybraný soft-core procesor NEORV32 architektúry RISC-V pre simulácie embedded aplikácií v prostredí FPGA. Testbench bol vytvorený v prostredí Vivado s cieľom jeho rozšírenia na testovací a validačný framework. Boli vybrané a implementované základné moduly ako GPIO, PWM, UART a PC. Pre tieto moduly bolo navrhnutých niekoľko testovacích scenárov. Testbench bol tiež doplnený o pomocné skripty, pre korektné hierarchické nastavenie projektu a spúšťanie testov. Práca ďalej navrhuje aj niekoľko možných spôsobov vylepšenia a rozšírenia testbenchu.

KĽÚČOVÉ SLOVÁ

CPU, soft-core, FPGA, testbed, testbench, HDL, VHDL, SystemVerilog, RISC-V, NEORV32, Vivado, tcl

ABSTRACT

The master's thesis deals with designing a testbench for a selected soft-core processor NEORV32 with a RISC-V architecture for simulations of embedded applications in an FPGA environment. The testbench was created in the Vivado environment with the aim of extending it to a testing and validation framework. Basic modules such as GPIO, PWM, UART, and PC were selected and implemented. Several test scenarios have been designed for these modules. The testbench has also been supplemented with additional scripts, to create hierarchically correct project setup and test execution. The work also suggests a few possible ways to improve and expand the testbench.

KEYWORDS

CPU, soft-core, FPGA, testbed, testbench, HDL, VHDL, SystemVerilog, RISC-V, NEORV32, Vivado, tcl

Vysadené pomocou balíčku `thesis` verzie 4.03; `http://latex.feec.vutbr.cz`

PRUSÁK, Lukáš. *Automatizovaný testbed pre SIL/PIL testovanie firmware*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačných technologií, Ústav automatizace a měřicí techniky, 2022, 73 s. Diplomová práca. Vedúci práce: Ing. Jakub Arm, Pg.D.

Vyhlásenie autora o pôvodnosti diela

Meno a priezvisko autora:	Bc. Lukáš Prusák
VUT ID autora:	203328
Typ práce:	Diplomová práca
Akademický rok:	2021/22
Téma závěrečnej práce:	Automatizovaný testbed pre SIL/PIL testovanie firmware

Vyhlasujem, že svoju záverečnú prácu som vypracoval samostatne pod vedením vedúcej/cého záverečnej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej záverečnej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto záverečnej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podpisuje iba v tlačenej verzii.

POĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce Ing. Jakubovi Armovi, Ph.D. za odborné vedenie, cenný čas pri konzultáciách, trpezlivosť a podnetné návrhy k práci.

Obsah

Zoznam obrázkov

Úvod

U klasických hard-core procesorov, ktoré sa používajú pri embedded aplikáciách je obtiažne testovať správnu funkcionalitu kódu, či samotného mikrokontroléru, kvôli zložitosti získavania dát o aktuálnom stave vstupov, výstupov CPU a periférií.

Cieľom tejto práce je vytvoriť testbench pre simuláciu embedded aplikácie pre vhodne vybraný soft-core procesor v prostredí FPGA. To umožní lepšie sledovanie aktuálneho stavu celého implementovaného SoC. Pred výberom soft-core procesora pre túto prácu je vhodné sa najprv zoznámiť s jeho vlastnosťami, ktoré sú dôležité pri návrhu testbenchu. Tie sú otvorenosť, podpora platforiem a nástrojov pre prácu s FPGA návrhom či ich verejná využiteľnosť v budúcnosti.

Zo zistených vlastností je cieľom vybrať najvhodnejšieho kandidáta medzi dostupnými soft-core procesormi. Pred návrhom testbenchu je vhodné sa zoznámiť s architektúrou vybraného procesora. Je potrebné vybrať vhodnú platformu, na ktorej sa bude navrhovať testbench v HDL jazyku, ktorý si je tiež potrebné zvoliť.

Ďalej sa táto práca zaoberá samotným návrhom testbenchu pre vybraný soft-core procesor s cieľom rozšírenia do väčšieho frameworku. Prioritou nie je pokryť všetky moduly a periférie, ktoré procesor poskytuje, ale skôr vytvoriť funkčnú štruktúru, ktorú bude do budúcna možné rozšíriť do testovacieho a validačného frameworku. Súčasťou tejto práce je aj vytvorenie softvéru, ktorý má za úlohu overiť funkčnost testbenchu. Ten je potrebné navrhnúť, aby bolo možné nie len otestovať samotnú funkcionalitu ale aj vyhodnotiť jeho vlastností.

1 Soft-core procesory

Soft-core procesor je špecifický model procesora, ktorý je hardvérovo definovaný jazykom (HDL) a ktorý môže byť upravovaný pre ľubovoľnú aplikáciu. V mnohých aplikáciách majú soft-core procesory výhody oproti procesorom navrhnutým na mieru ako nižšie náklady, flexibilita, nezávislosť od platformy a imunita voči zastaranosti. [1]

Nové priemyslové požiadavky znižujú životný cyklus mikrokontrolérov a mnoho procesorov sa stáva zastaralými v kratšom čase. Najväčšou výzvou pre hardvérových architektov je splnenie nových požiadavok ako vysoký výkon, energetická účinnosť a zníženie času návrhu produktu. Pre rýchly vývoj SoPC (System on Programmable Chip), mnoho komerčných procesorov ponúka s veľkou rozmanitosťou IP (Intellectual Property - duševné vlastníctvo) periférií. Vďaka soft-core procesorom je možné periférie jednoducho pridať alebo odobrať. [2] Dokonca je možné z soft-core procesora jednotlivé nepoužívané inštrukcie pre danú aplikáciu čím sa ďalej zefektívniť. [3]

V minulosti sa FPGA (Field Programmable Gate Array) primárne používali na vytváranie prototypov a debugovanie. Ale s ich stúpajúcou popularitou mnoho komerčných produktov teraz zahŕňa FPGA, vrátane Soft-core procesorov. Už v neskorých deväťdesiatych rokoch FPGA dodávatelia zaviedli SoC zariadenia, ktoré obsahovali jeden alebo viac Hard-core procesory a FPGA na jednom integrovanom obvode, čo umožnilo viac komplexnejší dizajn, ktorý zahŕňal ko-integráciu hardvéru a softvéru.

Zatiaľ čo tento spôsob poskytuje vyššie rýchlosti behu programu, ale neprináša flexibilitu modifikáciu pre aplikácie. Práve preto mnohí výrobcovia poskytujú riešenie používaním soft-core procesorov, ktoré môžu byť upravené vnútri FPGA. Podľa rôznych faktorov si dizajnér môže zvoliť Hard alebo Soft procesor podľa požiadavok danej aplikácie. [5]

Výhody Soft-core

- Flexibilita
- Nižšie náklady pri návrhu
- Rekonfigurovateľnosť

Nevýhody Soft-core

- Menšia rýchlosť
- Využitie zdrojov

1.1 ASIC procesory

ASIC (Aplication Specific Integrated Circuits) procesory, ako už z názvu vyplýva, sú vyvinuté pre jednu konkrétnu aplikáciu, ktorú budú vykonávať po celú dobu svojej životnosti.

Ak sa ASIC už raz aplikuje do kremíku, nie je možné ho zmeniť. Preto jeho vývin musí byť simulovaný a otestovaný, aby sa pri jeho finálnej implementácii vedelo, že bude fungovať správne a nenachádzajú sa v návrhu žiadne chyby. Preto sú počiatočné náklady na vývin podstatne vyššie ako pri FPGA prevedení, ktoré je možné kedykoľvek upraviť. [4]

Avšak po ich úspešnom vývine sú lacnejšie ako univerzálne Soft-core prevedenia a taktiež môžu dosahovať vyšších výkonov zároveň s menšími tepelnými stratami, keďže sú optimálnejšie navrhnuté. Preto sú vhodnejšie pri masových výrobách, kde sa výdaje pri počiatočnom vývine vykompenzované ziskami z následného predaja. [4]

Výhody ASIC

- Vyšší výkon
- Energetická efektivita
- Nižšia výrobná cena jedného kusu

Nevýhody ASIC

- Vysoké náklady pri návrhu
- Nemožné aktualizovať dizajn

1.2 Architektúra soft-core

Architektúra soft-core procesorov je v zásade rovnaká ako architektúra klasických ASIC procesorov, ale s tým rozdielom, že je procesor realizovaný na konfigurovateľnom hradlovom poli (FPGA), Soft-core procesory majú rovnaké parametre podľa, ktorých sa vyberajú pre konkrétnu aplikáciu:

Dĺžka slova - udáva počet bitov, s ktorými dokáže daný procesor pracovať. Bežne sa používa (8, 16, 32, 64) bitov. Zatiaľ čo menší počet bitov zabezpečuje jednoduchšiu hardvérovú architektúru procesora, tak väčší počet bitov poskytuje väčšií celkový výpočtový výkon. [6]

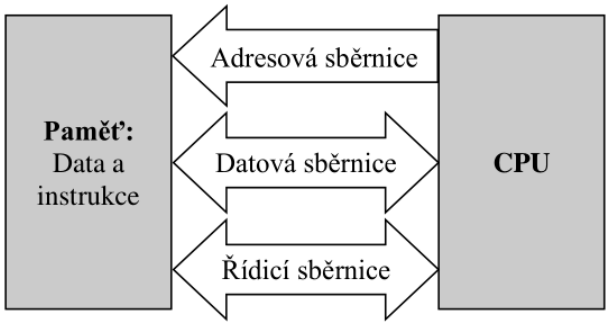
Inštrukčná sada

- RISC (Reduced Instruction Set Computer) - redukovaná inštrukčná sada, ktorá je celkovo rýchlejšia, ale zaberá viacej miesta v pamäti, keďže zložitejšie inštrukcie skladá z jednoduchších.
- CISC (Complex Instruction Set Computer) - komplexná inštrukčná sada obsahuje väčšie množstvo inštrukcií, ktoré sú komplexnejšie a ich spracovanie trvá dlhšiu dobu.

[6]

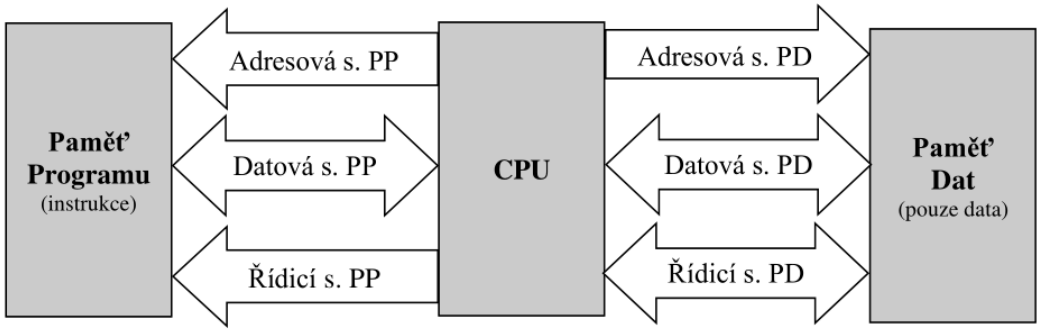
Architektúra

- Von Neumannova - spoločná pamäť programu a dát. Obr.č. 1.1



Obr. 1.1: Bloková schéma Von Neumannovej architektúry. [11]

- Harvardská - oddelená pamäť programu a dát. Obr.č. 1.2



Obr. 1.2: Bloková schéma Hardvardskej architektúry. [11]

1.2.1 ALU

Základnou súčasťou každého procesora, či už Hard-core alebo Soft-core je ALU, ktorá je schopná vykonávať logické operácie ako AND, OR, EX-OR, NOT atď. a taktiež aritmetické operácie ako sčítanie a odčítanie. Dĺžka slova určuje následne,

aký počet bitov dokáže ALU spracovať v jednom cykle. Inštrukcie a dáta z pamäte do ALU privádza radič za pomoci registrov, ktoré sú taktiež prispôsobené na rovnakú dĺžku slova. ALU následne vykoná danú inštrukciu s dodanými dátami a vypočíta výsledok.

Dĺžka slova, frekvencia vykonávania inštrukcií a komplexnosť inštrukcie následne udávajú výkon procesora. So zvyšujúcou sa komplexnosťou inštrukcií sa taktiež zvyšuje veľkosť ALU, ktorá tak odvádza viacej tepla. Je preto potrebné nájsť optimálny pomer veľkosti/komplexnosti ALU. [7]

1.2.2 Radič

Radič je podstatne komplexnejší oproti ALU, ktorá má jedinú úlohu, a to vykonať požadovanú inštrukciu. Jeho veľkosť a komplexnosť taktiež záleží od množstva inštrukcií, ktoré musí ovládať. Okrem poskytovania inštrukcií a dát pre ALU je jeho úlohou aj spracovanie prerušení, interných chýb či komunikácia so zbernicami. V drvivej väčšine prípadov sa používa procesor pipeline, ktorý rozdeľuje vykonávanie inštrukcie na jednotlivé časovo približne rovnako trvajúce fázy pre zvýšenie efektivity procesora. Túto úlohu má zase na starosti radič. [8]

1.2.3 Registre

Ako už bolo spomenuté v predošlých častiach 1.2.1 a 1.2.2, pre správnu funkciu CPU je potrebné dostať z pamäte inštrukcie a dáta. Registre sú špeciálny typ pamäte, ktorý je priamo zakomponovaný do procesora, čo podstatne urýchľuje prístup radiča k nim, keďže nie je potrebný komplikovaný proces posielania dát cez zbernicu. Pre zachovanie rýchlosti a veľkosti CPU je táto pamäť veľmi malá. [9]

Existujú rôzne typy registrov, kde ich veľkosť je rovnako ako pri ALU a radiči určená dĺžkou slova, v niektorých prípadoch môžu byť však násobne väčšie. [10]

- **Akumulátor** - najčastejšie využívaný register, ktorý sa používa na uloženie dát z pamäte.
- **Adresové registre** - slúžia na uloženie adresy v pamäti, v ktorej sa nachádzajú požadované dáta.
- **Dátové registre** - obsahujú dáta, ktoré sa majú zapísať alebo prečítať z adresy danej adresovým registrom.
- **Všeobecné registre** - slúžia na dočasné uloženie dát.
- **Programový čítač** - sleduje pozíciu nasledujúcej inštrukcie v programe.

- **Inštrukčný register** - obsahuje inštrukciu, ktorá sa bude vykonávať ako nasledujúca.
- **Stavový register** - tento register obsahuje rôzne vlajky (*flags*), ktoré indikujú stav prebiehajúcich operácií (`Carry, Overflow, Zero, Negate, Extend`).
- **Kontrolné registre** - tieto registre sú špecifické pre mikrokontroléri a slúžia na rýchlu komunikáciu s perifériami.

1.2.4 Pamäť

Ako už bolo naznačené v kapitole 1.2, procesor môže mať dve základné hierarchie (Harvardská, Von Neumanova). Väčšina mikroprocesorov používa práve Harvardskú architektúru, ktorá má fyzicky oddelený pamäťový priestor pre inštrukcie programu a dáta.

Keďže podľa princípu lokalít aj pri časovej a priestorovej, alebo pre inštrukcie a dáta platí, že ak je použité nejaké inštrukcie alebo dáta, dá sa očakávať, že v blízkej dobe, budú použité znovu, alebo budú použité inštrukcie, či dáta na blízkych adresách. Preto sa podľa týchto princípov ďalej pamäte delia od rýchlejších a menších po pomalšie a väčšie. Kde najrýchlejšia dostupná pamäť pre procesor sú už spomínané registre v časti 1.2.3. [11]

Cache

Cache pamäť je najlbližšia a najrýchlejšia pamäť pre CPU po registroch. Pred tým než procesor potrebuje čítať z operačnej pamäte, najprv skúsi, či sa dáta nenachádzajú v pamäti cache. Ak sa v nej dáta nenachádzajú pokračuje do pomalšej operačnej pamäte. So zvyšujúcou kapacitou sa taktiež znižuje rýchlosť pamäte, preto sa znovu podľa princípu lokalít rozdeľuje na ďalšie levely označované L1, L2, L3, postupne od najnižšej veľkosti a najvyššej rýchlosti. Level cache pamäte L3 sa vyskytuje iba pri viacjadrových procesoroch, ktoré sa pri mikrokontroléroch často nevyskytujú a v tejto práci je ich zložitosť nežiaduca. [11]

Operačná pamäť

Operačná pamäť slúži na uloženie všetkých potrebných dát alebo inštrukcií, podľa hierarchie procesora.

Podobne ako registre aj pamäť cache je rozdelená na rovnako veľké bloky, ktorých veľkosť je určená dĺžkou slova. Jednotlivé bloky sú následne adresované za pomoci adresového registra. [11]

Rovnako cache pamäte aj operačná pamäť je na ASIC procesoroch umiestnená oddelene ako externá časť CPU. Pri Soft-core prevedení, kde FPGA predstavuje jeden SoC (System on Chip), čo znamená, že rýchlosti logiky a pamäte sú oveľa bližšie. Taktiež pri Soft-core procesoroch je možné túto pamäť doplniť, ak to veľkosť FPGA umožnuje. [3] [5]

1.3 Vlastnosti soft-core

Pred praktickou časťou je nutné si vybrať konkrétny soft-core procesor, pre ktorý bude testbench špecificky navrhnutý. Tento výber bude uskutočnený podľa vopred očakávaných vlastností, prípadne skúseností pri práci s ním.

V nasledujúcich kapitolách budú popísané vlastnosti, ktoré sú všeobecne vhodné pre ktorýkoľvek soft-core či hard-core procesor, ale aj konkrétne vhodné pre pokračovanie tejto práce. Následne sa urobí porovnanie jednotlivých možných variant soft-core procesorov a nakoniec sa vyberie najvhodnejší kandidát, s ktorým sa bude ďalej pracovať. Ak by nastal problém v pokročilejších fázach návrhu testbenchu, je možné si podľa tohto porovnania znovu vybrať.

1.3.1 Open-Source

Open-source je síce zdrojový kód, ktorý je voľne dostupný, no niektoré spoločnosti aj napriek voľne dostupného kódu ho neumožňujú meniť.

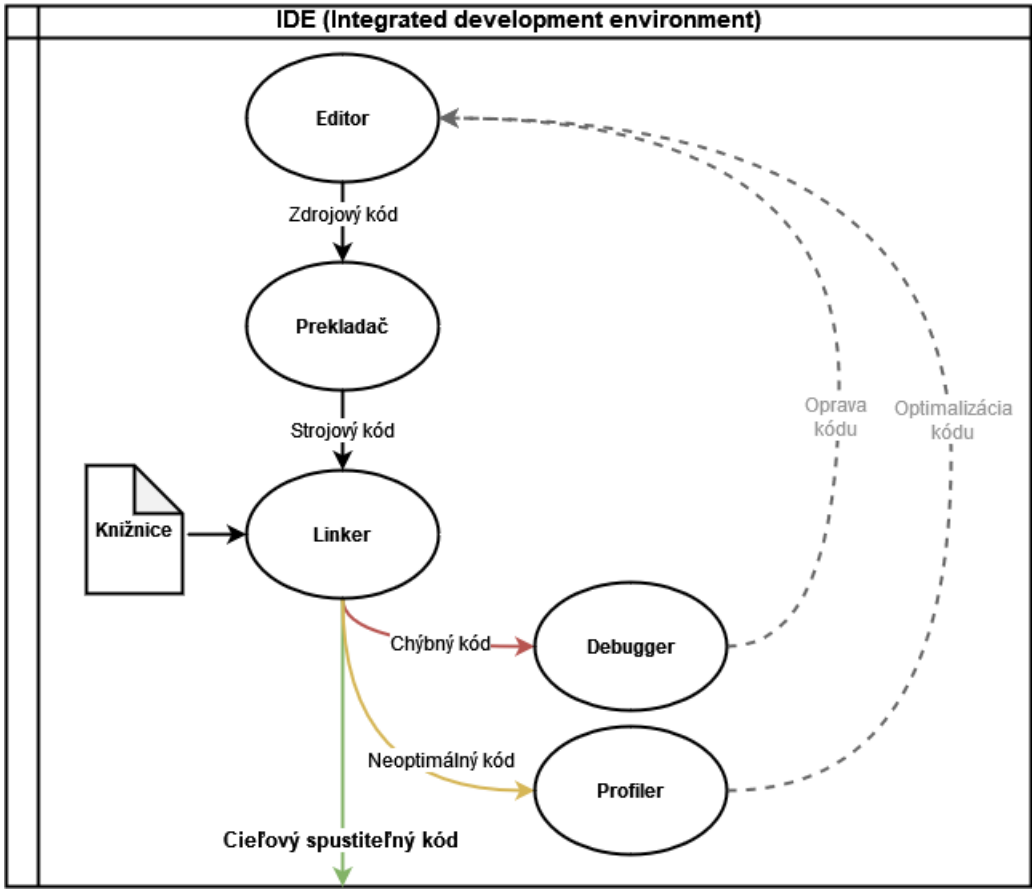
Na trhu je mnoho soft-core procesorov, ktoré sú ale intelektuálnym vlastníctvom originálneho dizajnéra a on určuje licenčné podmienky. Takéto hardvérové alebo softvérové vlastníctvo sa nazýva IP core (Intelectual Property Core).

Biznis niektorých spoločnosti je založený iba na licencovaní nimi vytvoreného intelektuálneho vlastníctva. Ak si druhá spoločnosť zakúpi licenciu na IP core, zvyčajne dostane všetko potrebné pre návrh, testovanie a používanie. Zakúpenie IP core často zahrňuje aj práva na možnú zmenu pôvodného dizajnu. Niekedy sú však poskytnuté vo formáte nízkeho jazyka (napr. rozloženie tranzistorov), potom ich držitelia licencie nemôžu výrazne upravovať.

1.3.2 Nástroje

Pre prácu s implementovanými soft-core procesormi v HDL jazyku sú potrebné určité nástroje na ich programovanie, simuláciu a prípadnú implementáciu firmvéru na samotný soft-core. Výber platformy pre návrh soft-core a testbenchu v HDL jazyku

je popísaná v časti 1.4.2. Táto časť sa ale bude zaoberať nástrojmi, ktoré by malo všeobecne obsahovať IDE (Integrated Development Environment), čiže prostredie na návrh firmvéru pre soft-core procesor, najčastejšie v jazyku C/C++. Taktiež je podstatné, aby tieto jednotlivé nástroje boli kompatibilné s vybraným soft-core procesorom.



Obr. 1.3: Krížová mapa IDE.

Editor

Editor je prvá časť IDE pre návrh firmvéru pre vstavené systémy. Táto časť nám umožňuje písať zdrojový kód, ktorý potom teiž spracovávajú ďalšie časti.

Editor môže obsahovať aj rôzne funkcie, ako je zvýrazňovanie syntaxe, ktoré zvyšuje prehľadnosť kódu alebo automatické dopĺňanie kódu, ktoré urýchľuje jeho návrh.

Tento zdrojový kód, ktorý je pre vstavané systémy väčšinou písaný v jazyku C/C++, je následne spracovaný prekladačom, ktorý je popísaný v nasledujúcej časti 1.3.2.

Prekladač

Zdrojový kód, ktorý bol napísaný v editore, nemôže byť ihneď spracovaný procesorom, ale najprv je nutné ho prepísať do strojového jazyka. Túto úlohu vykonáva prekladač a taktiež aj iné kontroly, či prenesený zdrojový kód je správny. Následné chyby a upozornenia ohlási užívateľovi v rámci IDE. [12] Zdrojový a cieľový (strojový) kód musia byť vzájomne ekvivalentné, čiže pre všetky hodnoty vstupov musia výstupy nadobúdať rovnaké výsledky. [13]
Podľa spracovania zdrojového kódu v čase existujú dva typy prekladačov: [13]

- **Kompilačný prekladač** - cieľový (strojový) kód je zostavený pred samotným behom programu.
- **Interpretačný prekladač** - zdrojový kód sa spracováva po častiach, prekladá sa a zároveň vykonáva hneď po spracovaní jednej časti. Tento typ prekladača je pomalší ako kompilačný.

Linker

Ako z názvu vyplýva, slúži na spájanie súborov a knižníc. Keďže preložené súbory do strojového jazyka môžu byť rôzne rozdelené, je ich potrebné spojiť dokopy v jeden spustiteľný program. Linker má za úlohu taktiež odhaľovať chyby, napríklad ak by nejaký súbor či knižnica neboli dostupné. [12] Proces linkovania nastáva aj hneď po prekladaní, ale zároveň aj pri procese načítavania, keď sa nahráva program do pamäti pomocou loadera.

Debugger

Debugger slúži na testovanie funkčnosti programu tak, že umožňuje pozorovať program počas jeho behu v reálnom čase. Umožňuje taktiež program pozastavovať v konkrétnom bode, zobrazovať stav pamäte či registrov alebo manuálne upravovať ich hodnoty. [12]

Niektoré debuggery majú funkciu, ktorá umožnuje spätné debuggovanie. Tieto debuggery dokážu program spúšťať po krokoch späť v čase. Táto funkcia je však použiteľná len v špecifických prípadoch, preto sa používa len zriedka. [14]

Pri vstavaných systémoch sa dubuggovanie môže vykonávať na konkrétnom hardvéri, v tomto prípade soft-core prevedenia procesora na FPGA. Skenovanie stavu CPU je zabezpečené pomocou technológie JTAG (Joint Test Action Group). Procesor je zvyčajne možné aj simulovať, ak hardvér nie je ešte dostupný alebo nie je vhodné riskovať jeho zničenie.

Profiler

Tento nástroj slúži na vyhľadávanie miest v programe, ktoré by mohli byť vhodné pre optimalizáciu. Profiler môže sledovať výskyt jednotlivých funkcií, čas strávený v nich, využívanie pamäte alebo volanie prerušení. Podľa týchto údajov je možné následne určiť efektivitu programov. [15]

Pri používaní profileru môže nastať určité skreslenie jeho výstupných údajov vyvolané tým, že samotné sledovanie programu profilérom zaberie nejaký výpočtový výkon a priestor v pamäti. [15]

1.3.3 Popisný jazyky pre FPGA

Všeobecne sa jazyky pre popis hardvéru označujú skratkou HDL (Hardware Description Language). Namiesto toho aby sa generoval počítačom spustiteľný program, HDL prekladač poskytne hradlovú mapu. Táto hradlová mapa môže byť stiahnutá do programovateľného zariadenia FPGA, kde sa následne kontroluje požadovaná funkcionalita obvodu. [16]
HDL jazyky popisujú digitálny obvod v troch rôznych leveloch: [16]

- **Structural** - popisuje štruktúru jednotlivých komponentov
- **Behavioral** - popisuje, ako by sa mal obvod správať
- **Dataflow** (Gate level) - popisuje tok dát z vstupu na výstup na hradlovej úrovni

Medzi klasickými programovacími jazykmi ako C a HDL jazykmi je jeden podstatný rozdiel. Jednotlivé inštrukcie v HDL kóde sa vykonávajú paralelne, ak nie je časť kódu určená ako proces, kde sa kód rovnako ako v programovacích jazykoch vykonáva sekvenčne.

Keď píšeme klasický program, rozumieme tým, že procesor bude spúšťať jednotlivé riadky jeden za druhým, zhora nadol, rovnako, ako to funguje pri čítaní. Ale v kóde HDL sa popisuje digitálny hardvér, kde jednotlivé časti hardvéru môžu fungovať súčasne, aj keď popis kódu je organizovaný rovnako ako pri klasickom programovaní, zhora nadol. Preto sa vytvorili rôzne typy dátových objektov, ktoré určujú, ako sa budu správať.

Signál (Wire) - Používa sa k popisu vstupov, výstupov alebo vnútorných prepojení. Signál môže byť tiež považovaný za reálny drôt alebo prepojenie v schéme, ktoré má určitú hodnotu. Signál sa zvyčajne nepoužíva v štruktúre proces, pretože

svoju hodnotu zmení až na konci celého procesu, preto je v niektorých situáciách vhodnejšie používať premennú.

Premenná (Variable) - Funguje ako klasické premenné v programovacom jazyku, kde pri zápise mení svoju hodnotu okamžite. Premenná však, môže byť deklarovaná iba v rámci procesu.

Konštanta (Constant) - Keďže konštanty nemôžu meniť svoju hodnotu počas behu programu, nemajú podobné problémy ako signály a premenné. Konštanta môže byť definovaná v rámci použitej architektúry alebo iba v rámci procesu, v ktorom bola deklarovaná. [17]

Medzi najpoužívanejšie jazyky patrí Verilog a VHDL.

VHDL

Narozdiel od Verilogu je VHDL podrobnejší a veľmi typizovaný a hardvérový popisný jazyk. Vďaka tomu sa považuje za samo-dokumentujúci, keďže syntax si vyžaduje väčší obsah kódu. Z toho vyplýva, že písanie kódu vo VHDL môže trvať dlhšie ako vo Verilogu, ale je ľahšie odhaliť chybu kvôli presne danej štruktúre a prehľadnejšiemu popisu.

Verilog

Verilog ako hardvérový popisný jazyk je vhodnejší pre modelovanie na nízkej úrovni, keďže je odvodený z programovacích jazykov C a Hilo. Je pomerne obmedzený a slabo typizovaný jazyk, pretože má všetky dátové typy preddefinované. Vývojári ho považujú za flexibilný a stručný jazyk, ale to je aj jeho nevýhodou, keďže je ťažšie odhaliť chybu kvôli nejednoznačnosti, hlavne ak sa pri kódovaní nedodržiavajú správne praktiky. Vďaka týmto vlastnostiam umožňuje vývojárom rýchlo navrhnúť požadované modely.

SystemVerilog

SystemVerilog je mix HDL a HVL (Hardware Verification Language), čiže ide skôr o jazyk, ktorý sa špecializuje na overovanie harvérových dizajnov. SystemVerilog je Verilog rozšírený o ďalšiu funkcionalitu, čo umožnuje dizajn, simulácia, testovanie a uvedenie harvéru do prevádzky. [18]

Drvivá väčšina testbencov je písaná práve v jazyku SystemVerilog, pretože má mnoho rozšírení oproti Verilogu, ktoré umožnuje verifikovať dizajny a vytvárať komplexnejšie štruktúry testbenchov. Preto aj v tejto práci sa bude používať SystemVerilog na vytváranie testbenchu.

1.3.4 Periférie

Ako klasické hard-core, tak aj soft-core mikrokontroléry môžu obsahovať rôzne periférie, pomocou ktorých komunikujú s ostatnými vstupno-výstupnými zariadeniami. Medzi základné patria:

- **GPIO** (General Purpose Input and Output)
- **Časovač**
- **PWM** (Pulse Width Modulation)
- **ADC** (Analog to Digital Converter)
- **Serialová komunikácia** (UART, SPI, I2C, Ethernet)
- **Externá pamäť** (EEPROM)
- **JTAG** - slúži na nahranie firmvéru a debugovanie

1.3.5 RTOS

RTOS (Real Time Operating System) sa tvária, že umožňujú spustenie viacerých programov súčasne. V skutočnosti každé jadro procesora dokáže vykonávať iba jednu činnosť v akomkoľvek okamihu. Časť tohto operačného systému nazývaná plánovač (scheduler) je zodpovedná za rozhodovanie spúšťania jednotlivých programov, čo vytvára ilúziu súčasného vykonávania rýchlym prepínaním medzi jednotlivými programami.

Plánovač RTOS je navrhnutý tak, aby poskytoval predvídateľný vzor vykonávania. To je obzvlášť zaujímavé pre vstavané systémy ako mikrokontroléry, pretože ako bolo spomenuté, vstavené systémy majú často požiadavku fungovania v reálnom čase a to je možné, len ak sa dokáže predvídať správanie plánovača operačného systému. Tradične sa táto predvídavosť (determinizmus) dosahuje tým, že užívateľ priradí každej úlohe (vláknu) prioritu vykonávania. Plánovač sa následne rozhoduje podľa priority, aby vedel, ktoré vlákno má spustiť ďalšie. [19]

RTOS môžu taktiež umožňovať logovanie rôznych parametrov počas behu programu, ktoré neumožňujú ani bežné nástroje na debugovanie.

- **Zoznam úloh**
- **Čas strávený v jednotlivých úlohách**

- **Výskyt prerušení**
- **Chyby**
- **...**

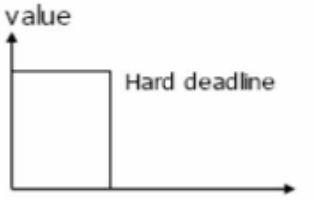
Tieto dáta môžu byť následne využité pri testovaní programu či mikrokontroléru pomocou testbedu.

Riadenie v reálnom čase

Pre spracovanie údajov v reálnom čase je nutné, aby dané zariadenie poskytovalo takmer instantný výsledok. Pre takéto spracovanie je taktiež nutné neustále privádzanie vstupných dát, aby sa neprerušil tok dát zo vstupu na výstup. [20]

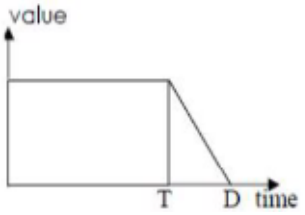
Riadenie v reálnom čase v mikrokontroléroch umožnuje práve RTOS, ktorý poskytuje logický výsledok v rámci požadovaného deadlinu. RTOS špecifikuje maximálny čas pre každú operáciu, ktorú vykonáva. Na základe stupňa tolerancie pre dodržiavanie termínov sú RTOS rozdelené do nasledujúcich kategórií. [21]

- **Hard deadline** - zmeškanie tohto deadlinu môže mať za následok katastrofálne zlyhanie systému, preto nie je možné výslednú hodnotu po termíne použiť. Obr.č. 1.4



Obr. 1.4: Graf hodnoty výsledku v čase pre Hard deadline. [21]

- **Firm deadline** - zmeškanie znižuje tzv. kvalitu výslednej hodnoty, až napokon ju stratí úplne a je rovnako nepoužiteľná ako v prípade Hard deadlinu. Obr.č. 1.5

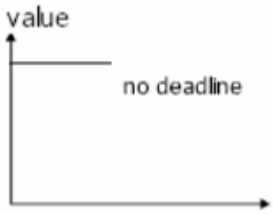


Obr. 1.5: Graf hodnoty výsledku v čase pre Firm deadline. [21]

- **Soft deadline** - podobne ako Firm dead line výsledná hodnota po deadline stráca na kvalite, ale nevedie to ku katastrofálnym následkom. Obr.č. 1.6

Obr. 1.6: Graf hodnoty výsledku v čase pre Soft deadline. [21]

- **No deadine** - obr.č. 1.7



Obr. 1.7: Graf hodnoty výsledku bez deadlinu v čase. [21]

1.3.6 Komunita

Medzi dôležité vlastnosti pri výbere softvéru je komunita. Pod týmto pojmom sa berie do úvahy hlavne dostupnosť zdrojov o danom produkte, či už sa jedná o voľné dostupné zdrojové kódy, odborné práce spojené s daným produktom, návody na používanie alebo informácie, ako riešiť často vyskytujúce sa problémy. Dostupnosť zdrojov a informácií môže zabezpečiť kvalitnou užívateľskou podporou aj sprostredkovateľ duševného vlastníctva, pokiaľ ide o zakúpený produkt.

Veľkosť komunity môže znamenať, aj ako veľmi je procesor využívaný, čo priamo súvisí s užitočnosťou tejto práce, čiže väčšou pravdepodobnosťou, že táto práca bude nápomocná.

1.4 Porovnanie soft-core procesorov

Podľa popísaných vlastností je nutné si zvoliť vhodný soft-core procesor tak, aby umožňoval nasledujúci návrh testbenchu. Keďže návrh soft-core procesora pre FPGA nie je úlohou tejto práce a taktiež zakúpenie licenčných práv, ktoré by umožnilo návrh upravovať, je príliš nákladné, výber je obmedzený iba na soft-core procesory, ktoré sú open-source a ich zdrojové kódy sú verejne dostupné.

Cortex-M1/M3

Cortex procesory sú vyvinuté firmou ARM, ktorá je jednou z najpokročilejších. Obe Cortex-M1 a výkonnejšia verzia Cortex-M3 sú dostupné ako zdrojové kódy oficiálne od výrobcu, ktorý si ale uplatňuje právo na licenčné podmienky. Zdrojové kódy nie sú dostupné vo VHDL alebo Verilog jazyku, preto nie je ani len možné zistiť ich funkcionalitu a následne ich upravovať podľa potreby.

LEON 3/4

Leon 32-bitové procesory sú kompatibilné s architektúrou SPARC V8. Je voľne použiteľný na výskum a vzdelávacie účely s tým, že v prípade komerčného použitia je nutné kúpiť licenciu. Zdrojové kódy sú priamo prístupné na webe sprostredkovateľa a zároveň sú v jazyku VHDL, ktorý je možné podľa potreby upraviť. Okrem zdrojových kódov sú na webe prístupné aj rôzne simulačné nástroje ako aj RTOS a všetká potrebná dokumentácia.

Obe modely sú založené na Harvardskej architektúre, ktorá má oddelené dáta a inštrukcie. Pre zvýšenie výpočtového výkonu sú implementované v rámci SoC (System on Chip) aj ďalšie hardvérové prvky ako násobička a delička frekvencie a cache pamäte. ALU procesora vykonáva inštrukcie v 7-stupňovej pipeline, ktorá je nezvyčajná u podobných návrhoch procesora, viac stupňov ale nemusí znamenať vyššiu výpočtovú rýchlosť oproti iným procesorom s kratšími pipelinami. Taktiež disponuje Power-down módom, ktorý umožňuje znížiť spotrebu pri špecifických aplikáciách. [22]

Celkovo sa jedná o robustný, efektívny, konfigurovateľný a výkonný procesor hlavne v ASIC prevedení, kde môže dosahovať až desaťnásobné frekvencie oproti FPGA prevedeniu. Táto práca sa však zaoberá Soft-core prevedením na FPGA, kde možná maximálna frekvencia tohto procesora je jedna z najnižších v porovnaní s ostatnými možnosťami. [22]

Dostupné periférie [22]

- Časovače
- Ovládač prerusení
- UART
- GPIO
- USB
- RS232
- Ethernet

Pico/MicroBlaze

Oba procesory boli vyvinuté firmou Xilinx a patria medzi viac komerčne používané produkty. Preto rovnako ako pri procesoroch Cortex je možné ich zdrojový kód získať od majiteľa licencie na IP core, ale nie je možné ho upravovať alebo komerčne používať bez zaplatenia licencie.

Blaze procesory majú ale veľa konfigurovateľných parametrov, ktoré môžu nastaviť pri navrhovaní. Medzi ne patrí aj FPU (floating-point unit), hardvérová delička, Barrel shifter, dátové a inštrukčné pamäte cache, spracovanie výnimiek (exception handling), hardvérová debugovacia logika a ďalšie funkcie. [1]

Dostupné periférie [23]

- Časovače
- Ovládač prerusení
- Ovládače rôznych typov pamätí
- UART/UART Lite
- GPIO
- SPI
- I2C
- Ethernet

Nios II

Nios II bol vyvinutý firmou Altera, ktorá zaberala podobné trhové miesto ako Xilinx a zároveň aj s podobnými licenčnými podmienkami na ich IP core pre procesor Nios II, čo ho robí v tejto práci nepoužiteľným.

Ak by pre túto prácu nebola nevyhnutná úprava zdrojového kódu, Nios II je jedným z popredných soft-core procesorov, hlavne kvôli poskytovaným nástrojom ako je Quartus II, ktorý umožňuje dizajnérom syntetizovať, programovať, ladiť, a dokonca aj pridávať periférie pomocou Avalon Interface Bus. [1]

OpenRISC

OpenRISC je plnohodnotný open-source projekt, ktorý stojí za vývojom OR1200 procesora popísaného vo Verilogu. Aj keď je ešte v praxi používaný, jeho vývoj už nie je aktívny, preto OpenRISC vyvinulo ďalší procesor s názvom mor1kx. Obe varianty majú zdrojové kódy verejne dostupné na ich stránkach. Taktiež majú dostupné aj nástroje na simuláciu, RTOS a mnoho ďalších.

Plasma

Plasma je malý 32-bitový RISC procesor, ktorý je možné implementovať vo VHDL.

Najnovšia verzia Plasma procesora obsahuje obojsmerný sériový port, radič prerušení a hardvérový časovač. Tieto periférie však obsahuje aj väčšina ostatných soft-core procesorov. Vo verzii 3.5 však Plasma pridala radič pre DDR SDRAM, Ethernet a rozhranie pre Flash pamäť. Vyvinutý bol aj pokročily RTOS, ktorý obsahuje väčšinu potrebnej funkcionality. [24]

Zvyčajne sa ale používa iba na akademické účely, okrem tohto je jeho hlavnou nevýhodou aj nedostatočná komunita používateľov, čo znamená, že je voľne dostupných len veľmi malo ďalších materiálov.

RISC-V

RISC-V je piata generácia RISC (Reduced Instruction Set Computer) dizajnu procesora, ktorý bol vyvinutý na Kalifornskej Univerzite v Berkeley. Ide o kvalitný dizajn, ktorý je plne open-source a v poslednom období zažíva rýchly nárast na popularite v priemysle aj na akademickej pôde.

RISC v však nie je kompletný implementovaný soft-core procesor, ktorý by mohol byť ihneď použitý pre túto prácu. Vďaka tomu, že je plne open-source, je možné jeho zdrojové kódy používať na vytvorenie rôznych návrhov. Tento dizajn má vďaka svojej popularite dostatočne veľkú komunitu, vďaka ktorej sú voľne dostupné implementácie mikrokontrolérov s rôznymi vlastnosťami, perifériami a nástrojmi potrebnými na prácu s daným návrhom.

1.4.1 Výber soft-core procesora

Vlastnosti soft-core procesorov v podkapitole 1.3 sú zobrazené v prehľadnej tabuľke č. 1.4.1. Z tabuľky je možné hneď vylúčiť procesory, ktoré nie sú plne open-source a neposkytujú upravovateľné zdrojové kódy, pretože by obmedzovali možnosti tejto práce.

Z ostatných open-source projektov, sú LEON, OpenRISC a Plasma veľmi podobné, keďže všetky ich zdrojové kódy sú dostupné na jednom mieste a ich implementácia je relatívne jednoduchá. Zaostávajú však svojou používateľnosťou oproti RISC-V architektúre, ktorá v posledných rokoch prudko naberá na popularite. Mnoho veľkých firiem sa zapája do výskumu tejto architektúry a návrhu ich vlastných procesorov, ako napríklad Intel.

Porovnanie					
Soft-core	Open-source	Nástroje	Periférie	RTOS	Komunita
Cortex	Nie	Dostupné	Dostupné	Dostupný	**Veľká**
LEON	**Áno**	Dostupné	Dostupné	Dostupný	Stredná
Blaze	Nie	Dostupné	Dostupné	Dostupný	**Veľká**
Nios II	Nie	Dostupné	Dostupné	Dostupný	**Veľká**
OpenRISC	**Áno**	Dostupné	Dostupné	Dostupný	Stredná
Plasma	**Áno**	Dostupné	Dostupné	Dostupný	Malá
RISC-V	**Áno**	Dostupné	Dostupné	Dostupný	**Veľká**

Tab. 1.1: Porovnávacia tabuľka vlastností všetkých vybraných soft-core procesorov.

Práve pre túto popularitu, veľkosť komunity a konkurencieschopnosti IP Core alternatívam je najvhodnejší výber práve architektúru RISC-V, a to aj pre budúcu využiteľnosť tejto práce.

Ako už bolo spomenuté v časti 1.4, RISC-V je len architektúra procesora, a preto je nutné ešte vybrať konkrétne prevedenie soft-core procesora.

1.4.2 Výber soft-core platformy

Pre implementáciu a simuláciu soft-core procesora a testbenchu na FPGA je potrebná platforma, ktorá to umožňuje. Najčastejšie používané platformy na trhu sú Quarthus Prime od firmy Intel a Vivado od firmy Xilinx.

Ako bolo často spomínané v predošlej podkapitole 1.4, väčšina open-source soft-core procesorov mala svoje vlastné nástroje, ktoré ho dokázali implementovať a boli taktiež open-source. Tieto nástroje je možné použiť, ak by túto prácu obmedzovali nejaké licenčné podmienky nasledujúcich platených platform.

Quarthus Prime

Quarthus Prime je dostupný aj v tzv. `LITE EDITION`, ktorá je bezplatná, ale neobsahuje všetky dostupné funkcie. Podporuje základné FPGA dosky a základné nástroje, ako napríklad plánovač čipu na FPGA v jazyku VHDL aj Verilog, simulačné a debugovacie nástroje. [27]

Zo skúseností a pri testovaní platformy Quarthus sa prišlo na problém, ktorý sa v licenčných podmienkach neudáva, a to je obmedzená dĺžka kódu. Tento problém môže podstatne obmedzovať túto prácu, keďže predpokladaný rozsah kódu práce bude pravdepodobne väčší ako dané obmedzenie.

Vivado

Vivado svoju bezplatnú verziu pomenovalo `ML Standard Edition`, ktorá taktiež podporuje len niektoré FPGA dosky, ktoré sa líšia od tých firmy Intel. [28] Vivado ale umožňuje doplniť FPGA dosky od iných výrobcov, s ktorými spolupracuje, ako napríklad Digilent, a to prostredníctvom databáze dostupnej na Git-Hube. Všetky nástroje v platenej verzii sa nachádzajú aj v štandardnej (bezplatnej) verzii, ktoré sa iba mierne líšia, ale nemali by nijako obmedzovať návrh testbenchu. [28]

Projekt obsahuje aj inštaláciu pre bezplatnú FPGA dosku pre Quarthus platformu, kde ale existuje spomínané obmedzenie, preto je vhodnejšie si pre pokračovanie vybrať platformu Vivado.

GHDL

GHDL (G Hardware Design Languge) je open-source analyzér, compilér, simulátor a aj experimentálne synthetyzér, ktorý dokáže spracovať, takmer každý VHDL dizajn. [29]

Oproti ostatným simulačným nástrojom GHDL prekladá VHDL súbory priama do strojového kódu, bez potreby sprostredkujúceho jazyka. Preto by mala byť kompilácia kódu a analýza rýchlejšia.[29]

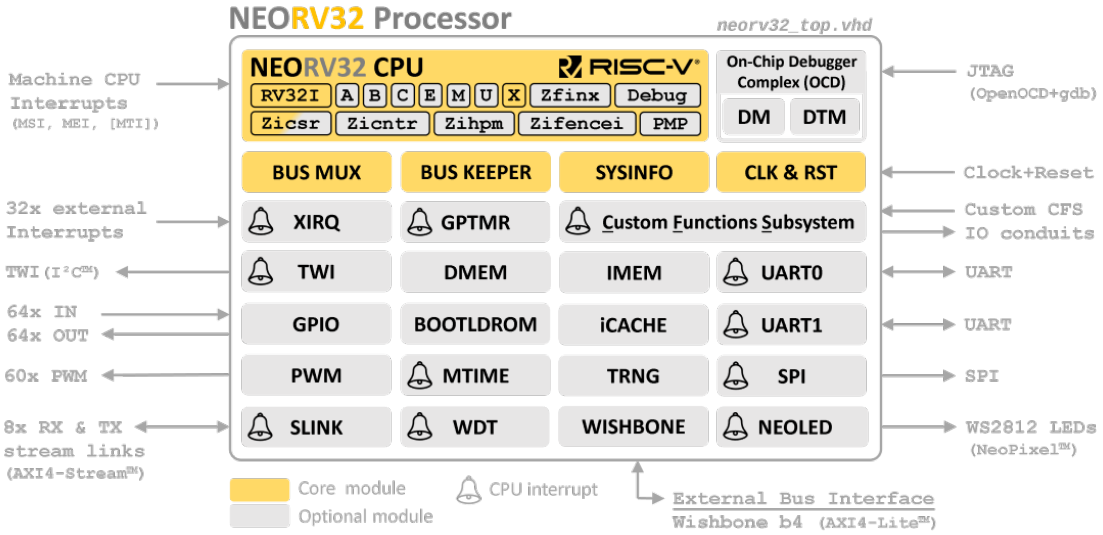
Aktuálna verzia GHDL neobsahuje žiadne vstavané grafické prostredie, ktoré by zobrazovalo priebehy simulácie. Správanie hardvérových dizajnov je stále možné kontrolovať pomocou testbenchov, či frameworkov. Okrem toho GHDL, dokáže vytvárať priebeh signálu, ktoré je si možné zobraziť pomocou externých nástrojov. [29]

GHDL sa hlavne špecializuje na VHDL jazyk a keďže v tejto práci sa bude používať na vytváranie testbenchu jazyk SystemVerilog, ktorý nie je kompatibilný, preto nie je možné v tejto práci používať tento nástroj.

2 NEORV32

NEORV32, ako už bolo viackrát spomenuté, je plne open-source RISC-V kompatibilný soft-core procesor, ktorý je možné hneď používať. Tento soft-core procesor je možné používať samostatne alebo v rámci väčších návrhov SoC kvôli veľkej dostupnosti periférií a nástrojov, ktoré sú plne voliteľné vo výslednom dizajne SoC. Podrobnejšie sú opísané v podkapitole 2.4. [25]

NEORV32 nebol vyvinutý za účelom konkurencieschopnosti ostatným procesorom založených na RISC-V architektúre. Podľa vývojárov sa dôraz kladie hlavne na bezpečnosť vykonávania kódu, najmä na definované a predvídateľné správanie. Vývojári do funkcionality CPU pridali kontrolné opatrenia, ktoré zaisťujú, že všetky prístupy do pamäte musia byť potvrdené a taktiež ak nastane nejaká neočakávaná situácia. [25]



Obr. 2.1: Bloková schéma modulov soft-core procesora NEORV32. [25]

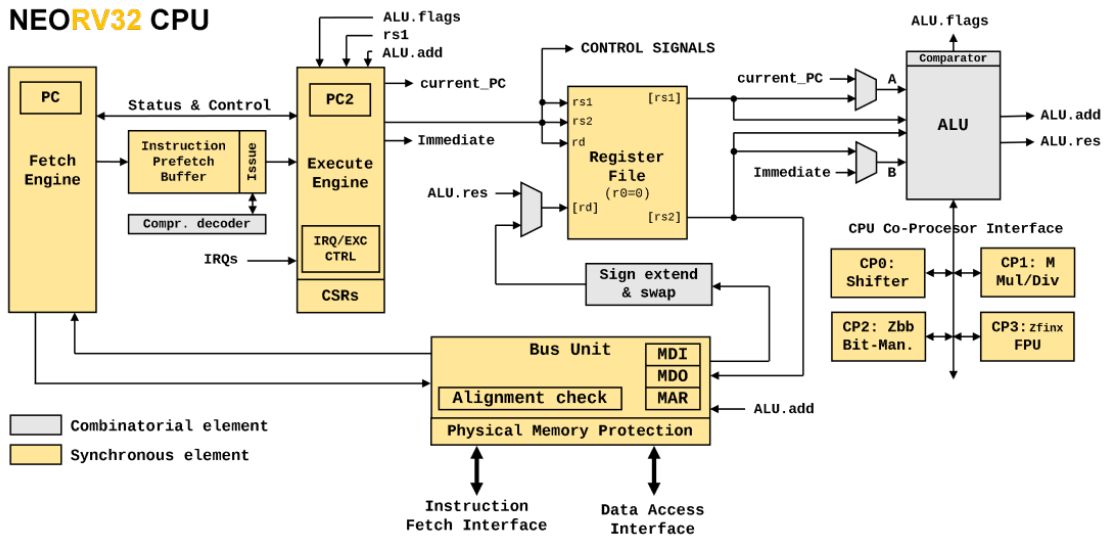
Na obr. č. 2.1 je možné vidieť všetky časti blokovej schémy NEORV32 rozdelené do dvoch základných skupín, CPU a periférie. Bloky v architektúre NEORV32 sú oddelené aj farebne, kde bloky označené žltou farbou sú základné časti, bez ktorých by procesor nemohol fungovať. Ďalšie farebne neoznačené bloky sú voliteľné v soft-core architektúre procesora a je možné ich pridávať alebo odoberať pomocou popisného jazyka na FPGA. [25]

Táto voliteľnosť modulov procesora umožňuje optimálnejší návrh a implementáciu soft-core procesora na FPGA, čo sa týka veľkosti a spotreby energie, keďže

na FPGA sú implementované iba využívané moduly.
V blokovej schéme sú taktiež zaznačené periférie, ktoré majú implementované prerušenia do svojej funkcionality symbolom zvončeka.

2.1 NEORV32 CPU

NEORV32 CPU architektúra bola navrhnutá podľa oficiálnych špecifikácií RISC-V. Na obr. č. 2.2, je možné vidieť zjednodušenú architektúru NEORV32 CPU, z ktorej je možné odvodiť jej funkcionalitu.



Obr. 2.2: Zjednodušená architektúra NEORV32 CPU. [25]

NEORV32 CPU implementuje viaccyklovú architektúru, ktorá sériovo vykonáva mikro-operácie. V zjednodušenej schéme môžeme vidieť, že spracovanie inštrukcií sa vykonáva v dvoch stupňoch, aby sa zvýšil výkon procesora.

Jednotlivé stupne sú oddelené prostredníctvom FIFO vyrovnávacej pamäte (Instruction Prefetch Buffer). To umožňuje načítať nové inštrukcie, zatiaľ čo sa stále vykonáva predošlá inštrukcia. [25]

1. **Instruction fetch** (Načítanie inštrukcie - PC) - slúži na načítanie 32-bitových častí inštrukčných slov (32-bitová inštrukcia, dve 16-bitové inštrukcie).
2. **Instruction execution** (Vykonanie inštrukcie - PC2) - zodpovedá za vykonávanie inštrukcií. Zahŕňa aj *issue engine*, ktorý zostavuje 32-bitové inštrukcie, (prípadne dekomprimuje 16-bitové inštrukcie).

Architektúra NEORV32 nie je úplne totožná s klasickou pipelinovou architektúrou, kde každá časť si vyžaduje presne jeden procesný cyklus, taktiež to nie je klasická multi-cyklová architektúra, ktorá celý proces vykonávanie inštrukcie vykonáva sériovo. Táto architektúra poskytuje kompromis medzi oboma, kde zvyšuje rýchlosť vykonávania inštrukcií, ale zároveň znižuje hardvérovú veľkosť procesora.

NEORV32 CPU obsahuje dve nezávislé zbernice pre načítanie inštrukcií a dát, ako je možné vidieť v dolnej časti na obr. č. 2.2. Tieto zbernice sa následne zlučujú do jednej cez prepínač priority zbernice, kde prístup k dátam má vyššiu prioritu. Celý pamäťový priestor vrátane periférií je potom namapovaný do jedného 32-bitového adresného priestoru. Preto sa vnútorne jedná o Von-Neumannovu architektúru, spomínanú v podkapitole 1.2. [25]

2.1.1 NEORV32 Cache

Procesor poskytuje aj voliteľnú pamäť cache pre inštrukcie na zlepšenie výkonu ak sa používajú pamäte s vysokými odozvami. Cashe je priamo pripojená na inštrukčné rozhranie procesora a poskytuje prístup do celého adresového priestoru. [25]

Inštrukčnú pamäť cache je možné implementovať pomocou premennej `ICACHE_EN`. Veľkosť pamäte cashe je taktiež možné modifikovať podľa potreby premennými `ICACHE_BLOCK_SIZE` a `ICACHE_NUM_BLOCKS`, pričom veľkosť jedného bloku musí byť mocninou 4 a počet blokov musí byť mocninou 2. [25]

Ak má procesor predávané inštrukcie z internej pamäti, kde je oneskorenie iba jeden cyklus cashe pamäť neposkytuje žiadne relevantné zlepšenie vo výkone, ale aspoň mierne zníži prevádzku na internej zbernici procesora. [25]

2.2 NEORV32 ISA rozšírenia

Základná RISC-V (rv32i) architektúra poskytuje niekoľko voliteľných ISA (Instruction Set Architecture) rozšírení, ktoré je možné vidieť na obr. č. 2.3. Tieto rozšírenia zväčšujú inštrukčnú sadu CPU a umožňujú vykonávať sofistikovanejšie operácie, samozrejme je to na úkor hardvérovej zložitosti CPU.

Obr. 2.3: Bloky možných rozšírení inštrukčných sád pre NEORV32 CPU. [25]

ISA rozšírenia je možné jednoducho implementovať nastavením daného príznaku `CPU_EXTENSION_RISCV_<konkrétne ISA rozšírenie>`.

A - Atomic Memory Access umožňuje sofistikovanejšie operácie s pamäťou, ako je implementácia semaforov a mutexov. [25]

B - Bit-Manipulation Operations rozširuje inštrukcie pre bitové operácie. Toto rozšírenie (B) implementuje iba základné bitové, preto je možné pridávať ďalšie podmnožiny sady B, ako *Zbb* a *Zba*. Štandardne sa bitové operácie vykonávajú iteratívne. Pre zvýšenie výpočetnej rýchlosti týchto operácií na úkor hardvérových zdrojov je možné povoliť implementáciu paralelnej logiky (barrel shifter) pre všetky B inštrukcie. [25]

C - Compressed Instructions poskytujú 16-bitové kódovanie často používaných inštrukcií na redukovanie veľkosti kódu. Samozrejme, rýchlosť klesá kvôli nutnosti dekódovať inštrukcie. [25]

E - Embedded CPU redukuje počet všeobecných registrov z 32 na 16, aby sa znížili hardvérové požiadavky, napríklad veľkosť RAM. [25]

I - Base Integer ISA sada, ktorá podporuje základné celočíselné inštrukcie `rv32i`. Táto sada je vždy povolená v NEORV32 CPU. NEORV32 sa v základe snaží udržovať nízke nároky na hardvér, preto sa inštrukcie vykonávajú sériovo až 32 cyklov, záleží na počte posuvov. Pri povolení paralelného spracovania (barrel shifters) je možné dokončiť operáciu v dvoch cykloch bez ohľadu na počet posuvov. [25]

M - Integer Multiplication and Division implementuje hardvérovo akcelerované operácie násobenia a delenia celých čísel. Aj s hardvérovým urýchlením sú tieto operácie vykonávané sériovo alebo je možné implementovať DSP (Digital Signal Procesor).

Zmmul - Integer Multiplication je čiastočné rozšírenie od `M` ISA rozšírenia. Implemetuje iba operácie násobenia z `M` rozšírenia, ktoré si vyžaduje približne iba polovicu hardvéru oproti plnému `M` rozšíreniu. [25]

U - Less-Privileged User Mode k základnému operačnému módu s najvyššími privilégiami, ďalší *užívateľský operačný mód*, ktorý má menej privilégií. Kód, ktorý je vykonávaný v užívateľskom móde, nemá prístup k `CSR` (Control and Status Registers). Prístup v užívateľskom móde môže byť ďalej obmedzený pomocou `PMP` (Physical Memory Protection), ktorá je bližšie popísaná v časti 2.2. [25]

X - NEORV32-Specific (Custom) Extensions je stále povolené, ako je možné vidieť na obr. č. 2.3. Rozšírenie poskytuje 16 rýchlych prerušenía a taktiež je prepojené s `CSR` (Control and Status Registers), ktoré je možné využívať na ľubovoľnú funkcionalitu. [25]

Zfinx - Single-Precision Floating-Point Operations je alternatívou k štandardnému `F` ISA rozšíreniu v RISC-V architektúre, ktoré poskytuje operácie s pohyblivou desatinou čiarkou. `Zfinx` používa súbor celočíselných registrov na ukladanie a prácu s dátami na rozdiel od štandardného `F`, ktoré využíva vyhradený súbor registrov pre prácu s číslami s pohyblivou čiarkou.Rozšírenie `Zfinx` potom vyžaduje menej hardvérových zdrojov a rýchlejšiu manipuláciu s dátami. To taktiež znamená, že nie sú nutné žiadne špeciálne inštrukcie pre presun dát z/do vyhradených registrov. [25]

`Zfinx` rozšírenie podporuje prácu iba s presnosťou jednej desatinnej čiarky, čo je rovnaké ako u štandardného `F` rozšírenia.

Toto rozšírenie nie je zatiaľ oficiálne podporované a nemá zatiaľ žiadnu podporu, ale vnútorná knižnica poskytuje podporu pre využitie `Zfinx` rozšírenia pre jazyk C. [25]

Zicsr - Control and Status Registers Access / Privileged Architecture implementuje inštrukcie pre prístup k `CSR` (Control and Status Registers) a systémy pre výnimky (exceptions) a prerušenia (interrupts).

Bez tohto rozšírenia CPU neposkytuje žiadne privilegované funkcie, ktoré sú potrebné na spustenie zložitejších úloh, ako je operačný systém. [25]

Zicntr - CPU Base Counters rozširuje CPU o základné cykly, počítadlá času, a taktiež príslušné registre. V oficiálnych parametroch pre architektúru RISC-V, je toto rozšírenie určené ako povinné, ale pri aplikáciach, ktoré si obmedzené veľkosťou, je možné toto rozšírenie odstrániť. [25]

Zihpm - Hardware Performance Monitors navyše od základných cyklov a počítadiel NEORV32 v tomto rozšírení poskytuje aj ďalších 29 hardvérových monitorov výkonu, ktoré môžu byť použité napríklad na hodnotenie aplikácie. [25]

Zifencei - Instruction Stream Synchronization umožňuje manuálnu synchronizáciu inštrukčného toku pomocou inštrukcie `fence.i`. Táto inštrukcia resetuje načítanú inštrukciu a vymaže vyrovnávaciu pamäť. To umožňuje čisté znovunačítanie modifikovanej inštrukcie. [25]

PMP - Physical Memory Protection môže byť použitá na obmedzenie práce s pamäťou (čítanie/zapisovanie/vykonávanie) pre všetky dostupné stupne privilégií. Veľkosti blokov, ktoré je možné pomocou PMP, obmedzovať je možné nastavovať, pričom minimálna veľkosť je 8 bajtov. Taktiež je potom možné nastaviť počet týchto blokov, ku ktorým sa vytvoria príslušné registre na ich ovládanie. Väčší počet blokov a ich menšia veľkosť následne zvyšujú hardvérové požiadavky a zároveň sa oneskoruje načítanie inštrukcií alebo dát.

Každý prístup do pamäte je potom testovaný, či nie je chránený PMP pomocou nastavenej adresy a povolený pomocou PMP konfiguračného registra. Ak sa adresa zhoduje s chráneným blokom, operácia s pamäťou sa zamietne a vyvolá sa výnimka pre chybu (fault exception). [25]

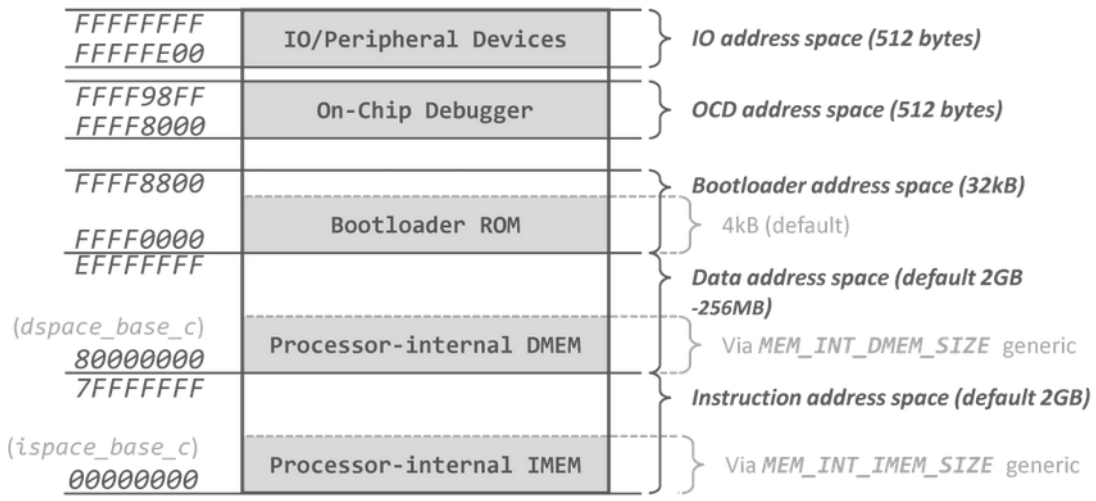
2.3 NEORV32 pamäťový priestor

NEORV32 procesor v základe poskytuje 4GB fyzického adresového priestoru rozdeleného po 32 bitoch, ktorý sa dá samozrejme kedykoľvek rozšíriť. Tento pamäťový priestor je ďalej rozdelený na 5 základných oblastí zobrazených na obr. č. 2.3:

1. **Inštrukčný pamäťový priestor** (Instruction address space) - veľkosť inštrukčného pamäťového priestoru je možné upraviť pomocou premennej `MEM_INT_IMEM_SIZE`. Dôležitá je adresa, na ktorej pamäťový priestor začína `(0x00000000)`, pretože na túto adresu sa nastavuje `program counter` po reštarte procesora.
2. **Dátový adresový priestor** (Data address space) - veľkosť dátového pamäťového priestoru je možné upraviť pomocou premennej `MEM_INT_DMEM_SIZE`.
3. **Adresový priestor pre bootloader** (Bootloader address space) - tento adresový priestor je nemenný, pretože je používaný pre internú pamäť bootloadera.
4. **Adresový priestor pre debugger** (On-Chip Debugger address space) - tento pamäťový priestor taktiež nie je potrebné rozširovať kvôli tomu, že je využívaný a veľkostne optimalizovaný iba pre debbuger.

5. **Adresový priestor pre IO/periférie** (IO/peripheral address space) - rovnako nemenný pamäťový priestor slúžiaci na mapovanie IO a periférií ako napríklad UART.

[25]

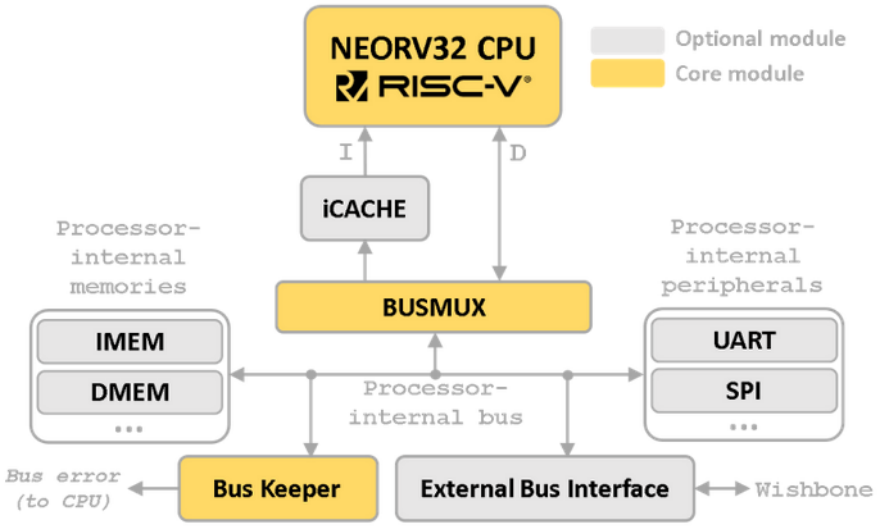


Obr. 2.4: Rozdelenie adresového priestoru NEORV32 procesora. [25]

2.3.1 Dátový a inštrukčný prístup

CPU má prístup k celému 4GB adresovému priestoru, ktoré si môže načítať z rozdeleného rozhrania pre inštrukcie (I) alebo rozhrania pre dáta (D), ako je zobrazené na obr. č. 2.5.

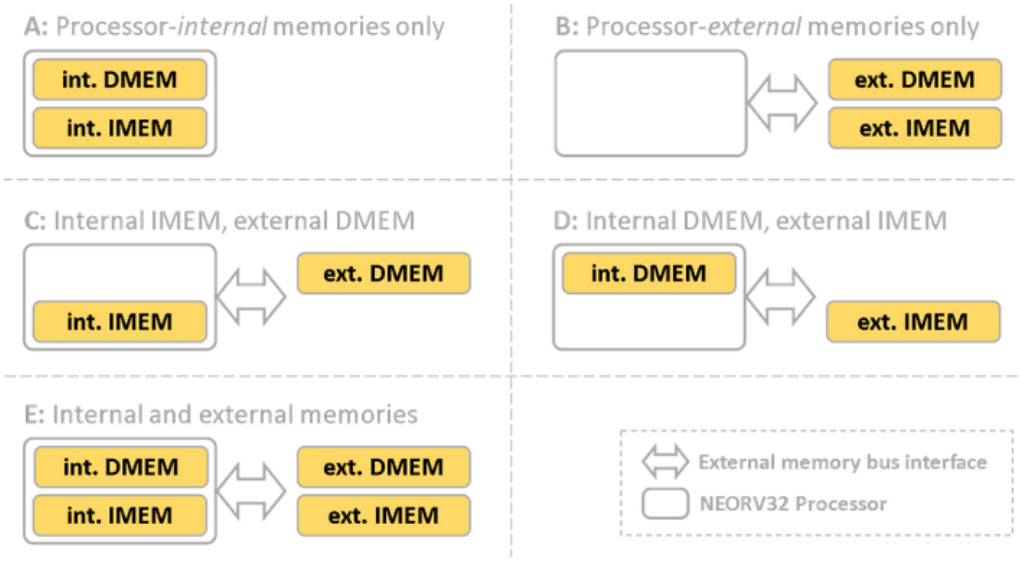
Tieto rozhrania sú však multiplexované prepínačom zbernice (BUSMUX) do jednej internej zbernice procesora. Na túto zbernicu sú pripojené aj všetky interné i externé pamäte a periférie. Rozhrania pre dáta a inštrukcie sú síce rozdelené pre procesor, ale sú prepojené do jednotnej zbernice, ktorá má prístup k celému pamäťovému priestoru, čo znamená, že sa jedná o modifikovanú von-Neumannovu architektúru procesora. [25]



Obr. 2.5: Vnútorná architektúra prístupu do pamäťového priestoru procesora NEORV32. [25]

2.3.2 Konfigurácia pamäte

NEORV32 procesor umožňuje veľkú flexibilitu aj čo sa týka konfigurácie pamäte. Možné je nastaviť nezávislé adresový priestor pre inštrukcie a dáta, či už z internej alebo externej pamäte. Na obr. č. 2.6 je možné vidieť takmer všetky možnosti využívania interného a externého pamäťového priestoru. [25]



Obr. 2.6: Možnosti pripojenia interného a externého pamäťového priestoru pre NEORV32. [25]

V prípade, že by neboli implementované žiadne interné pamäte procesora, tak

sa základný pamäťový priestor automaticky premapuje na externú pamäť. To znamená, že ak napríklad premenná `MEM_INT_IMEM_EN = false`, procesor presmeruje každý prístup na adresový priestor cez externú zbernicu do externého pamäťového priestoru. [25]

Na testovanie NEORV32 procesora v tejto práci bude postačujúca základná konfigurácia, ktorá využíva iba interný pamäťový priestor. Konfigurácia a na obr. č. 2.6.

2.3.3 Boot konfigurácia

Kvôli rôznym spôsobom konfigurácie opísanej v predošlej časti 2.3.2 procesor umožňuje aj rôzne spôsoby bootovania. Na obr. č. 2.7 sú zobrazené dva základné spôsoby bootovania, nepriamy a priamy. [25]

Samozrejme, existuje viac možností, ktoré sú viac sofistikované a kombinujú internú a externú pamäť, ale v rámci tejto práce sa nebudú používať, preto ich nie je nutné ani popisovať.

Nepriame bootovanie (Indirect Boot) - pri tomto nastavení je povolená premenná `INT_BOOTLOADER_EN = true`, ktorá implementuje internú pamäť Bootloader ROM. Táto pamäť umožnuje len čítanie a obsahuje základný firmvér pre bootloader, ktorý je vložený do nej počas syntetizácie. [25]

Tento bootloader môže načítať spustiteľný súbor cez UART alebo z externej flash pamäte pomocou SPI, ktorý nahrá ako binárne inštrukcie na počiatočnú adresu inštrukčnej pamäte, aby ho mohol CPU spustiť. [25]

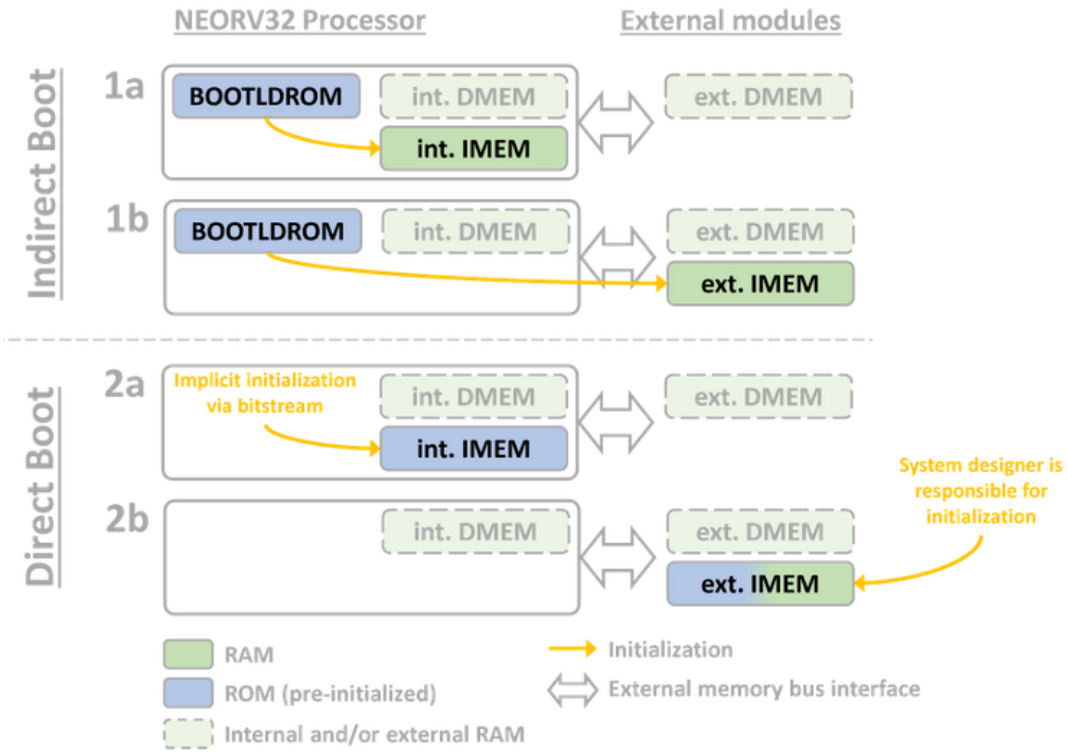
Ako bolo spomenuté v predošlej časti 2.3.2, ak je implementovaná interná pamäť (scenár 1a), bootloader načítava spustiteľný kód z internej pamäte. V prípade, že interná pamäť nie je implementovaná (scenár 1b), bootloader sa automaticky pokúša načítať kód na spustenie z externej pamäte. [25]

Priame bootovanie (Direct Boot) - pri priamom bootovaní sa nepoužíva interný bootloader, čiže premenná `INT_BOOTLOADER_EN = false` a Bootloader ROM nie je implementovaná. CPU začne vykonávať inštrukcie hneď po reštarte od začiatku inštrukčného pamäťového priestoru. [25]

V prípade scenára 2a musí byť interná inštrukčná pamäť povolená premennou `MEM_INT_IMEM_EN = true`. Následne je táto pamäť inicializovaná inštrukciami už pri syntéze. Ak chceme do procesora nahrať iný spustiteľný súbor, je nutné najprv

vytvoriť *.vhd* súbor, ktorý sa vloží do knižnice pred samotnou syntézou.
Tento spôsob je síce najjednoduchší, ale v prípade, že chceme zmeniť len spustiteľný kód, musíme nanovo syntetizovať celý procesor. [25]

V prípade, že interná pamäť je zakázaná `MEM_INT_IMEM_EN = false` (scenár 2b), je nutné, aby externá pamäť bola inicializovaná a obsahovala spustiteľný kód. [25]



Obr. 2.7: Možnosti bootovania NEORV32. [25]

2.4 Interné moduly procesora

Procesor je zložený z CPU 2.1, pamätí 2.3.2, periférií a zbernicovej infraštruktúry, ktorá to celé prepája. V tejto časti práce je popísané, ako sú implementované niektoré dôležité interné moduly, periférie procesora a zbernica *Wishbone* na pripojenie ďalších externých periférií. [25]

2.4.1 Wishbone

Zbernicové rozhranie *Wishbone* je implementované iba v prípade ak je premenná `MEM_EXT_EN = true`. Toto rozhranie môže byť použité na pripojenie externých pamätí, hardvérových akcelerátorov alebo ďalších IO zariadení. [25]

Tieto externé zariadenia nemajú namapované špecifické miesto v pamäti, ako je zobrazené na obr. č. 2.4. Namiesto toho ak adresa nesedí so žiadnymi internými modulmi, prístup je automaticky prenastavený na externé moduly pomocou zbernice Wishbone. Pre upresnenie CPU prístup je presmerovaný na externú zbernicu, ak adresa nesmeruje na internú inštrukčnú pamäť IMEM, internú dátovú pamäť DMEM alebo internú pamäť bootloaderu ROM či iné IO zariadenia. [25]

2.4.2 GPIO

GPIO maximálne poskytuje 64 bitov paralelných vstupov a ďalších 64 bitov paralelných výstupov, ktoré môžu byť použité aj na externé ovládanie (LED, tlačidlá ...) alebo interne na ovládania modulov. GPIO modul je možné v prípade potreby aj neimplementovať s nastavením premennej `IP_GPIO_EN = false`. [25]

Keďže NEORV32, je iba 32 bitový procesor, prístup k GPIO portom je rozdelený na dva 32 bitové registre.

2.4.3 PWM

Kontrolér pre impulzová šírková modulácia (PWM) pre NEORV32 procesor dokáže implementovať do dizajnu až 60 PWM kanálov s nezávislým 8-bitovým počítadlom. Počet PWM kanálov sa dá v rámci dizajnu nastaviť pomocou premennej *IO_PWM_NUM_CH*. Každý kanál má okrem počítadla taktiež nezávislý komparator, ktorý slúži na určenie striedy. [25]

Strieda je následne nastavovaná pomocou DUTY registrov, ktorých môže byť až 15 podľa počtu implementovaných kanálov. Každý register potom obsahuje maximálne 4 PWM kanály na nastavenie striedy. Napr. strieda kanálu 0 sa definuje pomocou bitov 7:0 v registry DUTY[0] a kanál 2 sa definuje pomocou bitov 23:16 v rovnakom registry DUTY[0]. [25]
Intenzitu (striedu) PWM kanálu je potom možné vypočítať pomocou rovnice č. 2.1.

$$Duty = DUTY[y](i * 8 + 7 \; downto \; i * 8)/2^8 \tag{2.1}$$

[25]

Frekvencia je odvodená od frekvencie hodín procesora a delené deličkou cez 3-bitový kontrolný register `PWM_CTRL_PRSCx`. Všetky možné deliace pomery sú v tabuľke č. 2.1. [25]

PWM_CTRL_PRSCx	000	001	010	011	100	101	110	111
Deliaci pomer	2	4	8	64	128	1024	2048	4096

Tab. 2.1: Dostupné deliace pomery pre PWM deličku. [25]

Výsledná frekvencia PWM kanálu sa vypočíta pomocou nasledujúcej rovnice č. 2.2, kde f_{main} je frekvencia hodín procesora a *clock_prescaler* je vybraná deliaci pomer deličky zvolený v PWM kontrolnom registri.

$$f_{PWM} = \frac{f_{main}}{2^8 * clock\_prescaler} \; [Hz] \tag{2.2}$$

[25]

2.4.4 UART

UART je štandardný komunikačný protokol, ktorý sa nachádza v takmer každom procesore a NEORV32 nie je výnimkou. Procesor môže implementovať dva UART periférie pomocou premenných `IO_UART0_EN` a `IO_UART1_EN` .

Tieto periférie majú pevne nastavený počet prenášaných bitov na 8, nastaviteľnú paritu a pevne určený jeden stop bit. Baudrate je nastaviteľný a taktiež je možné si nastaviť aj veľkosti FIFO vyrovnávacej pamäte nezávisle pre vysielač prijímač. [25]

Periéria UART poskytuje aj ďalšie hardérové piny na ovládanie komunikácie, ktoré však nemusia byť využité a je možné ich zakázať. [25]

- **RTS** je možné povoliť nastavením bitu registru `UART_CTRL_RTS_EN`. UART nastaví `uart_rts_o` signál na `log. 0` iba v prípade, že periféria je pripravená na prijímanie nových dát. V prípade, že RTS je zakázané, `uart_rts_o` je stále `log. 0`. RTS pin je automaticky nastavený na `log. 1`, ak bol zaznamenaný nový charanter.
- **CTS** je možné povoliť nastavením bitu registru `UART_CTRL_CTS_EN`. Vysielač nezačne vysielať, pokiaľ signál `uart_cts_i` nie je nastavený na `log. 0`.

2.4.5 Prerušenia

Modul prerušení poskytuje maximálne 32 kanálov, ktorých počet môže byť nastavený pomocou premennej `XIRQ_NUM_CH`. Jednotlivé kanály môžu byť nastavené pomocou `NEORV32_XIRQ.IER` registra. [25]

2.4.6 Custom Function Subsystem

V tomto procesore je možné implementovať vlastnú logiku na špeciálnu aplikáciu. Pre tento modul je vyhradená pamäť o veľkosti 32x32 bitov interných registrov, ku ktorým môže pristupovať CPU pomocou normálnych operácií. Samozrejme, funkcionalita týchto registrov musí byť definovaná pomocou hardvérového dizajnéra. [25]

Výhodou je, že subsystém nemusí byť prepojený pomocou externej periférie alebo zbernice, ale môže k nemu mať prístup priamo procesor, čo zjednodušuje komunikáciu a zrýchľuje odozvu. [25] Rovnako môže tento subsystém, ako aj externé subsystémy pracovať nezávisle, čo znamená, že CPU a subsystém môžu bežať paralelne. [25]

2.4.7 JTAG Debug

Dôležitý je taktiež OCD (On-Chip Debugger (modul), ktorý je kompatibilný s oficiálnou špecifikáciou pre RISC-V. [25] OCD obsahuje:

- JTAG port
- Ovládanie behu CPU (krokovanie ...)
- Prístup k registrom
- Nepriamy prístup k adresovému priestoru
- Breakpoints

Procesor sa dokáže dostať do debugovacieho módu pomocou tzv. halt requestu, nastavením signálu `db_halt_reg_i`. Počas tohto módu je CPU pozastavený a debugger ho môže monitorovať a upravovať. [25]

3 Návrh testbenchu

Úlohou tejto práce je vytvoriť architektúru testbenchu, ktorý by testoval základnú funkcionalitu implementovaného kódu na procesore. Je vhodné zvoliť si štruktúru, ktorú bude do budúcna možné rozširovať. Bližšie popísaná štruktúra testbenchu sa nachádza v podkapitole 3.2.

3.1 Testované moduly

NEORV32 má v sebe zakomponované mnohé interné modly a periférie, ktoré je možné používať a taktiež aj testovať. V rámci tejto práce boli vybrané len základné moduly, aby sa predviedla funkcionalita testbenchu a prípadne sa mohla rozširovať o ďalšie dostupné moduly.

Medzi základné signály, ktoré je nutné pri testovaní ovládať patria *clock* a *reset* signály. Ich funkcionalita je však jasne známa, preto sa táto práca nebude zaoberať ich podrobným popisovaním.

3.1.1 GPIO

GPIO je jedna zo základných periférií v procesoroch, ktorá je taktiež veľmi jednoduchá na testovanie. Vstupné a výstupné piny procesora sú priamo prepojené s testbenchom pomocou rozhrania (*interface*), popísaného v časti 3.2.1. Týmto spôsobom môže vstupné GPIO piny testbench jednoducho nastavovať na požadované hodnoty v požadovaných časových okamihoch, ktoré sú definované testovacím scenárom, bližšie popísanom v kapitole 4.

Výstupné piny sa taktiež pomocou priameho prepojenia môžu čítať, prípadne logovať do súbora alebo porovnáva s referenčnými hodnotami, ktoré vyhodnotia správnosť vykonávania kódu.

3.1.2 PWM

Procesor NEORV32 nemá implementované žiadne analógové piny, čiže ani ADC prevodníky. PWM piny sú vyvedené ako samostatné výstupy, čo zjednodušuje testovanie.

Následne je postačujúce sledovať nástupné hrany na daných PWM pinoch, aby bolo možné určiť periódu modulácie a závernú hranu, ktorá slúži na určenie striedy. PWM parametre sa určia pomocou rovníc 3.1 a 3.2, kde

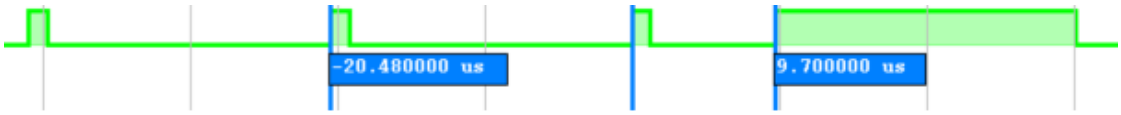
f_{PWM} je frekvencia PWM,
$duty\_cycle$ je strieda PWM,
T_{PWM} je perióda PWM,
$pos\_edge(t)$ je čas aktuálnej nábežnej hrany,
$pos\_edge(t-1)$ je čas predošlej nábežnej hrany,
$neg\_edge(t)$ je čas aktuálnej závernej hrany.

$$f_{PWM} = \frac{1}{T_{PWM}} = \frac{1}{pos\_edge(t) - pos\_edge(t-1)}\ [Hz] \tag{3.1}$$

$$duty\_cycle = \frac{T_{PWM}}{t_{pw}} \cdot 100 = \frac{pos\_edge(t) - pos\_edge(t-1)}{neg\_edge(t) - pos\_edge(t-1)} \cdot 100\ [\%] \tag{3.2}$$

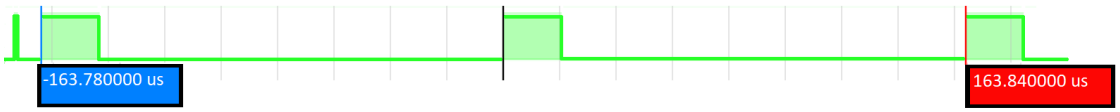
Tieto parametre sa musia merať neustále, ale aby sa každé meranie periódy nemuselo zapisovať do súboru, budú sa ukladať namerané parametre iba pri ich zmene na jednotlivých kanáloch (pinoch).

Pri testovaní PWM bolo potrebné zistiť aj správanie procesora pri zmene PWM parametrov (frekvencia, strieda). Predpokladalo sa, že pri zmene parametrov ich nebude možné správne odmerať. Tento predpoklad sa aj potvrdil, procesor hneď pri zmene parametrov nastaví PWM kanál na `log. 1`, čo spôsobí to, že sa nedokončí celá perióda predošlého PWM signálu a odmerané parametre sú nesprávne. Tento prípad je možne vidieť na obr. č. 3.1.



Obr. 3.1: Správanie výstupného signálu PWM pri zmene jeho parametrov.

Čo sa ale nepredpokladalo je, že aj nasledujúca perióda PWM signálu po zmene parametrov bude mierne chybná, tento prípad je možne vidieť na obr. č. 3.2. Táto odchýlka je však malá a pravdepodobne zanedbateľná voči reálnemu šumu mimo simulácie.



Obr. 3.2: Nekorektné hodnoty výstupného signálu PWM po zmene jeho parametrov.

Nekorektné meranie PWM parametrov spojené s ich zmenou je možné čiastočne vyriešiť „*filtrom ustálenej hodnoty*“. Tento filter porovnáva predošlé a aktuálne PWM parametre a sleduje, či nenastala zmena. V prípade zmeny si zaznamená čas, v ktorom nastala, aby bolo možné ho porovnať s referenčnou hodnotou, a neustále pokračuje v meraní parametrov a v ich porovnávaní. V prípade, že sa znovu predošlé a aktuálne parametre rovnajú, čiže PWM sa ustálila, môžu sa považovať za korektné a priradiť sa k času, kedy nastala zmena.

Takýto filter ale nemôže byť použitý v prípade, ak by sa PWM parametre menili častejšie a taktiež, ak by bol do simulácie zavedený umelý šum. Preto je možné tento filter aj zakázať. V tom prípade bude zaznamenávať každú zmenu v parametroch, ktorá na PWM kanáloch nastane.

3.1.3 UART

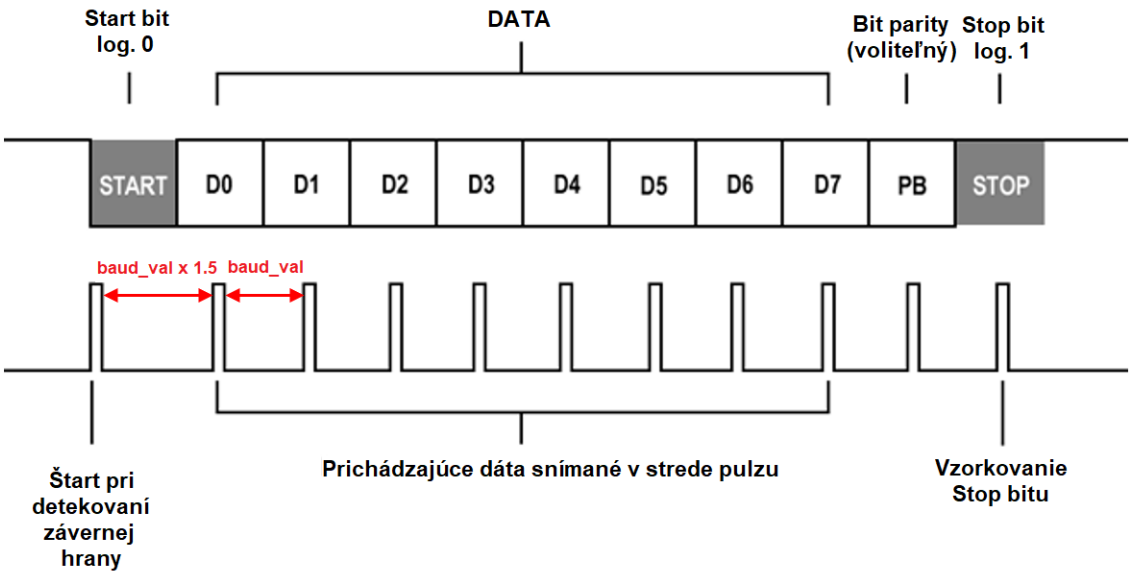
Podobne ako GPIO periféria má UART vstupné a výstupné signály, ktoré sú priamo prepojené s rozhraním testbenchu.

Ako už bolo spomenuté v časti 2.4.4, UART komunikácia má niektoré konfiguračné parametre fixne nastavené, napríklad počet dátových bitov na 8 a jeden stop bit. Taktiež pre účely tejto práce bude dostačujúce, ak testovaná UART komunikácia bude implementovaná bez kontroly parity.

Keďže UART je asynchrónna komunikácia, nie je priamo viazaná na hrany hodinového signálu, preto je nutné vypočítať, kedy a ako čítať či zapisovať na UART zbernicu. To je možné určiť z hodnoty baudrate pomocou rovnice 3.3.

$$baud\_val = \frac{f_{clk}}{baudrate}\ [-] \tag{3.3}$$

Hodnota $baud_{val}$ určuje koľko hodinových signálov sa má vykonať medzi jednotlivými bitmi odosielanými cez UART, ako je zobrazené na obr. č. 3.3. Táto hodnota sa potom rozdielne používa pri čítaní alebo zapisovaní znakov na UART.



Obr. 3.3: UART protokol. [31]

UART-RX

UART-RX testovanie spočíva v tom, že na vstupný pin testbench nastavuje binárne hodnoty a tým simuluje komunikáciu s externým zariadením.

Testbench postupne načítava znaky v žiadanom testovacom čase a zapisuje ich na UART pin s popísaným protokolom na obr. č. 3.3.
Medzi každou zmenou na pine čaká *baud_val* hodinových cyklov, pričom začína START bitom, ktorý nastaví signálový pin na `log. 0`, následne pošle 8 bitov dát a nakoniec sa komunikácia ukončí nastavením signálového pinu na `log. 1`.

UART-TX

Testovanie UART-TX slúži na kontrolu výstupu UART komunikácia z procesora. Testbench sníma výstupný pin a pri spustení komunikácie synchronizačným START signálom postupne zaznamenáva odosielané znaky, ktorých korektnosť môže byť následne analyzovaná.

Aby sa zbernicový UART pin čítal správne, je nutné synchronizovať čítanie dát po START bite tak, aby sa ich hodnota čítala v čase, keď je najmenej pravdepodobná ich ďalšia zmena. Preto sa od START bitu počíta 1,5-násobný počet hodinových impulzov hodnoty *baud_val*, aby sa čítal dátový impulz približne v strede, ako je zobrazené na obr. č. 3.3.

3.1.4 Počítadlo inštrukcií (Program Counter)

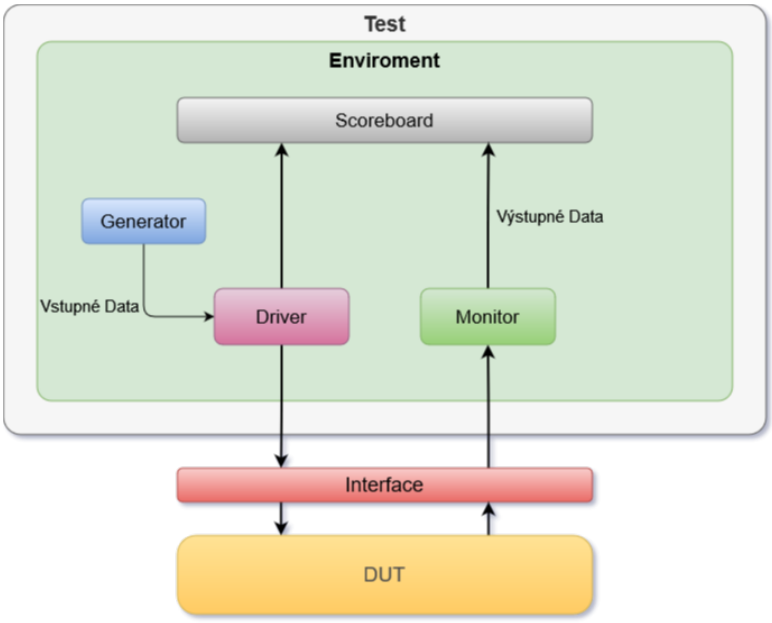
Hodnota PC (Program Counter) ukazuje na aktuálne inštrukciu v kóde. To môže slúžiť napríklad pri zisťovaní, v akom stave sa program nachádza a tak trasovať správanie algoritmu.

PC je ale vnútorná adresa CPU, čiže sa nachádza až v nižších vrstvách hierarchie HDL súborov. Pre jej sledovanie je nutné prepojiť ju, „vytiahnuť“ na vonkajší pin, aby mohol byť prepojený s testbenchom pomocou interface a tak umožnil sledovanie adresy PC.

3.2 Štruktúra testbenchu

Hardvérový dizajn najčastejšie pozostáva z niekoľkých súborov v jazyku Verilog alebo VHDL. V tomto prípade sú súbory NEORV32 procesora napísané vo VHDL jazyku. Dizajn má stále iba jeden top modul, v ktorom všetky ostatné sub-moduly sú vytvorené na to, aby sa dosiahla požadovaná funkcionalita. Tento hardvérový dizajn, ktorý je testovaný v štruktúre testbench, sa označuje ako DUT (Device Under Test) zobrazený na obr. č. 3.4.

Pri návrhu je nutné myslieť na to, aby testbench bol modulárny a rozšíriteľný, preto je daná štandardná štruktúra so všetkými štandardnými verifikačnými komponentami, ktoré to umožňujú. Túto štandardnú štruktúru testbenchu je možné vidieť na obr. č. 3.4.

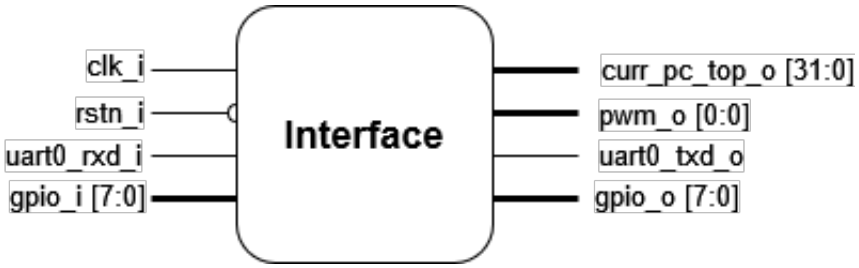


Obr. 3.4: Štandardná štruktúra pre návrh testbenchu. [30]

3.2.1 Interface

Hardvérový dizajn DUT môže obsahovať mnoho signálov, ktoré môže byť obtiažne prepájať a udržovať pri rozširovaní návrhu. Namiesto toho je možné uložiť všetky vstupné a výstupné porty do jedného kontajnera, ktorý sa nazýva *interface* pre DUT. Následne je možné používať tento interface na presúvanie hodnôt v dizajne testbenchu. [30]

V aktuálnom prevedení tenstbenchu interface obsahuje tieto signály ako je to na obr. č. 3.5:



Obr. 3.5: Aktuálny interface testbenchu.

3.2.2 Generator

Generator má za úlohu vytvárať validné dáta, ktoré následne predáva ďalej do driveru. Tieto dáta môže generátor vytvárať rôznymi spôsobmi, často to môžu byť náhodné hodnoty, ktoré prechádzajú rôzne kombinácie vstupov v náhodnom poradí. Taktiež je možné vytvoriť generátor, ktorý bude nastavovať hodnoty podľa určeného scenára. Tieto dáta poskytnuté generátorom môže driver jednoducho riadiť cez interface tak, aby to DUT mohlo spracovať. [30]

V tejto práci bude generátor nahradený vstupnými súbormi 4.2.2, ktoré budú obsahovať požadované vstupné hodnoty, ktoré sa majú nastaviť pomocou interface. Tieto súbory taktiež budú obsahovať časovú značku, ktorá určí, v akom čase sa majú vstupy nastaviť na požadované hodnoty.

3.2.3 Driver

Driver je verifikačný komponent, ktorý nastavuje vstupné porty na DUT pomocou definovanej úlohy v interface. Keď driver má riadiť vstupné premenné v dizajne, musí zavolať vopred definovanú úlohu. Samotný drive nepotrebuje poznať časovú reláciu medzi signálmi, pretože táto informácia je definovaná v úlohe poskytnutej cez interface. Toto je spôsob, akým sa zvyšuje level abstrakcie, ktorý je potrebný na to, aby testbench bol viacej flexibilný a rozšíriteľný.

Ak sa neskôr v návrhu interface zmení, nový driver môže zavolať rovnakú úlohu a tak bez problémov môže riadiť signály novým definovaným spôsobom. [30]

3.2.4 Monitor

V predošlých kapitolách bol vysvetlený spôsob, ako sú dáta privádzané do DUT. Cieľom testbenchu je overiť správnosť dizajnu, preto je potrebné sledovať, aj aké výstupy sa objavia na DUT. To má za úlohu monitor. Tento výsledok spracovaných dát je následne v monitore spracovaný do dátového objektu a posledný ďalej do bloku scoreboard. [30]

V testbenchi môže monitor snímať všetky výstupné piny uvedené v interface (časť č. 3.2.1) a v prípade ich zmeny ju zaznamenať tým, že sa zapíše do výstupného súboru. V prípade UART_TX bude stačiť ak monitor zapíše zmenu do súboru až po prečítaní celého znaku, čiže až si STOP_bite.

3.2.5 Scoreboard

Scoreboard v sebe obsahuje aj referenčný model, ktorý by sa mal správať rovnakým spôsobom ako DUT. Tento model popisuje očakávané správanie DUT. Rovnaké vstupy sú nastavené na referenčnom modele a DUT, takže ak má DUT problém s funkcionalitou, výstup DUT a referenčného modelu budú rozdielne. Tento rozdiel nám napovie, aké funkcionálne chyby obsahuje testovaný dizajn, a to je zvyčajne vykonávané v bloku scoreboard. [30]

V tejto práci nie je úlohou vytvárať testovací scenár s referenčnými hodnotami a vyhodnocovať výsledky testu, ale len získať dáta z procesora počas behu programu s tým, že je možné procesor aj ovládať, preto sa táto časť testbenchu nepoužije.

3.2.6 Enviroment

Práve enviroment robí verifikáciu viac flexibilnou a rozšíriteľnou, pretože rôzne komponenty môžu byť pridané do rovnakého enviromentu pre budúce projekty. [30]

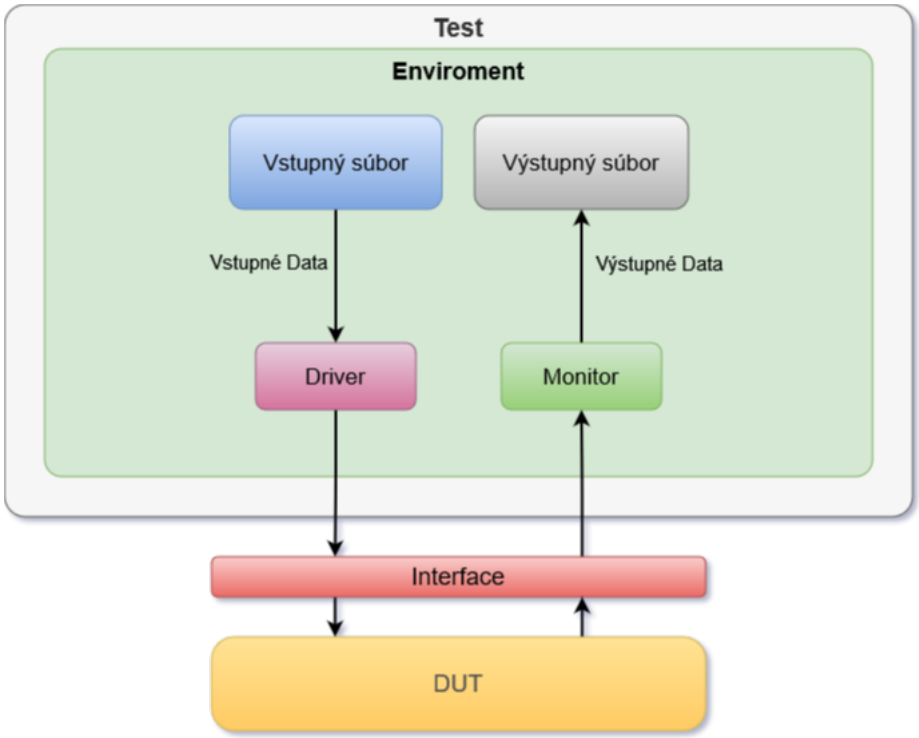
3.2.7 Test

Test je najvyššou inštanciou v testbech. Test vytvára objekt (enviroment) a nastaví spôsob, akým sa test bude vykonávať. Pri verifikácii dizajnu je pravdepodobné, že sa bude vytvárať veľa testov, preto nie je vhodné, aby sa kvôli každému testu

museli vykonať priame zmeny v prostredí (enviroment). Preto je potrebné, aby v prostredí boli určité nastaviteľné parametre, nad ktorými bude mať kontrolu každý test, čo je určite efektívnejšie. [30]

3.2.8 Výsledná štruktúra

Podľa požiadavok na testbench v popise v predošlých častiach, sa štandardná štruktúra upravila do vyhovujúcej formy, prispôsobenej testbenchu. Táto štruktúra je zobrazená na obr. č. 3.6.



Obr. 3.6: Upravená štruktúra pre testbench. [30]

Pri porovnaní obr. č. 3.4 a obr. č. 3.6, je vidieť, že generátor a scoreboard boli nahradené vstupným a výstupným testovacím súborom.

4 Testovací scenár

Táto kapitola sa zaoberá spôsobom, akým sa bude procesor NEORV32 testovať pomocou vytvoreného testbenchu. Celý testovací scenár sa bude odohrávať v prostredí Vivado, aby bolo možné využívať jeho nástroje. Prostredie Vivado môže byť plne ovládané pomocou skriptovacieho jazyku *tcl*, čo zjednoduší užívateľovi prácu a taktiež umožní proces zautomatizovať.

4.1 Vytváranie projektu

Pred testovaním procesora je nutné správne vytvoriť projekt v prostredí Vivado, čo je tiež možné cez *tcl* skript. V rámci materiálov pre procesor NEORV32 bol vytvorený skript na prvotné vyskúšanie procesora. Skript bol upravený tak, aby nastavil projekt takým spôsobom, aby bolo možné rovno procesor testovať s vytvoreným testbenchom.

Skript najprv nastaví správnu dosku, ktorá však nie je potrebná pre simuláciu a testovanie procesora. Ďalej skript vytvára nový priečinok s názvom *project*, v ktorom sa, ako z názvu vyplýva, budú nachádzať všetky dáta z projektu. V prípade, že tento priečinok už existuje, vymaže celý obsah priečinku, aby predošlé súbory nijako nekolidovali s aktuálnym projektom.

Keďže NEORV32 procesor je celý programovaný vo VHDL jazyku, musí sa taktiež nastaviť cieľový jazyk v prostredí na VHDL. Funkcionalitu testbenchu to nijako neovplyvní aj keď je písaná v jazyku SystemVerilog.

Následne sa do projektu pridajú základné VHDL súbory, z ktorých sa skladá NEORV32 procesor ako CPU, pamäte, periférie atď. a nastaví sa správne hlavný (*top*) súbor hierarchie, aby bolo možné procesor syntetizovať. Ak by bolo požadované procesor aj implementovať na reálne FPGA, nastavia sa *constraints*, ktoré prepájajú všetky signály s reálnymi pinami na FPGA.

Do projektu sa ešte nakoniec pridá do zdrojových súborov aj vytvorený testbench a nastaví sa ako hlavný simulačný súbor, aby sa v projekte vedelo ako sa má simulácia vykonávať. Taktiež sa nastaví parameter, ktorý rozbehne simuláciu hneď po jej načítaní na nulovú hodnotu, aby sa simulácia vykonávala presne určený čas a nebola predĺžená o tento vopred nastavený čas. To taktiež sprehľadní konzolu, keďže sa simulácia bude spúšťať len raz od začiatku.

Ako bolo opísané v časti 2.3.2 pre konfiguráciu pamäte, na testovanie sa bude používať základná konfigurácia, ktorá využíva internú pamäť. Konfigurácia A je na obr. č. 2.6. Pri takto nakonfigurovanej pamäti je potrebné vložiť softvér procesora uložený ako zoznam inštrukcií, k ostatným VHDL súborom pred syntézou dizajnu. Preto tento skript syntézu nevykonáva.

Pri testovaní bolo ale zistené, že pred zapnutím simulácie nie je nutné vôbec syntézu spúšťať, pretože simulácia prebiehala stále korektne s novo načítaným softvérom.

Výpis 4.1: Príklad vytvorenia projektu pomocou *tcl* skriptu.

```
tcl> cd .../testbench
tcl> source create_project.tcl
```

Po dobehnutí skriptu na vytvorenie projektu (výpis č. 4.1) je už možné začať s testovaním, ktoré je taktiež spracované pomocou *tcl* skriptu.

4.2 Testovací skript

Pre zjednodušenie testovania bola daná požiadavka na to, aby bolo možné spustiť test pomocou jedného príkazu s príslušnými parametrami cez *tcl* skript.

Celkovo je skriptu možné predať 5 parametrov v tomto poradí:

1. **Softvérový súbor** - presnejšie cestu k `.vhd` súboru s názvom `neorv32_application_image.vhd`, ktorý je preložený z main.c súboru, v ktorom je naprogramované správania procesora. Viac v časti 4.2.1.
2. **Súbor so vstupnými dátami** - súbor, ktorý obsahuje informácie o postupe testu a nastavenia vstupov podľa časových značiek. Viac v časti 4.2.2.
3. **Logovací súbor** - Súbor, ktorý slúži na ukladanie dát výstupov procesora s časovou značkou a adresou PC. Viac v časti 4.2.3.
4. **Čas testovania**
5. **PWM filter** - viac informácií v časti 3.1.2.

Príkaz, pomocou ktorého je možné spustiť skript (`source`), však nedokáže predávať parametre vo formáte `source script.tcl arg1`, preto je nutné predávať konfiguračné parametre pre správne spustenie skriptu nepriamo a to tak, že sa vytvorí konfiguračný súbor s pevne danými parametrami. Cesty k súborom musia byť definované v plnom formáte.

Konfiguračný súbor je základ pre popis testovacieho scenára, ktorý je možné ďalej rozširovať alebo prípadne upraviť jeho štruktúru, ktorú je možné vidieť vo výpise č. 4.2 a príklad konfiguračného súboru vo výpise č. 4.3.

Výpis 4.2: Štruktúra konfiguračného testovacieho súboru.

```
1. <instruction_code_file.vhd>
2. <input_file.csv>
3. <output_file.csv>
4. <time_of_simulation_in_ns_or_specified_time_unit>
5. <PWM_filter_enable(0;1)>
```

Výpis 4.3: Príklad konfiguračného testovacieho súboru.

```
.../neorv32_application_image.vhd
.../input_data.csv
.../output_data.csv
1 ms
0
```

Cesta k tomuto súboru sa zapíše do premennej `argv`, ktorá slúži na predávanie parametrov do príkazov. Následne je ešte potrebné nastaviť aj počet parametrov a to práve 1. S týmito nastavenými parametrami je možné spustiť testovací skript pomocou príkazu `source`. Príklad ako nastaviť parameter s testovacím súborom a spustiť testovací skript, je možné vidieť vo výpise č. 4.4.

Výpis 4.4: Príklad nastavenia parametrov a spustenia testovacieho skriptu.

```
tcl> cd .../testbench

tcl> set argv ./test_files/config.txt
tcl> set argc 1

tcl> source neorv32_test.tcl
```

Popis skriptu

Úlohou testovacieho skriptu je správne nastaviť parametre simulácia a prístupové cesty k požadovaným súborom, následne spustiť simuláciu v prostredí Vivado.

V skripte sa najprv overí počet zadaných parametrov, ktorý by mal obsahovať presne jednu cestu ku konfiguračnému súboru, ako to bolo spomenuté v predošlej časti podkapitoly 4. V prípade, že parameter nebol predaný alebo počet parametrov sa líši a nie je rovný jednému, skript na to upozorní vypísaním chyby na konzolu a predčasne sa ukončí s návratovou hodnotou, ktorá vykazuje počet predaných parametrov *tcl* skriptu.

Následne sa skript pokúsi o otvorenie konfiguračnému súboru, čo je taktiež ošetrené vyvodením chyby a predčasným ukončením skriptu v prípade zlyhania.

Ak súbor existuje a podarí sa ho bez problémov otvoriť, skript pokračuje jednoduchým parsovaním a ukladaním dát do vopred vytvorených premenných, ktoré majú svoje predvolené hodnoty kvôli tomu, že nie všetky parametre sú povinné.

Iba prvé 3 parametre sú nutné na to, aby bolo možné simuláciu spustiť. Výstupný súbor na dáta nemusí ani existovať, keďže je ho možné pred začiatkom simulácia vytvoriť a taktiež je potrebné, aby bol prázdny. Tým sa líši od prvých dvoch súborov (výpis č. 4.2), ktoré musia existovať, pretože obsahujú dáta potrebné pre spustenie simulácie. Pokračovanie skriptu je teda taktiež podmienené ich existenciou.

Ak sa v konfiguračnom súbore nedefinuje čas simulácie (testovania), zvolí sa predvolená možnosť, ktorá spustí príkaz `run all`. Tento príkaz spustí simuláciu, až pokiaľ sa neukončí kód testbenchu, ktorý má ako ukončovaciu podmienku v prípade, že sa prečíta celý súbor so vstupnými dátami. Túto možnosť je nutné si ale zvoliť manuálne, nastavením parametru času simulácie na hodnotu `all` do konfiguračného súboru. Čo sa týka PWM filtra, ten je defaultne vypnutý (3.1.2).

V prípade, ak by sa v konfiguračnom súbore nachádzalo viacej parametrov, ako je maximálny počet, tieto parametre by nenarušili chod testovania, ale skript upozorní na túto skutočnosť v konzole.

Kopírovanie softvérového súboru

Po načítaní parametrov a ich kontrole pokračuje kód skriptu nastavovaním parametrov simulácie. Súbor so softvérom procesora (`<instruction_code_file.vhd>` 4.2) je nutné prekopírovať do zdrojových súborov s cestou `./neorv32/rtl/core/` tak, že sa nahradí predošlý súbor, pričom oba súbory musia mať rovnaký názov `neorv32_application_image.vhd`, aby Vivado dokázalo nový súbor nájsť pri vytváraní simulácie.

V prípade, že cesta v konfiguračnom súbore je prázdna alebo súbory, ktoré sa mali nahradiť, majú rovnakú veľkosť, čiže sa považujú za identické, proces kopírovania sa nevykoná.

Predávanie parametrov testbenchu

Tcl skript v prostredí Vivado umožňuje predávať parametre do simulačného prostredia pomocou príkazu `set_property generic [list_of_parameters] [fileset]`. Tieto parametre sa zapíšu do simulačného prostredia a prenastavujú premenné

v testbenchi s dátovým typom `parameter` a s rovnakým názvom. Problém môže nastať ak by bolo tieto parametre nastavovať samostatne, pretože keď sa použije tento príkaz iba s jedným parametrom, ktorý je potrebné zmeniť, celý list sa vymaže a nastaví sa nanovo iba jeden zadaný parameter. V prípade ak by sa takto chcela simulácia ovládať, musel by sa nastavovať celý list alebo použiť iný spôsob.

Spustenie simulácie

Pri predávaní nových parametrov je nutné celú simuláciu znovu spustiť, aby sa nové parametre mohli prejaviť. V prípade, že simulácia je v nástroji Vivado už spustená, ukončí sa a následne spustí znovu. Na konci skriptu s nastavenými a predanými parametrami sa môže konečne spustiť simulácia s požadovanou dĺžkou času.

4.2.1 Softvér

Vytvorený testbench má za úlohu v prvom rade otestovať správnosť fungovania softvéru, ktorý je aktuálne nahraný v procesore.

Pre procesor NEORV32 je taktiež okrem zdrojových kódov hardvérového popisu procesora dostupná aj knižnica na písanie softvéru v jazyku C. Taktiež sú dostupné aj nejaké ukážkové programy, s ktorými je možné vyskúšať funkcionalitu samotného procesora alebo jeho periférie. Niektoré z týchto ukážkových súborov sú použité na ukážku testovania alebo ich základ je použitý pri vytváraní vlastných testovacích scenárov. Bližšie popísané ukážkové programy pre testovacie scenáre sa nachádzajú nižšie v tejto podkapitole.

Ako už bolo spomenuté v časti 2.3.2, nahrať softvér je možné rôznymi spôsobmi, pričom v tejto práci sa využíva priame nahranie inštrukcií do internej inštrukčnej pamäte pomocou preloženého súboru do VHDL kódu. Spôsob prekladu je podrobnejšie vysvetlený nižšie.

Preklad softvéru

Na preklad softvéru napísanému v jazyku C je potrebný tzv. toolchain, čiže súbor nástrojov, ktorý slúži na vytvorenie spustiteľného kódu. Tieto nástroje boli bližšie popísané v časti 1.3.2.

K prekladu procesora NEORV32 je potrebný špeciálny toolchain pre RISC-V architektúru a to `RISC-V gcc toolchain`. Je možné používať toolchain z tretích strán, ktoré sú schopné preložiť softvére pre RISC-V architektúru, ale taktiež NEORV32 poskytuje vlastný toolchain, ktorý je dostupný spolu s NEORV32 procesorom. Je to

asi tá najkompatibilnejšia voľba a taktiež návody na použitie sú tiež dostupné v dokumentácii NEORV32 procesora. Problém môže byť, že tento toolchain, je možné používať iba na Linuxovej platforme. [26]

V operačnom systéme (OS) Linux, ktorý bol samostatne nainštalovaný, nebol žiaden problém s používaním toolchainu a fungoval bez problémov. Toto riešenie však nie je optimálne, keďže po preklade softvéru je nutné ho otestovať, takže je nutné mať nainštalovaný nástroj Vivado na rovnakom OS. Ďalšia možnosť je používať *Virtual machine* alebo WSL (Windows Subsystem for Linux), v ktorých môže byť zložitejšie toolchain spojazdniť.

Existuje ešte jedna možnosť, ako vytvorený kód simulovať, a to pomocou ďalšieho nástroja GHDL, ktorý je popísaný v časti 1.4.2 [26]. Tento nástroj sa v tejto práci ale nepoužíval, keďže ho nahrádza nástroj Vivado, takže nie je možné popísať jeho funkcionalitu.

Ako už bolo spomenuté spolu s NEORV32 procesorom sú dostupné aj ukážkové príklady, ktoré je možné nájsť v priečinku *.../neorv32/sw/examples/*. Každý ukážkový priečinok obsahuje súbor `main.c`, v ktorom sa nachádza zdrojový kód ukážky a taktiež súbor `makefile`, podľa ktorého sa vytvárajú preložené súbory. Presnejšie sa v súbore `makefile` nachádzajú len ďalšie nastaviteľné cesty k súborom, ktoré sú nevyhnutné pre preklad. Preto je vhodné, ak sa preklad vykonáva v tomto priečinku, aby nenastávali problémy s nesprávnymi cestami.

Tento toolchain neobsahuje všetky nástroje, ako bolo popísané v časti 1.3.2, napríklad IDE. Celý toolchain je ovládaný pomocou príkazov v konzole. Základný príkaz, ktorý spustí preklad C súboru, je `make`.

Parametre prekladu

Následne je možné k príkazu pridávať ďalšie parametre a tak napríklad získať rôzne súbory po preklade alebo ovplyvniť ich funkcionalitu.

Stále je dobré používať parameter `clean_all`, ktorý slúži nato, aby bolo isté, že sa všetko preloží nanovo. [26].

Ako povinný posledný parameter sa môže použiť napríklad `all`

```
$ make clean_all all
```

Tento parameter vytvorí súbory pre všetky typy spustiteľných kódov na nahrávanie do internej pamäte procesora.

- `main.asm` - textový súbor s inštrukciami v assemblery
- `main.bin` - binárný súbor s inštrukciami
- `main.elf` - obohatený binárny súbor s inštrukciami a pozičnými tabuľkami, prípadne symbolmi určenými na debugovanie. [26]
- `neorv32_exe.bin` - spustiteľný súbor, ktorý sa používa pri nahrávaní softvéru pomocou bootloader.
- `neorv32_exe.hex` - textový súbor obsahujúci iba adresy inštrukcií anrozdiel od `main.asm`, ktorý obsahuje aj ich popis.
- `neorv32_application_image.vhd` - vhdl súbor s pevne definovanými adresami internej inštrukčnej pamäte procesora.

V prípade tejto práce kedy je potrebný pre testovanie iba posledný vhdl súbor, stačí použiť parameter `install`.

```
$ make clean_all install
```

Pretože v tejto práci sa bude pracovať s procesorom a testbenchom len v simulácii, je možné upraviť spôsob, akým bude fungovať UART. V prípade, že UART bude pri teste používaný len informačne, je možné ho nastaviť do simulačného módu, čo spôsobí to, že sa priamo presmeruje na vypisovanie do konzoly. To sa dosiahne použitím parametra `USER_FLAGS+=-DUART0_SIM_MODE`. Toolchain upozorní v prípade použitia tohto parametra, aby nebol použitý v prípade, keď je potrebné testovať UART periférium.

```
$ make USER_FLAGS+=-DUART0_SIM_MODE clean_all install
```

Po preložení je možné priamo spustiť aj simuláciu pomocou spomínaného nástroja GHDL pomocou parametra `sim`, ktorý sa ale v tejto práci nepoužíva. V prípade, ak by sa používal, mohol by to byť veľmi rýchly spôsob testovania, pretože takéto použitie môže urýchliť celý proces, čo ale nie je v tejto práci nijako odskúšané.

```
$ make USER_FLAGS+=-DUART0_SIM_MODE clean_all sim
```

Tento toolchain dokáže ďalej prispôsobiť spustiteľný súbor podľa konfigurácie ISA rozšírení a to nastavením parametra MARCH (Machine architecture). MARCH je reťazec znakov, ktoré určujú, aké ISA rozšírenia boli použité v hardvérovej architektúre procesora. Parameter MARCH má určitú štruktúru, kde nie je možné prehadzovať znaky:

```
rv32[i/e][m][a][f][d][g][q][c][b][v][n]...
```

Napríklad `rv32imac` je validné, ale `rv32icma` už nie je. Tento parameter je potrebné nastaviť presne podľa toho, ktoré RISC-V ISA rozšírenia, sú nastavené pomocou premenných `CPU_EXTENSION_RISCV_x`. ISA rozšírenia sú podrobnejšie popísané v podkapitole 2.3. [26]

```
$ make MARCH=rv32imc clean_all install
```

Po preložení programu na spustiteľný súbor je dobre si overiť jeho veľkosť, aby nepresahoval veľkosť internej inštrukčnej pamäte. Táto pamäť sa samozrejme v prípade tejto potreby dá rozšíriť zmenou premennej `MEM_INT_IMEM_SIZE`.

Ukážkové testovacie scenáre

Na ukážkové testovanie a prezentáciu funkcionality tentbenchu bolo vytvorených zopár testovacích scenárov v podobe softvéru. Niektoré z nich boli prevzaté alebo upravené z ukážkových programov pre NEORV32 procesor.

- **Hello world** - tento program slúži viac-menej na základné otestovanie funkcionality procesora. Softvér je preložený s nastaveným parametrom pre simuláciu UART módu `USER_FLAGS+=-DUART0_SIM_MODE`, to znamená, že vypisovanie UART je presmerované na konzolu. Celý program robí len to, že vypíše do konzoly NEORV32 logo a následne hlášku *Hello world :)*.
  Tento program bol prevzatý z ukážkových súborov procesora.

- **Blink LED** - pôvodný program testoval výstupnú periférium GPIO, v ktorej postupne nastavuje výstupný register a to tak, že každú ms pripočíta k hodnote registru + 1. Keďže čas v simulácii beží oveľa pomalšie, pôvodný test nebol veľmi vhodný pre testbench. Pomocou testbenchu je možné výstupné piny otestovať bez nutnej jedno milisekundovej pauzy, preto bol test mierne upravený, čas testu skrátený tak, aby sa vyskúšali všetky kombinácie výstupných GPIO pinov. UART v tomto prípade je tiež nastavený v simulačnom móde, pretože iba informuje o type testu na konzole.

- **PWM test** - slúži na otestovanie ďalšej implementovanej periférie, ktorú testbench dokáže merať. Pôvodný program nastavoval postupne striedu na 4 kanáloch PWM, pričom mal statickú frekvenciu nastavenú preddeličkou. Program bol potom upravený tak, aby bolo možné otestovať rôzne frekvencie a taktiež rôzne nastavenia striedy. Testbench má na ukážku na výstupe iba jeden PWM kanál, preto je program okresaný. Aby testovanie netrvalo príliš dlho, vybrali sa iba 4 frekvencie a hodnoty striedy.

- **GPIO loopback** - tento program narozdiel od *Blink LED* dokáže otestovať zároveň vstupné a výstupné piny GPIO periférie. Program veľmi jednoducho zapíše na výstup totožné hodnoty, aké prečíta na vstupe. Pri testovaní sa zistilo, že na vstup GPIO periférie je možné zapisovať až po nejakej dobe, pokiaľ sa program nedostane do nekonečnej slučky, v ktorej číta GPIO port. Keďže ani v tomto prípade nie je UART potrebný pri testovaní, je preložený v simulačnom móde.

- **UART loopback** - používa znovu podobný princíp ako *GPIO loopback*, kde postupne číta vstupnú zbernicu pre UART a znaky, ktoré prečíta znovu, prepošle na výstupnú zbernicu. Takto sa najjednoduchšie otestuje hlavne korektnosť vytvoreného testbenchu či zapisovanie a čítanie zberníc funguje, ako má. Pri tomto teste je potrebné dávať pozor na to, že procesor je schopný čítať dáta posielané na vstupnú zbernicu UART až po určitej dobe. Ak budú dáta posielané príliš skoro, dáta nebudú prečítané korektne. Taktiež každý znak potrebuje čas na odoslanie, tento čas záleží od hodnoty baudratu.
  Keďže ide o test striktne zameraný na testovanie UART periférie, je nutné ho preložiť bez parametru pre simulačný mód.

- **Kombinovaný test** - spája niektoré predošlé testy dokopy, aby sa odskúšalo, ako funguje testovanie všetkých implementovaných periférií v testbenchi. Taktiež sa tak kontroluje prehľadnosť vypisovania na konzolu a do výstupného logovacieho súboru.
  Program sa skladá z GPIO loopbacku, ktorý zároveň nastavuje hodnotu striedy PWM kanálu podľa vstupnej hodnoty prvých 8 bitov nastaveného GPIO portu. Hodnota frekvencie nastavená pomocou preddeličky je nemenná.
  Ďalej je v teste implementovaný aj UART loopback, aby sa otestovala aj táto periféria. Kvôli tomu sa zase nesmie UART prekladať iba v simulačnom móde.

Tieto testovacie programy slúžia iba ako ukážka funkcionality testbenchu a predvedenie zatiaľ dostupných periférií na testovanie. Testbench má do budúcna za úlohu tento softvér otestovať a vyhodnotiť správnosť jeho chovania.

4.2.2 Načítavanie vstupných dát

Dáta, ktoré sa budú zapisovať na vstupné piny rozhrania NEORV32 procesora, sú načítané zo samostatného súboru. Tieto dáta sa postupne čítajú zo súboru po riadkoch, ktoré majú časovú značku, aby testbench vedel, v akom čase nastaviť vstupné rozhranie na určené hodnoty.

Formát vstupného súboru

Pre vstupné aj výstupné dáta bol zvolený typ súboru s formátom `.csv` s oddeľovaním jednotlivých hodnôt znakom „;“. Ako môže vyzerať ukážkový vstupný súbor, je možné vidieť vo výpise č. 4.5.

Výpis 4.5: Ukážka formátu vstupného súboru (umelo zarovnané).

```
Timestamp [ns]  ;rstn_i ;gpio_i     ;uart0_rxd_i
100             ;1      ;00000000   ;
50100           ;       ;11111111   ;
80100           ;       ;           ;test
2200100         ;       ;0          ;off
3790100         ;0      ;           ;
```

Súbor obsahuje hlavičku výlučne iba na prvom riadku. Táto hlavička obsahuje štruktúru zapisovania vstupných parametrov pre test a taktiež môže obsahovať komentár ku konkrétnemu testu, napríklad čas kedy je už program pripravený alebo minimálny čas na odoslanie jedného znaku po zbernici UART. Tieto hodnoty sú tiež zohľadnené v ukážkovom súbore vo výpise č. 4.5.

Časovú značku je nutné pri tvorení testu nastavovať manuálne, aby test prebehol správne. V prípade, že by časové značky boli nastavené tak, že sa nestíhali odoslať všetky znaky na cez zbernicu UART, testbench dokončí odosielanie a následne upozorní na oneskorenie testu s konkrétnou hodnotou meškania aj do výstupného logovacieho súboru, ale taktiež priamo na konzolu.

Vstupný signál `rstn_i` je vždy potrebné na začiatku testu nastaviť na `log. 1`, aby nastalo spustenie procesora a začal vykonávať nahraný program z internej inštrukčnej pamäte. Taktiež je možné procesor týmto signálom vypnúť alebo reštartovať, samozrejme sa musí brať ohľad na to, aby sa všetky operácie na procesore dokončili, ako napríklad čítanie UART zbernice.

GPIO vstupný port je nastavovaný ďalším parametrov `gpio_i`, ktorý je nutné písať v binárnom formáte, pretože takto ho načítava testbench. V prípade, že nie sú zapísané všetky bity, nastavujú sa primárne iba bity od LSB (Least Significant Bit).

Poslednou hodnotou v jednou riadku tabuľky je reťazec znakov, ktorý sa má odoslať na zbernicu UART. Tento reťazec nie je ukončený znakom „\n“, ktorý zároveň

ukončuje celý riadok tabuľky. Nasledujúci riadok s časovou značkou musí teda zohľadniť to, aby bolo možné všetky znaky v reťazci odoslať. Táto hodnota nasledujúcej časovej značky sa vypočíta jednoducho podľa rovnice 4.2, kde t_{next} je nasledujúca časová značka, t_{prev} je predošlá časová značka, N_{char} je počet znakov v predošlom reťazci a $t_{transmit\_char}$ je čas posielania jedného znaku, ktorý je možné vypočítať podľa rovnice 4.1. Tento vzťah platí iba v prípade, že nevyužívame bit parity.

$$t_{transmit\_char} = (1_{START} + 8_{DATA} + 1_{STOP}) \cdot \frac{1}{baud\_rate} \ [ns] \tag{4.1}$$

Pri hodnote baudrate 19200 sa vypočíta $t_{transmit\_char}$ takto, pričom výsledná hodnota sa zaokrúhli nahor kvôli rezerve.

$$t_{transmit\_char} = 10bits \cdot \frac{1}{19200} = 520833 \approx 530000 \ ns$$

$$t_{next} = t_{prev} + N_{char} \cdot t_{transmit\_char} \ [ns] \tag{4.2}$$

Ako je možné vidieť vo výpise č. 4.5, nie je nutné zadávať všetky hodnoty do vstupného súboru, čo znamená, že sa vstupné hodnoty nebudú nijako meniť.

4.2.3 Zapisovanie do súboru

Zapisovanie do výstupného súboru je formátované podobne ako pri vstupnom súbore v `.csv` formáte iba pre výstupy. Hlavičku výstupného súboru je možné vidieť vo výpise č. 4.6.

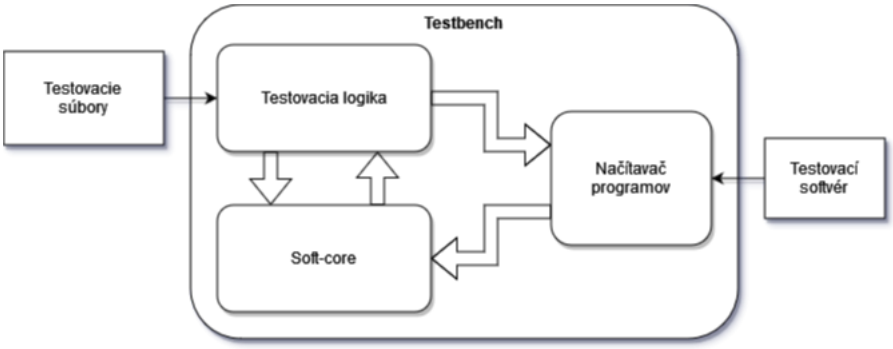
Výpis 4.6: Hlavička výstupného súboru (prvý riadok).

```
Timestamp[ns];
curr_pc_o[0x];
gpio_o[0b];
PWM_freq[Hz];PWM_duty[x/255];
uart0_txd_o
```

Do výstupného súboru sa zapisuje iba pri zmene nejakého parametra, pričom sa k nemu stále pripisuje časová značka a zároveň aktuálna pozícia PC (Program Counter), ktorá ukazuje na adresu vykonávanej inštrukcie kvôli trasovaniu programu, bližšie opísaná v časti 3.1.4.

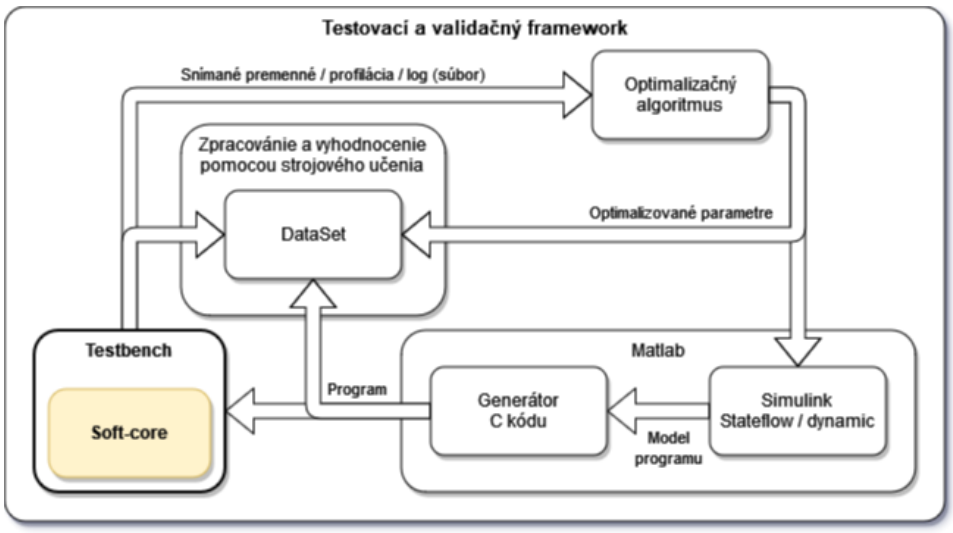
5 Vyhodnotenie vlastností testbenchu

Podarilo sa vytvoriť testbench pre konkrétny typ soft-core procesora NEORV32 architektúry RISC-V. Testbench dokáže pomocou príkazov v konzole načítať softvér do procesora a spustiť simuláciu, počas ktorej umožňuje manipuláciu vstupných signálov procesora postupne načítaných zo súboru pre konkrétny testovací scenár. Všetky zmeny na výstupných signáloch následne zapíše do výstupného logovacieho súboru. Popis výsledného testbenchu je možné vidieť na obr. č. 5.1.



Obr. 5.1: Popisná štruktúra vytvoreného testbenchu.

Tento testbench má do budúcna predstavovať len časť celkového testovacieho a overovacieho frameworku, ktorý je možné vidieť na obr. č. 5.2



Obr. 5.2: Testovací a overovací framework.

S týmto úmyslom bol navrhovaný celý testbench, aby jeho vlastnosti bolo možné využiť v rámci väčšieho celku. Najlepšie bude tieto vlastnosti popísať pri celkovom priebehu testovania.

5.1 Priebeh testovania

Pred samotným testovaním, ako je známe z predošlých kapitol, je potrebné navrhnúť test. Keďže NEORV32 soft-core procesor je veľmi konfigurovateľný, návrh testu pre softvér začína už pri úprave parametrov procesora, aby dokázal správne a prípadne čo neafektívnejšie vykonávať testovaný program.

5.1.1 Nastavenie parametrov procesora

Parametre procesora, napríklad ISA rozšírenia, veľkosti pamätí či povolenie určitých periférií je nutné nastaviť manuálne v rámci testbenchu, ako je vyobrazené vo výpise kódu č. 5.1. Tieto parametre sa automatický prenesú do hlavného (top) `neorv32_top.vhd` súboru hierarchie a procesor bude syntetizovaný práve s týmito parametrami.

Výpis 5.1: Časť kódu testbenchu s nastaviteľnými parametrami procesora.

```
parameter CPU_EXTENSION_RISCV_A          = 1'b1;
parameter CPU_EXTENSION_RISCV_B          = 1'b1;
parameter CPU_EXTENSION_RISCV_C          = 1'b1;
parameter CPU_EXTENSION_RISCV_E          = 1'b0;
parameter CPU_EXTENSION_RISCV_M          = 1'b1;
parameter CPU_EXTENSION_RISCV_U          = 1'b1;
parameter CPU_EXTENSION_RISCV_Zfinx      = 1'b1;
...
parameter EXT_IMEM_C                     = 1'b0;
parameter MEM_INT_IMEM_SIZE              = 16*1024;
parameter baud0_rate_c                   = 19200;
...
```

Keďže štruktúra testbenchu nie je plne implementovaná do stavu v podkapitole č. 3.4, nie je možné mať definované parametre procesora samostatne pre každý test, ale je nutné upravovať samotný testbench. S doterajším datasetom testovacích scenárov nie je potrebné parametre procesora nijako upravovať, preto nebolo potrebné štruktúru testbenchu doteraz plne implementovať.

Tento problém by bolo možné vyriešiť aj iným spôsobom. Parametre, ktoré by bolo potrebné upraviť by sa zapísali do konfiguračného súboru a následne spracovali testovacím skriptom, ktorý by ich predal testbenchu.

5.1.2 Nastavenie konfiguračného súboru

Pri nastavovaní parametrov je nutné dodržiavať formát konfiguračného súboru.

- Poradie parametrov sa nesmie zamieňať.
- Cesty k testovacím súborom musia byť zadávané manuálne a v plnom formáte.
- Názov softvérového súboru musí byť `neorv32_application_image.vhd`.
- Parameter softvérového súboru môže byť vynechaný a nahradený prázdnym riadkom v prípade, že ostáva nezmenený.
- Čas simulácie je defaultne v jednotkách *ns*, pričom je možné použiť aj jednotky (*µs, ms, s*) alebo použiť kľúčové slovo *all*.
- Parameter PWM filtra je možné vynechať, pričom jeho defaultná hodnota je `log. 0` (vypnuté filtrovanie).

V prípade zlé nastavených parametrov alebo nesplneného formátu test zlyhá zvyčajne už pri kontrole skriptom a následne upozorní užívateľa na chybne zadané parametre.

Výpis 5.2: Konfiguračný súbor pre kombinovaný test popísaný v časti 4.2.1.

```
.../testing_scenario/neorv32_application_image.vhd
.../testing_scenario/input_data.csv
.../testing_scenario/output_data.csv
all
0
```

Reálne nastavené parametre pre konfiguračný súbor je možné vidieť vo výpise č. 5.2, konkrétne pre kombinovaný test. Všetky potrebné súbory sa nachádzajú v jednom priečinku, pričom čas simulácie, je nastavený na `all`. To znamená, že simulácie sa ukončí hneď ako nebudú dostupné žiadne dáta vo vstupnom súbore, ktorého obsah je možné vidieť v tab. č. 5.1. PWM filter je pri tomto teste vypnutý.

5.1.3 Nastavenie vstupného testovacieho súboru

Ako už bolo spomenuté v predošlej časti 4.2.2, testovacie súbory sú vo formáte `.csv`. Na vytváranie vstupného testovacieho súboru je vhodné používať tabuľkové editory ako je ukázané v tab. č. 5.1. a to z toho dôvodu, že testbench nie je plne ošetrený proti nekorektnému formátu v súbore a môže spôsobiť zacyklenie simulácie.

Timestamp [ns]	rstn_i	gpio_i	uart0_rxd_i
100	1	00000000	
80000		01000000	T
6.1e4		10000000	e
1140000		11000000	s
1670000		11111111	t
2200000		0	
2730000	0		

Tab. 5.1: Vstupný súbor pre kombinovaný test popísaný v časti 4.2.1.

Pri zadávaní hodnôt do súboru nie je nutné všetky hodnoty vypĺňať v prípade, že sa nemenia, výnimkou je samozrejme časová značka, ktorá musí byť stále prítomná. Časovú značku je v prípade veľkých hodnôt možné vypĺňať aj vo vedeckom formáte čo je tiež ukázané v tab. č. 5.1

Pri navrhovaní časových značiek bolo potrebné oneskorenie cca 80000 ns, aby sa procesor dostal do stavu vykonávania nekonečnej slučky, v ktorej sa vykonáva hlavný testovaný program. Časový rozostup medzi odosielaním znakov cez UART bol taktiež dodržaný o hodnote 530000 ns, ktorá bola vypočítaná pomocou rovníc 4.1 a 4.2. Bolo potrebné tiež počítať s tým, že sa odoslaný znak po prečítaní procesorom, bude rovnaký čas odosielať naspať. Test sa preto ukončí s dvojnásobným oneskorením a tak testbench bude schopný prečítať posledný odosielaný znak.

5.1.4 Spúšťanie testov

Testovanie je možné spustiť pomocou nastavenia jedného parametra a to cesty ku konfiguračnému súboru a následne zadaním jedného príkazu. V prostredí Vivado sa spustí simulácia, ktorá sa po dokončení testu automatický zastaví a výsledky testu sú zapísané vo výstupnom súbore. Takýmto spôsobom je možné otestovať iba jeden test a neexistuje spôsob, akým by bolo možné spustiť viacej testov naraz.

Pred každým takýmto testom je nutné pri zmene akéhokoľvek parametra v konfiguračnom súbore znovu celú simuláciu spustiť, čím sa nahrajú aj nové parametre, čo trvá rádovo desiatky sekúnd. V prípade, že by sa implementoval spôsob, akým by bolo možné spustiť viacero testov po sebe, táto vlastnosť testbenchu by celé testovanie spomaľovala.

V tejto práci sa nepodarilo nájsť iný spôsob, akým predávať parametre, aby sa tento problém vyriešil alebo aby sa aspoň skrátila doba spustenia simulácie. Riešením môže byť aj použitie iného nástroja, napríklad spomínaného GHDL v časti 1.4.2. S týmto nástrojom sa však nepracovalo kvôli nekompatibilite s HDL jazykom SystemVerilog, v ktorom je testbench napísaný.

Po spustení a ukončení testu sú výsledky zapísané do súboru, ktorého tvar je možné vidieť v tabuľke č. 5.2, konkrétne pre kombinovaný test.

Timestamp[ns]	curr_pc_o	gpio_o	PWM_f	PWM_duty	uart_txd_o
41350	000000ac	00000000			
99170	00000cb8	01000000			
100150	000002a8		9985	16	
262250	00000cc4		6169	62	
426090	00000cc8		6104	64	
610920	00000cb8	10000000			
753770	000004f8		6104	128	
1044850	00000cc8				T
1140650	00000cc8	11000000			
1245290	00000cc8		6104	192	
1574580	00000cc8				e
1670380	00000cb8	11111111			
1736810	000004fc		6104	255	
2105150	00000cc8				s
2200950	00000cb8	00000000			
2634880	00000cc8				t

Tab. 5.2: Výstupný súbor pre kombinovaný test popísaný v časti 4.2.1.

Výsledku testu je možné si porovnať so vstupnými hodnotami v tab. č. 5.1, pričom nahraný softvér je popísaný v časti 4.2.1. V tabuľke je možné vidieť, ako sa prejaví meranie PWM parametra ak je PWM filter vypnutý, ako je nastavené v konfiguračnom súbore vo výpise č. 5.2.

5.2 Možnosti rozšírenia

Ako bolo spomínané na začiatku tejto kapitoly, ide o časť väčšieho budúceho frameworku, takže rozšírení, ktoré by mohli byť ďalej spravené, je mnoho. Táto podkapitola sa však bude zameriavať hlavne na rozšírenia pre testbench, ktorý bol vytvorený.

Doplnenie modulov a periférií

Základné moduly a periférie, ktoré momentálne testbench pokrýva, je možné rozširovať pridávaním ďalších do štruktúry testbenchu. Napríklad SPI, TWI, XRIQ, JTAG alebo ďalšie moduly a periférie, ktoré procesor NEORV32 poskytuje, zobrazené na obr. č. 2.1.

Samozrejme je nutné vymyslieť a implementovať spôsob testovania každého nového modulu a periférie, rozšíriť interface a upraviť štruktúru testovacích súborov.

Dokončenie štruktúry testbenchu

Pred doplnením modulov a periférií spomenutých v predošlej časti 5.2 je vhodné upraviť štruktúru testbenchu do podoby, ktorá je popísaná v časti 3.2, a zobrazená na obr. č. 3.6. a to z toho dôvodu, aby sa ďalšie doplnené moduly nemuseli znovu upravovať kvôli zmene štruktúry.

Sofistikovanejšia konfigurácia testu

Ako je spomenuté v častí 5.1.2, konfiguračný súbor má prísny formát. Tento súbor by bolo možné upraviť do formátu XML alebo JSON, ktoré umožňujú lepšiu rozšíriteľnosť konfigurácie. Jednoduchšie by sa pomocou neho nastavovali premenné procesora, či simulácie. Dáta z takto formátovaného súboru by bolo potrebné správne načítať, čo by si vyžadovalo implementáciu parsera.

Nahrávanie softvéru pomocou bootloadera

V teoretickej rešerši soft-core procesora NEORV32 v časti 2.3.2 sú popísané viaceré spôsoby nahrávania softvéru do internej inštrukčnej pamäte. Ďalšou základnou z nich, ktorá sa často používa v praxi, je nahrávanie pomocou bootloadera. V tomto prípade sa môže jednať o pomalejší proces v prípade, ak ide len o nahrávanie do internej pamäte, ale tento spôsob môže byť súčasťou testovacieho scenára, ktorý je vhodné otestovať.

Implementácia RTOS

V prípadoch zložitejšieho softvéru, kde by bolo potrebné spracovávanie dát v reálnom čase a manažovať systémové zdroje, bude potrebné v rámci softvéru používať RTOS. Preto pre ďalšie rozšírenie by bolo vhodné pridať do testbenchu funkcionalitu, ktorá by dokázala spracovávať dáta, ktoré RTOS poskytuje v rámci svojho operovania, ako napríklad profilácia programu alebo zaznamenávanie času strávených v jednotlivých úlohách (task).

Záver

Diplomová práca sa zaoberá návrhom testbenchu na vybraný soft-core procesor NEORV32 architektúry RISC-V pre simulácie embedded aplikácií v prostredí FPGA. Testbench bol vytvorený s cieľom jeho rozšírenia na testovací a validačný framework.

Prvá kapitola sa zo začiatku venuje porovnávaniu hard a soft prevedení procesorov, kvôli zoznámeniu sa s ich spoločnými a rozdielnymi vlastnosťami, ktoré sú dôležité pre výber vhodného soft-core procesoru a návrh testbenchu. V práci je následne popísaná architektúra procesorov, aby bolo možné správne pochopiť princíp testovania obvodu. Taktiež sú v kapitole vysvetlené základy hardvérových popisných jazykov, pomocou ktorých bol testbench navrhnutý.

Na základe porovnania dostupných soft-core procesorov bol vybraný open-source procesor NEORV32 s architektúrou RISC-V kvôli dostupnosti platforiem a nástrojov. Ďalšou jeho silnou stránkou je bezpečnosť vykonávania kódu a predvídateľné správanie. Model procesora bol následne implementovaný pomocou platformy Vivado, kde bol otestovaný pomocou dostupných materiálov k procesoru.

V praktickej časti bola pre tento procesor navrhnutá architektúra testbenchu so štandardizovanou štruktúrou. Boli vybrané a otestované základné moduly (GPIO, PWM, UART a PC). Pre tieto moduly bolo navrhnutých niekoľko testovacích scenárov, ktoré testujú moduly jednotlivo. Bol navrhnutý aj scenár pre otestovanie celkovej funkcionality testbenchu, v ktorom boli skombinované jednotlivé testy. Tieto scenáre boli implementované a je ich možné spúšťať pomocou vytvoreného *tcl* skriptu špeciálne vytvoreného pre nástroj Vivado. Pre korektné nastavenie dizajnu NEORV32 procesora testbenchu v rámci nástroja Vivado bol vytvorený pomocný skript, ktorý automaticky založí projekt so správnym hierarchickým nastavením, aby ho tester mohol ihneď používať.

Posledná kapitola sa zaoberá vyhodnotením vlastností testbenchu. Tie sú popísané pri postupnom priebehu návrhu testovacieho scenára a samotnom testovaní. Pri navrhovaní testovacieho scenára je nutné dodržiavať striktný formát konfiguračných a testovacích súborov. Simuláciu v testbenchi je v prostredí Vivado nutné pri každom teste nanovo spustiť, čo môže viesť k spomaleniu testovania v prípade veľkého množstva testov. Tento problém nie je možné vyriešiť v prípade, že je do procesora nahrávaný nový softvér. Výstupom testovania je logovací súbor s formátom `.csv`, zobrazený v tab. č. 5.2.

Kapitola je ukončená popísaním možností rozšírení funkcionality navrhnutého testbenchu. Ten môže byť doplnený o ďalšie moduly a periférie, ktoré NEORV32 poskytuje, napríklad SPI, prerušenia alebo JTAG. Konfiguračný súbor má príliš striktný formát a nie je možné pomocou neho konfigurovať parametre. Preto by bolo vhodnejšie použiť formát konfiguračného súboru JSON alebo XML. Ďalším vhodným rozšírením by bola implementácia RTOS, čo by umožnilo profiláciu programu. Tieto úpravy by nie len zlepšili funkcionalitu, ale zároveň by umožnili lepšie prepojenie s testovacím a validačným frameworkom.

Literatúra

[1] *G. TONG, Jason, Ian D. L. ANDERSON a Mohammed A. S. KHALID. Soft-Core Processors for Embedded Systems* [online]. University of Windsor – Department of Electrical and Computer Engineering Research Centre for Integrated Microsystems Windsor, Ontario, Canada, 2006 [cit. 2021-9-26]. Dostupné z URL: <`https://reds.heig-vd.ch/share/cours/reco/documents/soft_core_processors.pdf/`> University of Windsor.

[2] *NADE, J. B. a Dr. R. V. SARWADNYA. The Soft Core Processors: A Review. INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN ELECTRICAL* [online]. 2015, December 2015, , 7 [cit. 2021-10-2]. Dostupné z URL: <`https://ijireeice.com/upload/2015/december-15/IJIREEICE%2041.pdf`>

[3] *P. Yiannacouras, J. G. Steffan and J. Rose, Ëxploration and Customization of FPGA-Based Soft Processors,"* in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 2, pp. 266-277, Feb. 2007, doi: 10.1109/TCAD.2006.887921. [cit. 2021-10-5].

[4] *JAYARAMAN, Rajeev. FPGA Vs ASIC: Differences Between Them And Which One To Use? Numato* [online]. 2001, 5 [cit. 2021-11-03]. Dostupné z: <`https://numato.com/blog/differences-between-fpga-and-asics/`>

[5] *JAYAKRISHNAN, Vivek a Chirag PARIKH. Embedded Processors on FPGA: Soft vs Hard* [online]. School of Engineering Grand Valley State University, Grand Rapids, MI, 49504, 2019 [cit. 2021-10-2]. Dostupné z URL: <`http://people.cst.cmich.edu/yelam1k/asee/proceedings/2019/1/6.pdf`>

[6] *HÁJEK, Radek. Implementace mikroprocesoru AVR do obvodu FPGA* [online]. Brno, 2014 [cit. 2021-10-18]. Dostupné z: <`http://hdl.handle.net/11012/33997`>. Bakalářská práce. Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií. Ústav mikroelektroniky. Vedoucí práce Marián Pristach.

[7] *YADAV, Chandu. Arithmetic Logic Unit (ALU). Tutorialspoint* [online]. 02.01.2019 [cit. 2021-11-02]. Dostupné z: <`https://www.tutorialspoint.com/arithmetic-logic-unit-alu`>

[8] *CPU's Control Unit. Witscad* [online]. 2019 [cit. 2021-11-02]. Dostupné z: <`https://witscad.com/course/computer-architecture/chapter/cpu-control-unit`>

[9] *Assembly - Registers. Tutorialspoint* [online]. web [cit. 2021-11-02]. Dostupné z: <https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm>

[10] *AGARWAL, Khushal. Different Classes of CPU Registers. Geeksforgeeks* [online]. 2020, 23 Dec, 2020 [cit. 2021-11-02]. Dostupné z: <https://www.geeksforgeeks.org/different-classes-of-cpu-registers/>

[11] *MACHO, Tomáš. Vestavné systémy a mikroprocesory* [online]. Brno, 2020 [cit. 2021-11-02]. Skripta. VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ.

[12] *FAHAD, Engr. C++ Programming Environment: Editor, Compiler, Linker, Debugger, Profiler. Electroniclinic.com* [online]. 2020, April 2, 2021 [cit. 2021-11-07]. Dostupné z: <https://www.electroniclinic.com/c-programming-environment-editor-compiler-linker-debugger-profiler/#The_Profiler>

[13] *PONČÁK, Matej. Rozšíření překladače jazyka C o podporu dalších embedded mikroprocesorů* [online]. VUT v Brně, 2020 [cit. 2021-11-07]. Dostupné z: <https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=216576.> Bakalárska práca. Vysoké učení technické v Brně. Vedoucí práce Ing. Petr Petyovský, Ph.D.

[14] *ONSMAN, Alex. What is a Debugger Program? Tutorialspoint.com* [online]. 2018, 28-Aug-2018 [cit. 2021-11-07]. Dostupné z: <https://www.tutorialspoint.com/what-is-a-debugger-program>

[15] *Profiling (computer programming). Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001 [cit. 2021-11-07]. Dostupné z: <https://en.wikipedia.org/wiki/Profiling_(computer_programming)>

[16] *Hardware Description Language. Www.mepits.com* [online]. 15 July 2014 [cit. 2021-11-17]. Dostupné z: <https://www.mepits.com/tutorial/143/vlsi/hardware-description-language>

[17] *Data Objects: Signals, Variables and Constants* [online]. [cit. 2021-11-18]. Dostupné z: <http://www.euroelectronica.ro/5-data-objects-signals-variables-and-constants/>

[18] *Verilog-vs-systemverilog. In: Educba* [online]. [online]. [cit. 2022-05-14]. Dostupné z: <https://www.educba.com/verilog-vs-systemverilog/>

Verilog-vs-systemverilog. In: Educba [online]. [cit. 2022-05-14]. Dostupné z: https://www.educba.com/verilog-vs-systemverilog/

[19] *What is An RTOS? Freertos* [online]. [cit. 2021-10-27]. Dostupné z: <https://www.freertos.org/about-RTOS.html>

[20] *Real-Time Data Processing. Techopedia* [online]. [cit. 2021-10-26]. Dostupné z: <https://www.techopedia.com/definition/31742/real-time-data-processing>

[21] *TRIVEDI, Panini. Real Time Operating System (RTOS) With Its Effective Scheduling Techniques* [online]. 2020, 4 [cit. 2021-11-01]. Dostupné z: <https://www.ijedr.org/papers/IJEDR1302020.pdf>

[22] *Gaisler: LEON3 Processor*[online]. Goteborg, Sweden [cit. 2021-11-01]. Dostupné z: <https://www.gaisler.com/index.php/products/processors/leon3>

[23] *MicroBlaze™ RISC 32-Bit Soft Processor*[online]. In: . 2002, August 21, 2002, s. 5 [cit. 2022-05-16]. Dostupné z: <https://www.ee.ryerson.ca/~courses/ee8205/Data-Sheets/sopc/MicroBlaze_DataSheet.pdf>

[24] *OpenCores* 2001 [cit. 2021-11-01]. Dostupné z: <https://opencores.org/projects/plasma>

[25] *NOLTING, Ing. Stephan. The NEORV32 RISC-V Processor: Datasheet* [online]. In: . [cit. 2021-12-16]. Dostupné z: <https://stnolting.github.io/neorv32/>

[26] *NOLTING, Ing. Stephan. The NEORV32 RISC-V Processor: User Guide* [online]. In: . [cit. 2022-5-13]. Dostupné z: <https://stnolting.github.io/neorv32/ug/>

[27] *Intel Quartus Prime Design Software: Compare PRO, STANDARD and LITE Editions. Intel.com* [online]. Intel Corporation [cit. 2021-11-08]. Dostupné z: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/ss-quartus-comparison.pdf>

[28] *Vivado ML. xilinx.com* [online]. Xilinx [cit. 2021-11-08]. Dostupné z: <https://www.xilinx.com/products/design-tools/vivado/vivado-ml.html#licensing>

[29] *About GHDL. Ghdl.github.io* [online]. [cit. 2022-05-14]. Dostupné z: <https://ghdl.github.io/ghdl/about.html>

[30] *SystemVerilog TeshBench* [online]. In: . s. 6 [cit. 2022-02-25]. Dostupné z: <https://www.chipverify.com/systemverilog/systemverilog-simple-testbench>

[31] *UART Explained* UART Explained. In: Www.developer.electricimp.com [online]. [cit. 2022-04-28]. Dostupné z: <https://developer.electricimp.com/resources/uart>

A Obsah elektronické přílohy

- `/` ..... koreňový adresár priloženého archívu
  - `neorv32` ..... adresár so zdrojovými súbormi procesora NEORV32
    - `...`
  - `testbench`
    - `README.md` ..... anglicky návod na nastavenie projektu a spustenie testov
    - `create_project.tcl` ..... skript na vytvorenie projektu
    - `neorv32_test.tcl` ..... testovací skript
    - `neorv32_sv_simple_tb.sv` ..... zdrojový súbor pre testbench
    - `test_files` ..... testovacie scenáre
      - `hello_world`
        - `config.txt`
        - `input_data.csv`
        - `output_data.csv`
        - `main.c`
        - `neorv32_application_image.vhd`
        - `...` ..... ďalšie pomocné súbory na preloženie softvéru
      - `blink_led`
        - `...` ..... rovnaké ako v predošlých scenároch
      - `gpio_loopback`
        - `...` ..... rovnaké ako v predošlých scenároch
      - `demo_pwm`
        - `...` ..... rovnaké ako v predošlých scenároch
      - `uart_loopback`
        - `...` ..... rovnaké ako v predošlých scenároch
      - `testing_scenario` ..... kombinovaný test
        - `...` ..... rovnaké ako v predošlých scenároch