FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Simulace procesorů v jazyce SystemVerilog
Student:	Bc. Vojtěch Jílek
Vedoucí:	Ing. Martin Kohlík, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Návrh a programování vestavných systémů
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Zadání práce:

- Seznamte se s konstrukcemi jazyka SystemVerilog a jeho knihovny UVM. Zaměřte se na část pro práci s registrovou vrstvou (RAL).
- Ověřte principy práce s RAL na jednoduchém procesoru.
- Sestavte verifikační prostředí pro pokročilý procesor (vícestupňový s pipeline) a otestujte jej.
- Vytvořte stručný nápomocný text k používaní RAL vrstvy s popisem hlavních problémů, se kterými se může začínající vývojář setkat.

Elektronicky schválil/a prof. Ing. Hana Kubátová, CSc. dne 22. prosince 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Simulace procesorů v jazyce SystemVerilog

Bc. Vojtěch Jílek

Katedra číslicového návrhu
Vedoucí práce: Ing. Martin Kohlík, Ph.D.

26. března 2022

Poděkování

Chtěl bych poděkovat především svému vedoucímu Ing. Martinu Kohlíkovi za jeho rady a ochotu se vším pomoci. Dále bych chtěl poděkovat i své rodině za vytrvalou podporu a vytvoření příjemného prostředí k psaní práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 26. března 2022 . . . . . .

Odkaz na tuto práci

Jílek, Vojtěch. *Simulace procesorů v jazyce SystemVerilog*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2022.

Abstrakt

Tato práce se zabývá návrhem simulačních prostředí pro simulaci procesorů v jazyce SystemVerilog. K simulaci procesorů je využita knihovna UVM, její registrový model a vývojové prostředí QuestaSim. V této práci je navrženo simulační prostředí pro dva procesory – jednocyklový procesor a zřetězený procesor. Součástí této práce je i stručný text s popisem několika problémů, se kterými se může začínající vývojář setkat při využívání registrového modelu knihovny UVM.

Klíčová slova SystemVerilog, knihovna UVM, registrový model, QuestaSim, simulace, jednocyklový procesor, zřetězený procesor

Abstract

This thesis deals with design of simulation environments for processor simulation in the SystemVerilog language. The UVM library, its register model and the QuestaSim development environment are used to simulate processors. In this work, a simulation environment for two processors is designed - a single-cycle processor and a pipeline processor. Part of this thesis is a brief text with a description of several problems that a novice developer may encounter when using the registry model of the UVM library.

Keywords SystemVerilog, UVM library, register model, QuestaSim, simulation, single-cycle processor, pipeline processor

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Verifikace se v posledních letech stává velmi důležitou součástí návrhu číslicových obvodů, a to proto, že každá neodhalená chyba v návrhu může stát velké množství peněz. Jako příklad může sloužit chyba FDIV odhalená v roce 1994 v procesoru Intel Pentium – Intel své ztráty vyčíslil na 500 milionů USD [1]. Chyby v návrhu samozřejmě mohou být odhaleny i během testování. Verifikace má ale výhodu v tom, že je ještě levnější, a navíc zařízení ještě nemusí být fyzicky vyrobeno.

Tato práce se zabývá vývojem verifikačních prostředí pro procesory a to jak pro jednocyklové, tak i vícestupňové. Pro vývoj verifikačních prostředí je využit jazyk SystemVerilog, knihovna UVM a její registrová vrstva (RAL).

Práce je rozdělena do tří hlavních kapitol: Knihovna UVM, Analýza a Realizace. V kapitole Knihovna UVM je představena knihovna UVM, její hlavní komponenty a dále pak její registrová vrstva (RAL), komponenty registrové vrstvy a metody přístupu k registrům. Kapitola Analýza obsahuje popis zvolených procesorů pro verifikaci a stručné představení využitého simulačního prostředí. Poslední kapitola Realizace se zabývá samotným návrhem verifikačních prostředí pro oba procesory.

Kapitola 1

Cíl práce

Cílem této práce je seznámit se s knihovnou UVM a její registrovou vrstvou a poté navrhnout verifikační prostředí pro dva procesory – nejprve pro jednoduchý jednocyklový procesor a poté i pro vícestupňový (zřetězený) procesor. Pro vývoj verifikačních prostředí bude využit jazyk SystemVerilog a simulační prostředí QuestaSim. Struktura navržených testbenchů bude založena na knihovně UVM a budou využity registrové modely (RAL).

Cílem kapitoly Knihovna UVM je představení základů UVM knihovny, jejích komponent a simulačních fází. V této kapitole je také představena registrová vrstva (RAL) a její komponenty.

Cílem kapitoly Analýza je představení obou verifikovaných procesorů, jejich instrukčních sad a jejich struktury s důrazem na registry, které bude nutné zahrnout do registrového modelu. V případě zřetězeného procesoru jsou v této kapitole analyzovány také hazardy a jejich řešení. V závěru kapitoly Analýza je stručně představeno využité vývojové prostředí QuestaSim.

Cílem kapitoly Realizace je popis postupu při návrhu verifikačních prostředí a popis samotných navržených komponent.

Kapitola 2

Knihovna UVM

UVM (Uviversal Verification Metodology) je standard, který umožňuje rychlý vývoj a znovupoužitelnost verifikačních prostředí. Je to řada třídních knihoven definovaných pomocí syntaxe a sémantiky jazyka SystemVerilog a je nyní standardem IEEE. Hlavní myšlenka za UVM je pomoci společnostem vyvíjet modulární, znovupoužitelné a škálovatelné testbenchové struktury poskytnutím API frameworku, který lze nasadit do více projektů [2].

UVM bylo odvozeno převážně z OVM (Open Verification Metodology). Knihovna UVM poskytuje generické nástroje jako konfigurační databáze, TLM a hierarchii komponent a také přináší vrstvu abstrakce, kde má každý komponent ve verifikačním prostředí specifickou roli [2]. Například driver je odpovědný pouze za přivádění signálů na rozhraní DUT, oproti tomu monitor je odpovědný pouze za sledování signálů na rozhraní DUT.

2.1 Struktura testbenche knihovny UVM

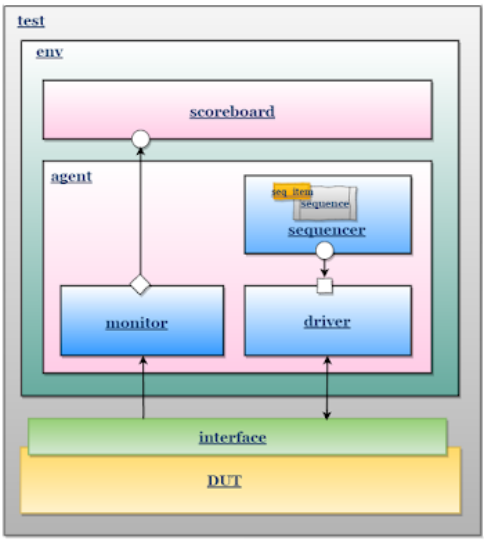
Na obrázku 2.1 můžete vidět základní strukturu testbenche, kterou kopíruje většina projektů využívajících knihovnu UVM. Knihovna UVM poskytuje bázové třídy všech komponent na tomto obrázku, vývojář si ale musí implementovat vlastní podtřídy těchto tříd a z nich si sestavit vlastní simulační prostředí. Všechny uživatelsky definované podtřídy UVM komponent musejí být registrovány do *uvm_factory* pomocí makra:

```
`uvm_component_utils()
```

V následujících podkapitolách bude popsána funkce jednotlivých komponent.

2.1.1 Item

Item se skládá z datových položek, které jsou nezbytné pro generování stimulů pro DUT. Za účelem generování stimulů jsou itemy v sekvencích randomizovány, proto je vhodné datové položky v itemu deklarovat jako *rand* a nastavit



Obrázek 2.1: Struktura testbenche knihovny UVM [3]

pro ně vhodná omezení (*constraint*). Kromě randomizace je možné do itemu přidat i kontrolu pokrytí pomocí *coverpoint* [4].

Item dědí od třídy *uvm_sequence_item* a musí obsahovat minimálně konstruktor a makra pro registraci všech položek do *factory* [4]:

```
`uvm_object_utils_begin(my_item)
    `uvm_field_int(data1,UVM_DEFAULT)
    `uvm_field_int(data2,UVM_DEFAULT)
    ...
`uvm_object_utils_end
```

Dále je možné využít také metody *pre_randomize* a *post_randomize*, které jsou volány před, respektive po randomizaci. V těchto metodách je možné nastavit některé datové položky a případně vyřešit omezení randomizace, která nelze implementovat pomocí *constraint*.

2.1.2 Sequence

Sekvence se skládají z několika itemů, které lze různými způsoby poskládat a vytvořit tak testovací scénáře. Jsou prováděny přiřazeným sekvencerem, který odesílá data do *driveru*. Sekvence tedy tvoří hlavní stimuly v simulačním prostředí [5].

Sekvence musí dědit od třídy *uvm_sequence* a musí obsahovat minimálně konstruktor a metodu *body*, ve které dochází ke generování stimulů. Pro jednoduché sekvence, které sestávají pouze z jedné datové položky je možné využít

makro *'uvm_do(item)*, které alokuje místo pro nový item, randomizuje ho a odešlo ho na výchozí sekvencer [5].

2.1.3 Sequencer

Sekvencer slouží k převodu sekvencí na datové transakce a odesílání transakcí do driveru. Sekvencer dědí od třídy *uvm_sequencer*, která je parametrizovatelná typem transakce (item) požadavku a odpovědi driveru (pokud jsou stejné, stačí zadat jeden) [6]. V některých případech – obzvláště pro jednoduché projekty – je možné využít přímo třídu *uvm_sequencer* díky tomu, že tato třída již obsahuje veškerou funkcionalitu, která je nutná pro komunikaci s driverem.

2.1.4 Driver

Driver je komponent, který má informace o rozhraní DUT a přivádí stimuly na toto rozhraní. Transakce na jejichž základě je stimulováno rozhraní DUT jsou získávány ze sekvenceru. Driver dědí od třídy *uvm_driver* a měl by implementovat minimálně konstruktor a metody *build_phase* a *run_phase*. Ve fázi build by si měl driver z konfigurační databáze vyzvednout referenci na rozhraní DUT pomocí funkce:

```
uvm_config_db#(virtual dut_if)::get(this, "", "vif", vif)
```

Ve fázi run dochází k buzení DUT – nejprve je třeba od sekvenceru získat novou transakci, k čemuž slouží členská proměnná *seq_item_port* a její metoda *get_next_item*. Na základě přijaté transakce dojde k přivedení odpovídajících signálů na rozhraní DUT a na závěr je sekvenceru odeslána odpověď označující transakci za hotovou pomocí metody *seq_item_port.item_done* [7].

2.1.5 Monitor

Monitor je odpovědný za sledování rozhraní DUT a převod signálů na datové transakce, které odesílá scoreboardu. K tomu potřebuje referenci na rozhraní DUT a také TLM analytický port (pro komunikaci). Monitor signály pouze sleduje a odesílá dále (v podobě transakcí), ke kontrole správnosti sledovaných hodnot dochází až ve scoreboardu [8].

Monitor dědí od třídy *uvm_monitor* a měl by poskytovat minimálně konstruktor a metody *build_phase* a *run_phase*. Ve fázi build si monitor, podobně jako driver, z konfigurační databáze vyzvedne referenci na rozhraní DUT pomocí funkce:

```
uvm_config_db#(virtual dut_if)::get(this, "", "vif", vif)
```

Ve fázi run jsou sledovány signály a převáděny na transakce, které jsou odeslány analytickým portem pomocí metody *analys_port.write(item)* [8].

2.1.6 Agent

Agent zapouzdřuje sekvencer, driver a monitor do jedné entity a je odpovědný za jejich instanciaci a propojení. Slouží pro oddělení testbenche od DUT – vně agenta by nemělo být nic závislé na rozhraní DUT, či protokolu komunikace (tím je zaručeno, že při změnách v rozhraní DUT bude třeba upravit jen agenta) [9].

Agent dědí od třídy *uvm_agent* a musí obsahovat minimálně konstruktor a metody *build_phase* a *connect_phase*. V build fázi agent vytvoří nové instance monitoru, sekvenceru a driveru a ve fázi connect propojí driver se sekvencerem pomocí metody:

```
my_driver.seq_item_port.connect(my_sequencer.seq_item_export);
```

2.1.7 Scoreboard

Scoreboard je odpovědný za kontrolu správné funkčnosti DUT, k tomu potřebuje TLM analytický port, pomocí kterého komunikuje s monitorem [10]. Scoreboard dědí od třídy *uvm_scoreboard* a musí obsahovat minimálně konstruktor a metody *build_phase* a *run_phase*. Fáze build slouží k inicializaci analytického portu a ve fázi run scoreboard vybírá transakce z analytického portu a kontroluje správnost výsledků.

2.1.8 Environment

Environment slouží k zapouzdření agentů a scoreboardu, případně i dalších environment (tyto komponenty by teoreticky mohly být instanciovány přímo v testu, ale pro lepší znovupoužitelnost je doporučeno vytvořit environment - jeden environment pak může být použit pro více testů) [11]. Environment dědí od třídy *uvm_env* a musí obsahovat minimálně konstruktor a metodu *build_phase*, která je odpovědná za inicializaci vlastněných komponent (agent, scoreboard). Dále může obsahovat také funkci *connect_phase*, která slouží k propojení komponent (typicky propojení analytických portů agenta a scoreboardu).

2.1.9 Test

Test je odpovědný za vytváření a spouštění sekvencí a definuje testovací scénář, který je testován v daném testbenchi. Test dědí od třídy *uvm_test* a musí obsahovat minimálně konstruktor a metody *build_phase* a *run_phase*. Ve fázi build dochází k inicializaci environmentu a ve fázi run jsou spouštěny testovací sekvence. Test je spouštěn z top modulu testbenche pomocí funkce *run_test("jméno testu")* [12].

2.2 Simulační fáze knihovny UVM

Simulace, které využívají knihovnu UVM jsou rozděleny do fází, které pomáhají synchronizovat jednotlivé komponenty. Fáze jsou reprezentovány callback metodami – množina předdefinovaných fází a korespondujících callback metod je součástí třídy *uvm_component*. Každá třída, která dědí od *uvm_component* může implementovat jakékoli z těchto metod (fáze jsou uvedeny v pořadí v jakém jsou spouštěny) [13]:

- **Build** – inicializace komponent a sestavení simulačního prostředí
- **Connect** – propojovaní komponentů v simulačním prostředí
- **End_of_elaboration** – finální úpravy simulačního prostředí před zahájením simulace
- **Start_of_simulation** – výpis topologie simulačního prostředí a jeho konfigurace
- **Run** – generování stimulů, buzení DUT, kontrola výsledků
- **Extract** – zpracování informací od scoreboardů a kontroly pokrytí
- **Check** – kontrola, zda se DUT chovalo během simulace správně, vyhodnocení chyb
- **Report** – zobrazení výsledků simulace, popř. zápis do logovacích souborů
- **Final** – dokončení všech operací, které nebyly stiženy v předchozích fázích

2.3 Registrový model knihovny UVM

Registrový model, jinak také RAL (Register Abstraction Layer), knihovny UVM poskytuje soubor tříd, pomocí kterých je možné implementovat objektově orientovaný přístup k registrům a pamětem DUT. K registrům by samozřejmě šlo přistupovat i bez registrového modelu, ale registrový model přináší oproti klasickému přístupu řadu výhod [14]:

- Poskytuje vyšší vrstvu abstrakce pro přístup k registrům – lze k nim přistupovat pomocí jejich jména.
- UVM obsahuje množinu předdefinovaných testovacích sekvencí, které lze využít k otestování základní funkce registrů.
- Registrový model umožňuje frontdoor i backdoor přístup (viz dále).

- K registrům v DUT je možné přistupovat nezávisle na protokolu sběrnice (metody read a write).
- K registrům v registrovém modelu je možné přistupovat paralelně z více vláken (přístup je interně serializován).
- Znovupoužitelnost – některé komponenty registrového modelu (případně i celý registrový model) mohou být využity pro několik simulačních prostředí.

2.3.1 Komponenty registrového modelu

V této podkapitole jsou popsány nejdůležitější komponenty registrového modelu:

uvm_reg_field – dílčí položka registru, ke které lze přistupovat samostatně. Nejdůležitější metodou této třídy je *configure*, pomocí které je nutné každou položku nakonfigurovat (velikost, pozice LSB, režim přístupu, hodnota po resetu, povolení resetu, povolení randomizace a povolení individuálního přístupu) [15]:

```
function void configure( uvm_reg        parent,
                         int unsigned   size,
                         int unsigned   lsb_pos,
                         string         access,
                         bit            volatile,
                         uvm_reg_data_t reset,
                         bit            has_reset,
                         bit            is_rand,
                         bit individually_accessible )
```

uvm_reg – registr reprezentuje množinu polí *uvm_reg_field*, ke kterým lze přistupovat jako k jedné entitě. Pro každý registr si registrový model pamatuje mirrored hodnotu (hodnota naposledy přečtená/zapsaná do DUT) a desired hodnotu (hodnota, kterou by registr v DUT měl mít) [16]. Při tvorbě registrového modelu si vývojář vytváří vlastní registry (třídy), které dědí od této třídy. Registr musí vždy obsahovat alespoň konstruktor a metodu *build*, ve které dochází ke konfiguraci všech datových položek registru.

uvm_mem – reprezentuje souvislý paměťový blok. Na rozdíl od registrů paměti nemají v registrovém modelu mirrored ani desired hodnotu [17]. Oproti registrům má ale tato třída výhodu v tom, že umožňuje přístup k souvislým blokům dat (metody *burst_write a burst_read* [17]).

uvm_reg_block – reprezentuje hierarchii v DUT, může obsahovat registry, paměti a další bloky [18]. Pro každý logický soubor registrů v DUT by měl být vytvořen samostatný registrový blok v registrovém modelu. Registrové bloky by měly dědit od třídy *uvm_reg_block* a musí obsahovat minimálně konstruktor a metodu *build*, ve které je každý registr (a paměť) v daném bloku

Atribut	Datový typ	Popis
addr	uvm_reg_addr_t	Adresa datového pole
data	uvm_reg_data_t	Data pro zápis či přečtená data
kind	uvm_access_e	UVM_READ nebo UVM_WRITE
n_bits	unsigned int	Počet přenášených bitů
byte_en	uvm_reg_byte_en_t	Povolení rozdělení sběrnice po bytech
status	uvm_status_e	UVM_IS_OK, UVM_NOT_OK nebo UVM_IS_X

Tabulka 2.1: Atributy registrové transakce [20]

inicializován, přiřazen adresní mapě a je mu nastavena HDL cesta ke korespondujícímu registru v DUT (viz ukázkový kód v příloze C).

uvm_reg_map – adresní mapa, do které jsou mapovány registry a paměti [19]. Pro využití frontdoor přístupu je třeba všechny registry namapovat do nějaké adresní mapy – transakce registrového modelu přistupují k registrům pomocí jejich adres. Nastavení offsetů jednotlivých registrů v rámci adresní mapy je možné provádět pomocí metody *add_reg*, samotné mapování je poté provedeno v rámci hlavního bloku registrového modelu pomocí metody *lock_model* (po použití této metody už nelze v registrovém modelu dělat žádné změny). Při využití blokové hierarchie je možné adresní mapu zakomponovat do mapy nadřazeného bloku [19].

uvm_adapter – Funkce pro přístup k registrům generují obecné transakce registrového modelu, které je nutné převést na konkrétní sběrnicové transakce vhodné pro sběrnici DUT. K tomu slouží právě adaptér a jeho metody *reg2bus* (převod registrových transakcí na sběrnicové) a *bus2reg* (převod sběrnicových transakcí na registrové) [20]. Adaptér by měl dědit od třídy *uvm_adapter*. Struktura registrových transakcí je popsána v tabulce 2.1.

2.3.2 Metody přístupu k registrům

Registrový model umožňuje dvě možnosti přístupu k registrům a to frontdoor přístup a backdoor přístup.

Frontdoor přístup je klasický přístup přes rozhraní DUT. To znamená, že je vytvořena nová transakce, kterou driver zpracuje a na jejím základě budí DUT. Aby bylo možné frontdoor přístup využít, je nutné implementovat adaptér (který dědí od třídy *uvm_adapter*) a jeho dvě metody *bus2reg* a *reg2bus*, které slouží k převodu sběrnicových transakcí na transakce registrového modelu a naopak.

Backdoor přístup je simulovaný přístup mimo rozhraní DUT, tento přístup využívá simulační databázi k přímému přístupu k signálům v DUT a nespotřebovává žádný simulační čas. Při využití tohoto přístupu je nutné po-

čítat s tím, že není aktivována logika, kterou využívá standardní přístup (přes sběrnici DUT), a tudíž se tato logika nemusí chovat dle očekávání. Pro využití backdoor přístupu je nutné u registrů v registrovém modelu nastavit HDL cestu k odpovídajícím registrům v DUT, a to na úrovni registru pomocí metody *add_hdl_path_slice* a na úrovni bloku registrů pomocí metody *add_hdl_path* [21].

2.3.3 Funkce pro přístup k registrům v registrovém modelu

V této podkapitole jsou představeny základní funkce pro přístup k registrům (metody třídy *uvm_reg*):

```
virtual function void set( uvm_reg_data_t value,
                           string         fname   = "",
                           int            lineno  = 0 )
```

Pomocí metody set je možné nastavit *desired* hodnotu registru (vůbec nedojde k přístupu k registru v DUT – je nastavena pouze *desired* hodnota registru v registrovém modelu). Požadovanou hodnotu udává parametr *value* [16].

```
virtual function uvm_reg_data_t get( string  fname    = "",
                                     int     lineno   = 0 )
```

Metoda get vrací aktuální *desired* hodnotu registru [16].

```
virtual function uvm_reg_data_t get_mirrored_value(
        string  fname    =  "",
        int     lineno   =   0 )
```

Metoda get_mirrored_value vrací aktuální *mirrored* hodnotu registru [16].

```
virtual function void reset( string kind = "HARD" )
```

Metoda reset slouží k resetování *mirrored* i *desired* hodnot registru. Parametr *kind* je možné nastavit na „HARD“ nebo na „SOFT“ – rozdíl spočívá v tom, že registry označené jako *write-once* mohou být modifikovány i po HARD resetu, po SOFT resetu už ne [16].

```
virtual task write ( output  uvm_status_e      status,
                     input   uvm_reg_data_t    value,
                     input   uvm_path_e path = UVM_DEFAULT_PATH,
                     input   uvm_reg_map       map       = null,
                     input   uvm_sequence_base parent    = null,
                     input   int               prior     = -1,
                     input   uvm_object        extension = null,
                     input   string            fname     = "",
                     input   int               lineno    = 0 )
```

Metoda write umožňuje zápis hodnoty do registru v DUT, přičemž automaticky dochází i ke změně *desired* a *mirrored* hodnot. Hodnota, která má být zapsána do registru, je dána parametrem *value*. Výstupní parametr *status* udává, zda se operace zdařila – nabývá hodnot *UVM_OK* (operace úspěšně proběhla), *UVM_NOT_OK* (operace skončila chybou) a *UVM_HAS_X* (operace úspěšně proběhla, ale byly zpracovávány bity s neznámou hodnotou). Pomocí parametru *path* je možné nastavit způsob přístupu k registru v DUT (*UVM_FRONTDOOR/UVM_BACKDOOR*). Parametr *map* je nutné zadat pouze v případě, že je registr namapován do více adresních map a je využit frontdoor přístup [16].

```
virtual task read (  output  uvm_status_e      status,
                     output  uvm_reg_data_t    value,
                     input   uvm_path_e path = UVM_DEFAULT_PATH,
                     input   uvm_reg_map       map       = null,
                     input   uvm_sequence_base parent    = null,
                     input   int               prior     = -1,
                     input   uvm_object        extension = null,
                     input   string            fname     = "",
                     input   int               lineno    = 0 )
```

Metoda read umožňuje čtení hodnoty z registru v DUT, přičemž automaticky dochází i ke změně *desired* a *mirrored* hodnot. Hodnota přečtená z registru je uložena do výstupního parametru *value*. Výstupní parametr *status* udává, zda se operace zdařila, *path* udává způsob přístupu a *map* adresní mapu (tyto parametry jsou podrobněji popsány u metody write) [16].

```
virtual task poke(  output  uvm_status_e      status,
                    input   uvm_reg_data_t    value,
                    input   string            kind      = "",
                    input   uvm_sequence_base parent    = null,
                    input   uvm_object        extension = null,
                    input   string            fname     = "",
                    input   int               lineno    = 0 )
```

Metoda poke slouží k zapsání hodnoty do registru v DUT pomocí backdoor přístupu, přičemž automaticky dochází i ke změně *desired* a *mirrored* hodnot. Hodnota, která má být zapsána do registru, je dána parametrem *value*. Výstupní parametr *status* udává, zda se operace zdařila (viz metoda write) [16].

```
virtual task peek(  output  uvm_status_e      status,
                    output  uvm_reg_data_t    value,
                    input   string            kind      = "",
                    input   uvm_sequence_base parent    = null,
                    input   uvm_object        extension = null,
                    input   string            fname     = "",
                    input   int               lineno    = 0 )
```

Metoda peek slouží ke čtení hodnoty z registru v DUT pomocí backdoor přístupu, přičemž automaticky dochází i ke změně *desired* a *mirrored* hodnot. Hodnota přečtená z registru je uložena do výstupního parametru *value*. Výstupní parametr *status* udává, zda se operace zdařila (viz metoda write) [16].

```
virtual task update( output uvm_status_e      status,
                     input  uvm_path_e  path = UVM_DEFAULT_PATH,
                     input  uvm_reg_map       map       = null,
                     input  uvm_sequence_base parent    = null,
                     input  int               prior     = -1,
                     input  uvm_object        extension = null,
                     input  string            fname     = "",
                     input  int               lineno    = 0 )
```

Pomocí metody update je možné *desired* hodnotu registru v registrovém modelu zapsat do registru v DUT. Výstupní parametr *status* udává, zda se operace zdařila, *path* udává způsob přístupu a *map* adresní mapu (tyto parametry jsou podrobněji popsány u metody write) [16].

```
virtual task mirror( output uvm_status_e      status,
                     input  uvm_check_e check = UVM_NO_CHECK,
                     input  uvm_path_e path = UVM_DEFAULT_PATH,
                     input  uvm_reg_map       map       = null,
                     input  uvm_sequence_base parent    = null,
                     input  int               prior     = -1,
                     input  uvm_object        extension = null,
                     input  string            fname     = "",
                     input  int               lineno    = 0 )
```

Metoda mirror čte hodnotu registru v DUT a uloží ji do mirrored hodnoty regisru v registrovém modelu. Výstupní parametr *status* udává, zda se operace zdařila, *path* udává způsob přístupu a *map* adresní mapu (tyto parametry jsou podrobněji popsány u metody write). Parametr *check* je možné nastavit na hodnotu *UVM_NO_CHECK* (nedojde k žádné kontrole) nebo *UVM_CHECK* (hodnota přečtená z DUT je porovnána s původní mirrored hodnotou a v případě neshody je vypsána chybová hláška na konzoli) [16].

```
virtual function bit predict ( uvm_reg_data_t    value,
                uvm_reg_byte_en_t be     = -1,
                uvm_predict_e     kind   = UVM_PREDICT_DIRECT,
                uvm_path_e        path   = UVM_FRONTDOOR,
                uvm_reg_map       map    = null,
                string            fname  = "",
                int               lineno = 0 )
```

Metoda predict slouží k nastavení *mirrored* hodnoty. Hodnota, která má být zapsána, je dána parametrem *value*. Funkce vrací true pokud vše proběhlo v pořádku a false v případě neúspěchu (k chybě může dojít například proto, že je do registru v současné době zapisováno odjinud) [16].

```
function bit is_busy()
```

Metoda is_busy slouží k ověření, zda je registr v registrovém modelu aktuálně zaneprázdněn (čtení/zápis) [16].

Kapitola 3

Analýza

Prvním krokem v mé práci byla volba vhodných procesorů pro simulaci – mým úkolem bylo nejprve simulovat jednoduchý jednocyklový procesor a poté i zřetězený (vícestupňový) procesor. Pro simulaci jednocyklového procesoru jsem zvolil procesor, který jsem navrhoval v rámci semestrální práce v předmětu BI-APS (zadání viz [23]).

Zřetězený procesor jsem původně dostal od svého vedoucího práce, ale bohužel se ukázalo, že jazyk, ve kterém byl navržen (VHDL), není kompatibilní s registrovým modelem (více viz kapitola *Problém s VHDL*). Proto jsem nakonec zvolil procesor navržený v SystemVerilogu, který jsem našel na webových stránkách FPGA 4 Students [25].

V následujících podkapitolách budou popsány oba zvolené procesory.

3.1 Rozdíl mezi jednocyklovým a zřetězeným procesorem

Základním rozdílem mezi jednocyklovým procesorem a zřetězeným (vícestupňovým, pipeline) procesorem je to, že jednocyklový procesor zpracuje v každém cyklu právě jednu instrukci (a naopak: každá instrukce je zpravována v právě jednom cyklu). Oproti tomu zřetězený procesor má datovou cestu rozdělenu do několika stupňů (fází) a v každé z těchto fází zpracovává jinou instrukci – tímto rozdělením se zkrátí nejdelší cesta mezi dvěma registry a proto je možné zvýšit hodinovou frekvenci procesoru a tím procesor zrychlit. Zřetězené procesory typicky obsahují 5 základních stupňů (které ale mohou být ještě dále rozděleny): načtení instrukce (IF), dekódování instrukce (ID), vykonání instrukce (EX), přístup do paměti (MEM) a zápis zpět do registrů (WB).

Zřetězený procesor ale nepřináší jen výhody – vzhledem k tomu, že je najednou zpracováváno více instrukcí, může zde docházet k různým hazardům, které je nutné řešit. Existují 3 hlavní kategorie hazardů [26]:

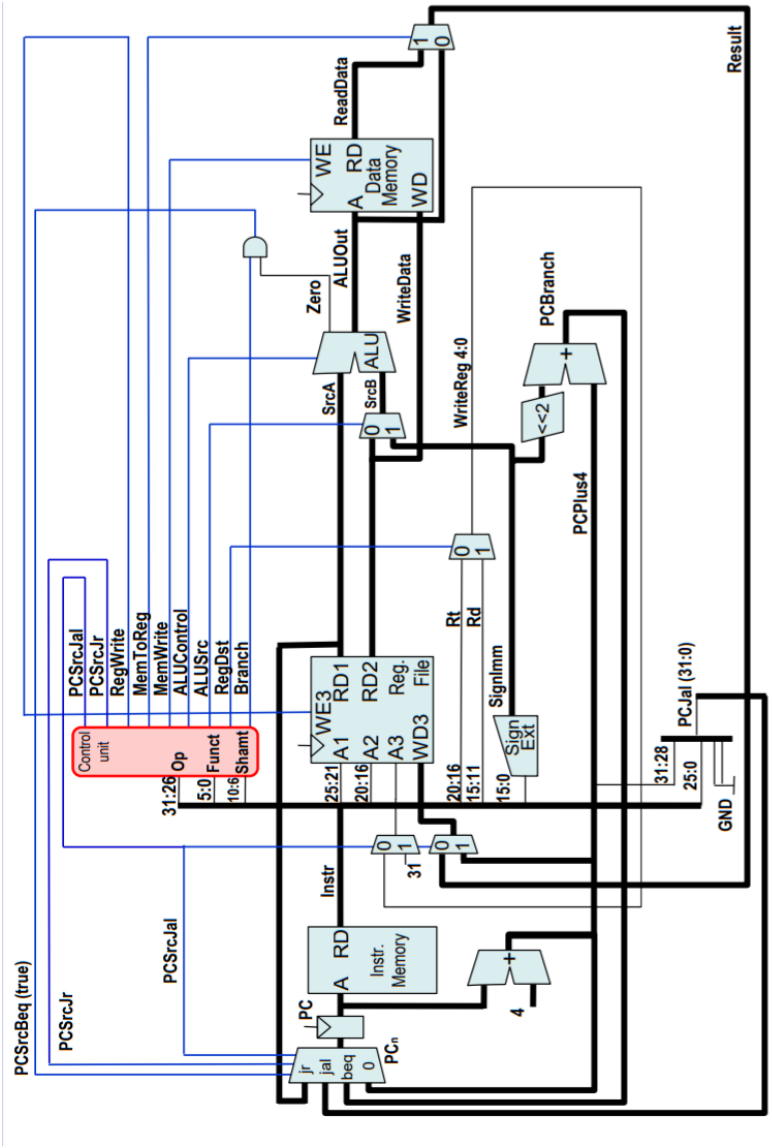
- **Strukturální**: hazardy způsobené nedostatkem hardwarových zdrojů.
- **Datové**: hazardy, které jsou způsobené tím, že aktuální instrukce závisí na předchozí instrukci, která je stále zpracovávána (dále se dělí na WAR – zápis po čtení, WAW – zápis po zápisu a RAW – čtení po zápisu).
- **Řídicí**: hazardy, které vznikají změnami PC.

Pro řešení těchto hazardů se využívají 3 techniky [26]:

- **Stall**: Pozastavení načítání nových instrukcí, dokud nebudou vypočítány nezbytné výsledky.
- **Forwarding**: Přeposílání vypočítaných hodnot ze stupňů MEM a WB zpět do stupně EX (odstranění nutnosti pozastavení).
- **Flush**: Odstranění zpracovávaných instrukcí z některých stupňů (řešení řídících hazardů).

3.2 Jednocyklový procesor

Zvolený jednocyklový procesor je 32bitový RISC procesor navržený ve Verilogu, který jsem navrhoval sám v rámci předmětu BI-APS. Instrukční sada procesoru je popsána v tabulce 3.1 (kódování instrukcí je obdobné jako u MIPS procesorů, ale je zde několik změn).



Obrázek 3.1: Schéma simulovaného jednocyklového procesoru [22]

3.2.1 Struktura procesoru

Blokové schéma procesoru můžete vidět na obrázku 3.1. Z hlediska návrhu registrového modelu jsou nejdůležitější registry obsažené v tomto procesoru: procesor obsahuje 32 32bitových GPR registrů a 32bitový program counter (PC), kromě nich je součástí navržené jednotky i instrukční a datová paměť.

Tabulka 3.1: Instrukční sada jednocyklového procesoru [23]

Instrukce	**Syntax**	**Operace**	**Kódování**
add	add d, s, t	d = s + t	0000 00ss ssst tttt dddd d000 0010 0000
sub	sub d, s, t	d = s - t	0000 00ss ssst tttt dddd d000 0010 0010
and	and d, s, t	d = s & t	0000 00ss ssst tttt dddd d000 0010 0100
or	or d, s, t	d = s | t	0000 00ss ssst tttt dddd d000 0010 0101
slt	slt d, s, t	d = (s<t) ? 1 : 0	0000 00ss ssst tttt dddd d000 0010 1010
addi	addi t, s, imm	t = s + imm	0010 00ss ssst tttt iiii iiii iiii iiii
lw	lw t, offset(s)	t = MEM[s + offset]	1000 11ss ssst tttt iiii iiii iiii iiii
sw	sw t, offset(s)	MEM[s + offset] = t	1010 11ss ssst tttt iiii iiii iiii iiii
beq	beq s, t, offset	if (s==t) PC = PC + 4 + (offset << 2) else PC = PC + 4	0001 00ss ssst tttt iiii iiii iiii iiii
jal	jal target	$31 = PC + 4 PC = (PC & 0xf0000000) | (target << 2)	0000 11ii iiii iiii iiii iiii iiii iiii
jr	jr s	PC = s	0001 11ss sss0 0000 0000 0000 0000 1000
addu.qb	addu.qb d, s, t	$d_{31:24} = s_{31:24} + t_{31:24}$ $d_{23:16} = s_{23:16} + t_{23:16}$ atd.	0111 11ss ssst tttt dddd d000 0001 0000
addu_s.qb	addu_s.qb d, s, t	$d_{31:24} = sat(s_{31:24} + t_{31:24})$ $d_{23:16} = sat(s_{23:16} + t_{23:16})$ atd.	0111 11ss ssst tttt dddd d001 0001 0000

sllv	sllv d, t, s	d = t << s	0000 00ss ssst tttt dddd dxxx xx00 0100
srlv	srlv d, t, s	d = (unsigned)t >> s	0000 00ss ssst tttt dddd d000 0000 0110
srav	srav d, t, s	d = (signed)t >> s	0000 00ss ssst tttt dddd d000 0000 0111
j	j target	PC = (PC & 0xf0000000) | (target << 2)	0000 10ii iiii iiii iiii iiii iiii iiii

3.3 Zřetězený procesor

Zvolený procesor je 32bitový 5stupňový MIPS procesor navržený v jazyce SystemVerilog, který jsem získal z webových stránek FPGA 4 Students [25]. Instrukční sada je popsána v tabulce 3.2 a plně odpovídá MIPS architektuře.

Tabulka 3.2: Instrukční sada zřetězeného procesoru [25]

Instrukce	**Syntax**	**Operace**	**Kódování**
add	add d, s, t	d = s + t	0000 00ss ssst tttt dddd d000 0010 0000
sub	sub d, s, t	d = s - t	0000 00ss ssst tttt dddd d000 0010 0010
slt	slt d, s, t	d = (s<t) ? 1 : 0	0000 00ss ssst tttt dddd d000 0010 1010
jr	jr s	PC = s	0000 00ss sss0 0000 0000 0000 0000 1000
j	j target	PC=(PC & 0xf0000000) | (target « 2)	0000 10ii iiii iiii iiii iiii iiii iiii
bne	bne s, t, offset	if (s != t) PC=PC + 4 + (offset « 2); else PC=PC + 4;	0001 01ss ssst tttt iiii iiii iiii iiii
xori	xori t, s, imm	t = s XOR imm	0011 10ss ssst tttt iiii iiii iiii iiii

lw	lw t, offset(s)	t = MEM[s + offset]	1000 11ss ssst tttt iiii iiii iiii iiii
sw	sw t, offset[s]	MEM[s + offset] = t	1010 11ss ssst tttt iiii iiii iiii iiii

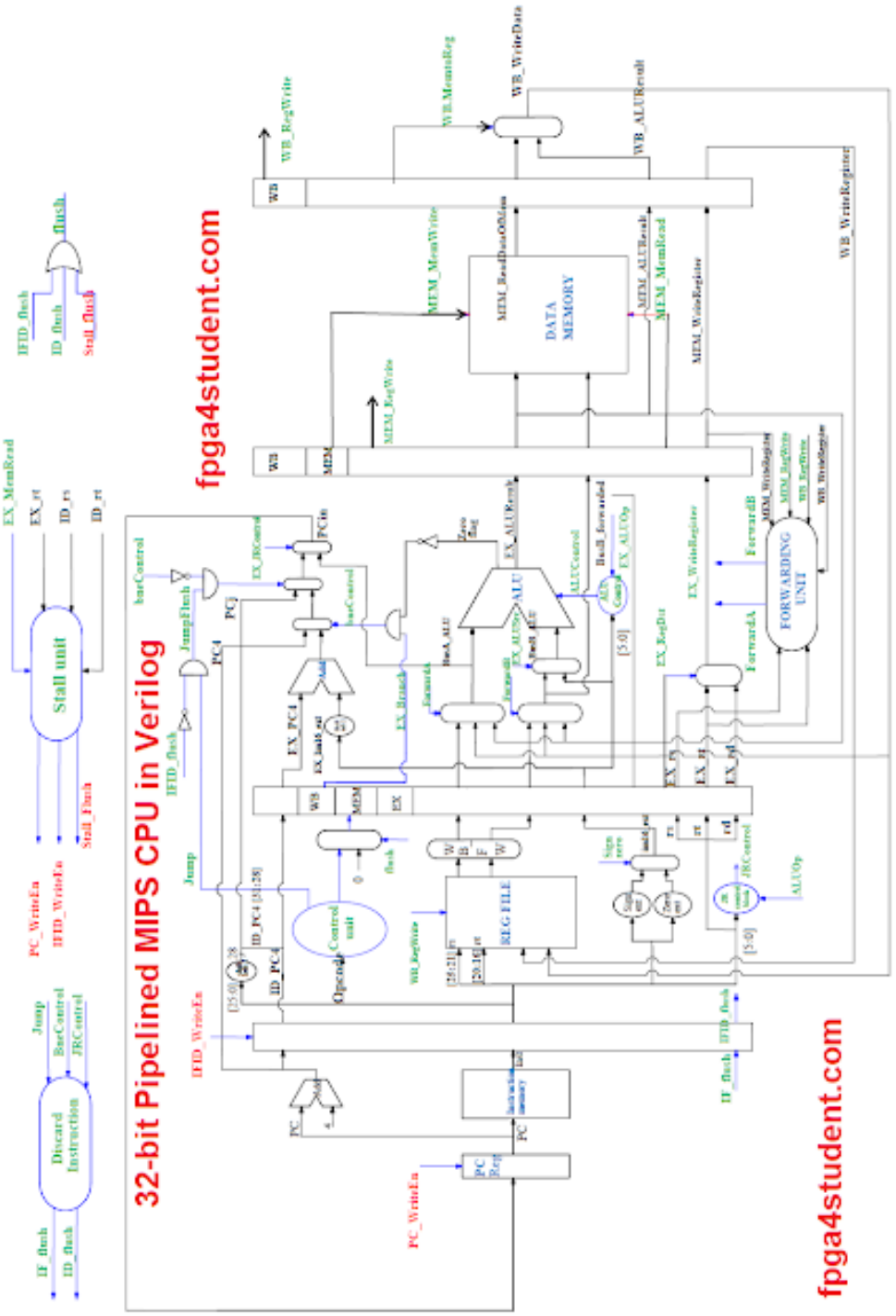
3.3.1 Struktura procesoru

Schéma procesoru můžete vidět na obrázku 3.2 (obrázek bohužel není v nejlepší kvalitě, ale návrhář procesoru bohužel lepší neposkytl). Datová cesta je rozdělena do 5 stupňů: načtení instrukce (IF), dekódování instrukce (ID), vykonání instrukce (EX), přístup do paměti (MEM) a zápis do registrů (WB). Z hlediska návrhu registrového modelu jsou opět nejdůležitější registry – procesor obsahuje 32 32bitových GPR registrů, 32bitový program counter a dále pak registry sloužící pro oddělení jednotlivých stupňů procesoru.

3.3.2 Řešení hazardů

Ve vybraném procesoru vznikají řídící hazardy a datové hazardy typu RAW (Read After Write). Řídící hazardy vznikají při skokových instrukcích (j, jr a bne) a pro jejich řešení procesor obsahuje jednotku *Flush Control Unit*, která v případě skoku aktivuje signál Flush, kterým jsou odstraněny instrukce ze stupňů IF a ID [24].

Datové hazardy RAW vznikají v případě, že aktuální instrukce ve stupni ID má jako jeden z operandů výsledek, který ještě nebyl vypočítán (je zpracováván instrukcí ve stupni EX, MEM, nebo WB). Pro řešení těchto hazardů procesor obsahuje jednotky *Forwarding Unit* a *Stall Control Unit*. *Forwarding Unit* je využita v případě, že výsledek je již vypočítán a pouze nebyl dosud zapsán do registru – v tomto případě je vypočítaná hodnota z fáze MEM nebo WB přeposlána zpět do EX. *Stall Control Unit* je využita v případě, že potřebný výsledek ještě nebyl vypočítán – v takovém případě je třeba pozastavit načítání nových instrukcí, dokud nebude výsledek vypočítán [24].



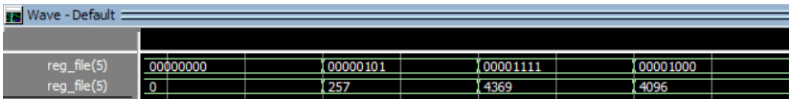
Obrázek 3.2: Schéma simulovaného zřetězeného procesoru [24]

3.3.3 Problém s VHDL

Jak již bylo řečeno v úvodu analýzy, původně jsem chtěl simulovat zřetězený procesor navržený ve VHDL, který mi poskytl můj vedoucí práce. Při testování přístupu k registrům se ale ukázalo, že backdoor zápis do registrů v modulu navrženém ve VHDL bohužel nefunguje správně. Problém je zobrazen na následujícím příkladu:

```
m_reg_model.m_GPR_gr.my_GPR[5].poke(status, 5);
#1ns;
m_reg_model.m_GPR_gr.my_GPR[5].poke(status, 15);
#1ns;
m_reg_model.m_GPR_gr.my_GPR[5].poke(status, 8);
```

Hodnoty registru 5:



Obrázek 3.3: Test zápisu do registrů (VHDL)

Zde je vidět, že jsem se pokoušel do registru zapsat hodnoty 5, 15 a 8 (0101, 1111, 1000 dvojkově), ale ze zobrazeného průběhu hodnot registru je vidět, že do něj byly zapsány hodnoty 0101, 1111 a 1000 šestnáctkově (257, 4369 a 4096 desítkově). Tento problém se vyskytl pouze u backdoor zápisu – backdoor čtení a frontdoor čtení i zápis fungovaly správně. Teoreticky tedy bylo možné ke všem registrům přidělat frontdoor přístup z vně procesoru, to by ale při velkém množství registrů ve zřetězeném procesoru bylo nepřiměřeně složité, a proto jsem se rozhodl radši zvolit procesor navržený v SystemVerilogu, kde bude backdoor přístup fungovat správně.

3.4 Vývojové prostředí

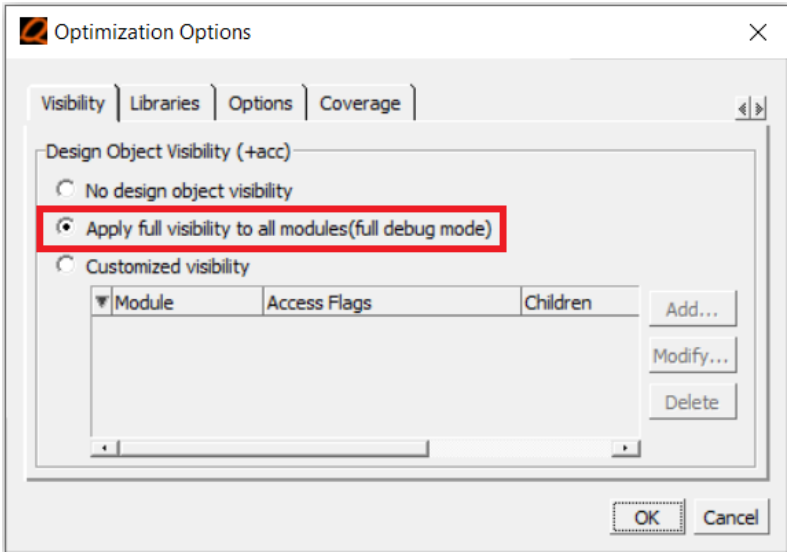
Jako vývojové a simulační prostředí jsem zvolil Questa Sim, protože jsem ho využíval už v předmětu NI-SIM a škola na něj má licenci. Questa Sim je software vyvinutý společností Mentor Graphics (dnes Siemens – Siemens koupil Mentor Graphics v roce 2017 [27]) za účelem pomoci s verifikací větších systémů. Popis Questy od výrobce:

„Questa automatizuje verifikaci a ladění složitých SoC a FPGA, dramaticky zvyšuje produktivitu a pomáhá společnostem řídit zdroje efektivněji. Questa využívá nejlepší technologie ve své třídě, které maximalizují efektivitu ověřování na úrovni bloku, subsystému a systému.“[28]

Questu je možné získat v 32bitové i 64bitové variantě pro operační systémy Windows nebo Linux.

3.4.1 Nastavení Questy

Při využití registrového modelu knihovny UVM a backdoor přístupu k registrům v DUT je třeba při spouštění simulace v Questě nastavit plnou viditelnost všech modulů. To je možné udělat v dialogu „Start Simulation“ kliknutím na tlačítko „Optimalization Options...“ a v nově otevřeném dialogu je pak třeba zvolit možnost „Aplly full visibility to all modules (full debug mode)“ (viz obrázek 3.4). Toto nastavení je třeba provést pouze před prvním spuštěním simulace, při dalších už si Questa nastavení pamatuje.



Obrázek 3.4: Nastavení viditelnosti modulů v programu Questa

Při ladění chyb v simulaci je také vhodné změnit parametr *UVM_VERBOSITY* na hodnotu *UVM_DEBUG* – s tímto nastavením vypisují třídy knihovny UVM debuggovací informace. Nastavení tohoto parametru je opět možné provést v dialogu „Start Simulation“ na záložce „Verilog“ v poli „User defined arguments“ – do tohoto pole je třeba připsat: +UVM_VERBOSITY=UVM_DEBUG. Při nastavování tohoto parametru je ale třeba nejprve zvolit simulovaný modul a pak teprve připsat parametr – v opačném pořadí by parametr po zvolení simulovaného modulu zmizel.

Kapitola 4

Realizace

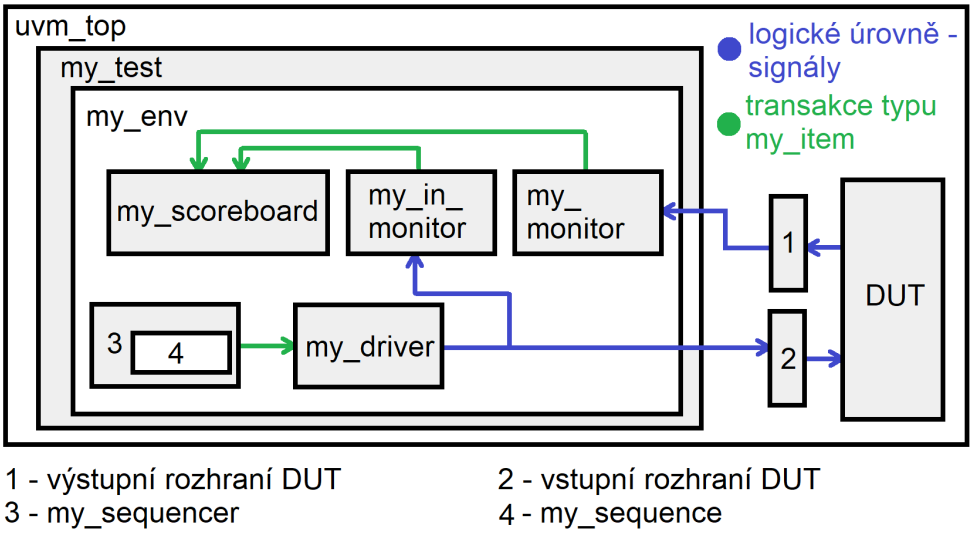
4.1 Kontrola pokrytí

Všechny simulace obou procesorů jsou řízeny kontrolou pokrytí – simulace je ukončena v případě, že bylo dosaženo pokrytí 95%. Body pokrytí (coverpoints) jsou definovány ve třídách *my_item* a *my_instruction* (viz další kapitoly). Body pokrytí jsou vytvořeny pro operační kód každé instrukce, operandy (registry) RS, RT a RD, pro 16bitovou a 26bitovou přímou hodnotu v instrukci a pro funkční kódy ALU instrukcí. Pro zkrácení simulačního času je nastaven maximální počet binů u bodu pokrytí na 8 – např. u 26bitové přímé hodnoty by bylo časově neúnosné testovat všech 2^{26} možných kombinací. Dále jsou vytvořeny body pokrytí pro každou instrukci, které vždy slučují operační kód s operandy dané instrukce – jako příklad uvádím bod pokrytí instrukce addi:

```
//operační kód
OP_ADDI: coverpoint instruction[31:26]
{
    bins x[] = {6'b001000};
}
// operandy
REGS: coverpoint instruction[25:21];
REGT: coverpoint instruction[20:16];
IMM16: coverpoint instruction[15:0];
// celá instrukce
ADDI: cross OP_ADDI, REGS, REGT, IMM16;
```

4.2 Simulace jednocyklového procesoru bez RAL

První simulační prostředí, které jsem navrhl, bylo pro simulaci jednocyklového procesoru bez využití registrového modelu, a to proto, abych mohl poté



Obrázek 4.1: Simulační prostředí jednocyklového procesoru bez RAL

porovnat simulační prostředí bez registrového modelu a s registrovým modelem. Strukturu navrženého prostředí můžete vidět na obrázku 4.1 – téměř se neliší od obecné struktury UVM testbenche uvedené v kapitole 2.1, pouze je zde navíc jeden monitor, který sleduje vstupy procesoru. V této variantě jsem simuloval pouze samotný procesor bez instrukční a datové paměti – potřebné hodnoty jsem dodával z testbenche.

Funkce jednotlivých komponent bude popsána v následujících podkapitolách.

4.2.1 My_item

Třída *my_item* reprezentuje datovou položku transakce a dědí od třídy *uvm_sequence_item*. Položka obsahuje 6 datových polí, a to:

```
rand bit [31:0] instruction;     // instrukce pro procesor
rand bit [31:0] data_from_mem;   // data z datové paměti
bit [31:0] PC;                   // hodnota PC
bit [31:0] data_to_mem;          // data do datové paměti
bit [31:0] mem_addr;             // adresa do datové paměti
bit mem_we;                      // WE datové paměti
```

Třída *my_item* je odpovědná za randomizaci instrukcí a dat z datové paměti. Pro zajištění generování validních instrukcí jsem zde využil *constraints* (omezení pro randomizaci) a funkci *post_randomize*, pomocí které na začátku simulace zajišťuji inicializaci registrů a poté ji ještě využívám pro úpravu

kódování instrukce jre. Kromě randomizace využívám tuto třídu i ke kontrole pokrytí simulovaných instrukcí.

4.2.2 My_sequence

Třída *my_sequence* dědí od třídy *uvm_sequence* a je realizovaná nejjednodušším možným způsobem – využil jsem zde pouze makro *'uvm_do*.

4.2.3 My_sequencer

Třída *my_sequencer* dědí od třídy *uvm_sequencer* a oproti bázové třídě neobsahuje nic navíc.

4.2.4 My_driver

Třída *my_driver* dědí od třídy *uvm_driver* a je odpovědná za buzení DUT – ve fázi build si driver vyzvedne referenci na vstupní rozhraní procesoru z konfigurační databáze UVM a ve fázi run je nekonečná smyčka, ve které driver vždy počká na náběžnou hranu hodin, poté si vyzvedne *my_item* od sekvenceru a následně přivede instrukci a „data z paměti“ na rozhraní procesoru.

4.2.5 My_monitor

Třída *my_monitor* dědí od třídy *uvm_monitor* a je odpovědná za sledování výstupního rozhraní DUT – ve fázi build si monitor vyzvedne referenci na výstupní rozhraní procesoru z konfigurační databáze UVM. Ve fázi run obsahuje monitor nekonečnou smyčku, ve které čeká na sestupnou hranu hodin, po které přečte hodnoty ze signálů: data do paměti, adresa do paměti a povolení zápisu do paměti. Následně ještě čeká na náběžnou hranu hodin, po které přečte hodnotu signálu PC. Na závěr tyto hodnoty uložené v objektu typu *my_item* odešle do scoreboardu na kontrolu.

4.2.6 My_in_monitor

Třída *my_in_monitor* dědí od třídy *uvm_monitor* a je odpovědná za sledování vstupního rozhraní DUT – ve fázi build si vyzvedne referenci na vstupní rozhraní procesoru z konfigurační databáze UVM. Ve fázi run monitor vstupů obsahuje nekonečnou smyčku, ve které vždy čeká na sestupnou hranu hodin a poté přečte hodnoty ze signálů: instrukce a data z datové paměti, tyto hodnoty poté uloží do objektu typu *my_item* a odešle do scoreboardu na další zpracování.

4.2.7 My_scoreboard

Třída *my_scoreboard* dědí od třídy *uvm_scoreboard* a je odpovědná za kontrolu činnosti procesoru. Vzhledem k tomu, že v této simulaci jsem nevyužil registrový model, musí si scoreboard pamatovat vlastní kopie registrů v DUT. Ve fázi run obsahuje scoreboard nekonečnou smyčku, ve které si vždy vyžádá novou transakci od obou monitorů, dekóduje instrukci, zavolá funkci pro vyhodnocení dané instrukce, navzorkuje objekt z monitoru vstupů pro kontrolu pokrytí a na závěr ještě ověří, zda již bylo dosaženo požadované úrovně pokrytí (pokud ano ukončí simulaci).

Scoreboard obsahuje funkce pro vyhodnocení každé instrukce. Vzhledem k tomu, že scoreboard zde nemá žádnou možnost přístupu k DUT, musí instrukce vyhodnocovat pouze na základě hodnot získaných od monitoru vstupů a vlastních kopií registrů. Stejně tak výsledky nemůže ověřovat přímo v registrech DUT, ale musí kontrolovat hodnoty snímané monitorem výstupů: adresa do paměti (výsledek ALU), data do paměti, povolení zápisu do paměti a PC.

4.2.8 My_env

Třída *my_env* dědí od třídy *uvm_env* a složí pro propojení komponentů simulačního prostředí. Ve fázi build tato třída vytváří instance všech ostatních komponent a ve fázi connect propojuje oba monitory se scoreboardem a driver se sekvencerem.

4.2.9 My_test

Třída *my_test* dědí od třídy *uvm_test* a je odpovědná za řízení simulace. Opět se jedná o velmi jednoduchou třídu – fáze run obsahuje pouze nekonečnou smyčku, ve které jsou vytvářeny nové sekvence a ty jsou odesílány do sekvenceru.

4.2.10 Uvm_top

Uvm_env je hlavní modul simulačního prostředí, ve kterém je instanciován procesor a jeho rozhraní, rozhraní jsou uložena do konfigurační databáze UVM a navíc zde dochází i ke generování hodin pro simulaci.

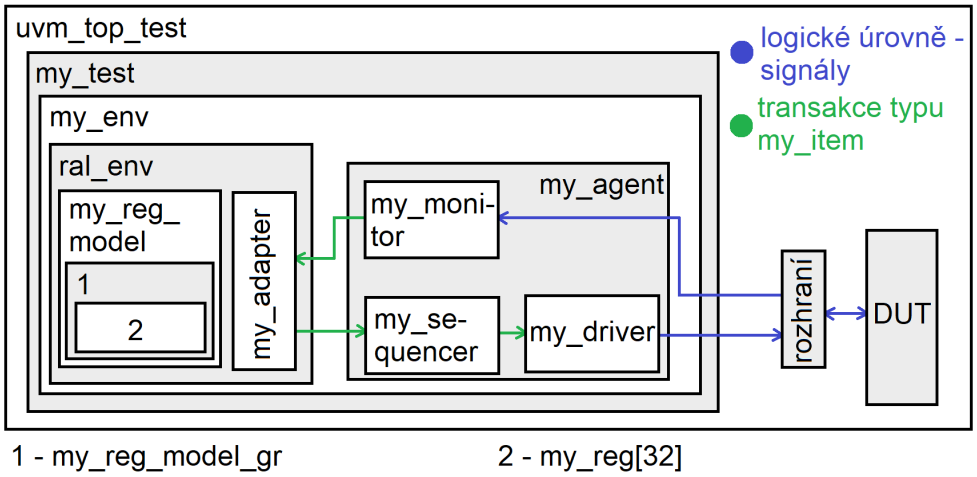
4.2.11 Výsledky simulace

V této variantě musí procesor pro dosažení požadovaného pokrytí 95% zpracovat 14 742 instrukcí. Simulační log můžete vidět na obrázku 4.2. Během této simulace jsem v procesoru odhalil chybu: do registru r0 bylo možné zapsat nenulovou hodnotu. Po opravě této chyby už vše funguje správně.

```
Scoreboard: coverage done -> finishing test.

Number of instructions: 14742
Number of errors: 0
```

Obrázek 4.2: Výstup na konzoli simulačního prostředí



Obrázek 4.3: Simulační prostředí pro verifikaci samotných registrů

4.3 Simulace samotných registrů pomocí RAL

Po úspěšném odsimulování jednocyklového procesoru bez registrového modelu byly na řadě simulace s registrovým modelem. Protože jsem si ale ze začátku nevěděl rady s tím, jak registrový model zprovoznit pro celý procesor (největší problém mi dělalo propojení registrů v registrovém modelu s registry v DUT), rozhodl jsem se nejprve si vyzkoušet registrový model pouze na modulu s registry. Strukturu navrženého prostředí můžete vidět na obrázku 4.3, není zde žádný scoreboard, protože správnou funkci ověřuji přímo v *my_test* tím, že si po každém zápisu do registru přečtu hodnotu, která byla zapsána.

Funkce jednotlivých komponent bude popsána v následujících podkapitolách (*uvm_sequence* a *uvm_sequencer* nepopisuji, protože jsem využil přímo třídy z knihovny UVM a nevytvářel jsem vlastní).

4.3.1 My_item

Třída *my_item* reprezentuje datovou položku transakce a dědí od třídy *uvm_sequence_item*. Položka obsahuje 4 datová pole, a to:

```
rand bit [31:0] addr;   // adresa registru
rand bit [31:0] rdata;  // přečtená data
rand bit [31:0] wdata;  // data k zápisu
rand bit write;         // typ transakce
```

Vzhledem k tomu, že cílem této simulace bylo pouze naučit se pracovat s registrovým modelem, neimplementoval jsem zde kontrolu pokrytí.

4.3.2 My_driver

Třída *my_driver* dědí od třídy *uvm_driver* a je odpovědná za buzení DUT – ve fázi build si driver vyzvedne referenci na vstupní rozhraní procesoru z konfigurační databáze UVM a ve fázi run je nekonečná smyčka, ve které driver vždy vyzvedne novou položku od sekvenceru a na jejím základě budí DUT.

4.3.3 My_monitor

Třída *my_monitor* dědí od třídy *uvm_monitor* a je odpovědná za sledování výstupního rozhraní DUT – ve fázi build si monitor vyzvedne referenci na výstupní rozhraní procesoru z konfigurační databáze UVM. Ve fázi run obsahuje monitor nekonečnou smyčku, ve které vždy čeká na náběžnou hranu hodin a poté přečte hodnoty na signálech rozhraní DUT.

4.3.4 My_agent

Třída *my_agent* dědí od třídy *uvm_agent* a je odpovědná za oddělení testbenche od DUT. Součástí agenta jsou monitor, driver a sekvencer – agent je ve fázi build inicializuje a ve fázi connect propojuje.

4.3.5 Ral_adapter

Třída *ral_adapter* dědí od třídy *uvm_reg_adapter* a je odpovědná za převod registrových transakcí na sběrnicové a naopak – za tímto účelem obsahuje funkce *reg2bus* a *bus2reg*.

4.3.6 My_reg

Třída *my_reg* dědí od třídy *uvm_reg* a reprezentuje 1 registr v registrovém modelu. *My_reg* obsahuje pouze jedno 32bitové datové pole nazvané data.

4.3.7 My_reg_model_gr

Třída *my_reg_model_gr* dědí od třídy *uvm_reg_block* a je odpovědná ze seskupení 32 registrů do jednoho bloku. Ve fázi build je tato třída odpovědná také za vytvoření všech 32 registrů, jejich přidání do adresní mapy a nastavení HDL cest k registrům v DUT:

```
default_map = create_map("my_map", 0, 128, UVM_LITTLE_ENDIAN);
for (int i = 0; i < 32; i++) begin
    my_reg_inst[i] = my_reg::type_id::create(
        $sformatf("my_reg_inst%d", i));
    my_reg_inst[i].build();
    my_reg_inst[i].configure(this);

    // HDL cesta k registru v rámci DUT
    my_reg_inst[i].add_hdl_path_slice($sformatf("rf[%d]", i),
        0, my_reg_inst[i].get_n_bits());
    default_map.add_reg(my_reg_inst[i], 4 * i, "RW");
end
// HDL cesta k dut v rámci testbenche
add_hdl_path("DUT");
```

4.3.8 My_reg_model

Třída *my_reg_model* dědí od třídy *uvm_reg_block* a reprezentuje registrový model. Ve fázi build tato třída inicializuje *my_reg_model_gr* a nastavuje HDL cestu k top modulu testbenche.

4.3.9 Ral_env

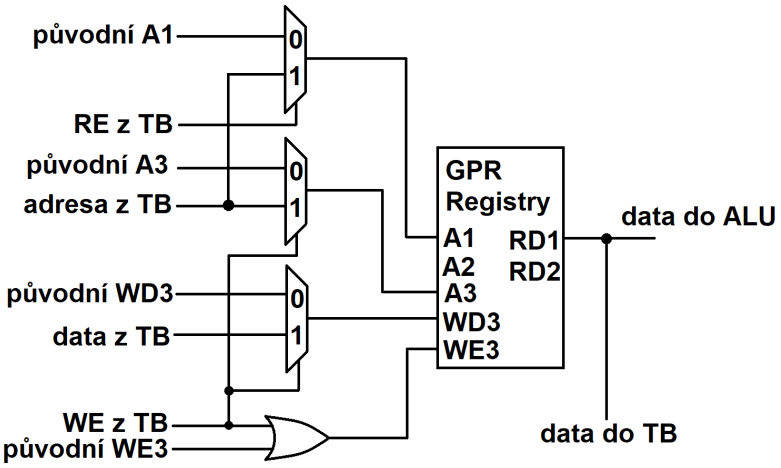
Třída *ral_env* dědí od třídy *uvm_env* a slouží ke správě komponent registrového modelu (*my_reg_model* a *ral_adapter*). Ve fázi build tato třída inicializuje registrový model a adaptér a poté ukládá referenci na registrový model do konfigurační databáze UVM.

4.3.10 My_env

Třída *my_env* dědí od třídy *uvm_env* a sdružuje *ral_env* a *my_agent*. Ve fázi build tyto komponenty inicializuje.

4.3.11 My_test

Třída *my_test* dědí od třídy *uvm_test* a definuje testovací scénář, v tomto případě je velmi jednoduchá, protože mým cílem v této simulaci nebylo ověření funkce DUT, ale pouze to, abych se naučil pracovat s registrovým modelem. Testovací scénář tedy sestává pouze ze tří zápisů do registru a následném přečtení zapsaných hodnot – nejprve jsem pro zápis i čtení využil frontdoor, poté frontdoor pro zápis a backdoor pro čtení a v posledním případě zapisuji pomocí backdoor přístupu a čtu pomocí frontdoor přístupu.



Obrázek 4.4: Úprava přístupu k registrům

4.3.12 Uvm_top_test

Modul *uvm_top_test* je hlavním modulem simulace – dochází v něm ke generování hodin, je zde instanciován modul DUT a jeho rozhraní a také je zde uložena reference na rozhraní DUT do konfigurační databáze UVM.

4.4 Simulace jednocyklového procesoru s využitím RAL

Poté, co jsem se v předchozí simulaci naučil pracovat s registrovým modelem, mohl jsem již začít pracovat na hlavní části mé diplomové práce: simulaci procesorů pomocí RAL. Jako první jsem vytvářel simulační prostředí pro jednocyklový procesor. Zde se hned na začátku projevila velká výhoda registrového modelu – vzhledem k tomu, že jsem měl již hotovou simulaci samotných registrů, mohl jsem velkou část simulačního prostředí (*my_reg, my_reg_model_gr, ral_env, ral_adapter, ral_agent a my_env*) zkopírovat a jen s drobnými úpravami ji přesunout do simulace jednocyklového procesoru.

4.4.1 Úpravy procesoru

Abych mohl realizovat simulaci zvoleného jednocyklového procesoru, musel jsem v něm udělat dvě změny. První změnou bylo přidání rozhraní, které umožňuje přímý přístup k registrům v DUT z testbenche – za tímto účelem jsem musel přidat také multiplexory, pomocí kterých se volí, zda budou do registru zapsána data z testbenche nebo z datové cesty procesoru. Zapojení přidaných multiplexorů pro GPR registry můžete vidět na obrázku 4.4.

Druhá úprava, kterou jsem musel udělat, spočívala v omezení velikosti pamětí procesoru – adresovatelný prostor procesoru je 2^{32} bytu, to bylo ale pro můj počítač, na kterém jsem simuloval, neúnosné (simulace by trvala příliš dlouho). Proto jsem zavedl parametr MEM_SIZE, který udává velikost datové i instrukční paměti (v 4bytových slovech – to znamená, že velikost paměti v bytech je MEM_SIZE * 4). V základní variantě je parametr MEM_SIZE nastaven na 1024.

4.4.2 Registrový model

U tohoto procesoru bylo nutné do registrového modelu zahrnout:

- PC: 1 32bitový registr
- GPR registry: 32 32bitových registrů
- instrukční paměť: MEM_SIZE * 4 bytů
- datovou paměť: MEM_SIZE * 4 bytů

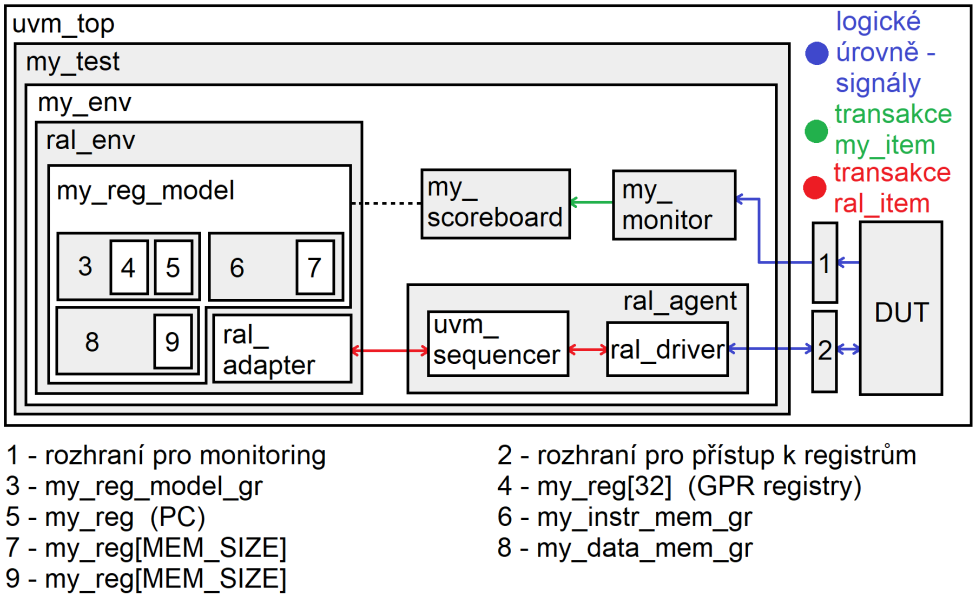
Všechny tyto položky jsem se rozhodl modelovat pomocí třídy *uvm_reg*. Pro instrukční a datovou paměť by sice bylo možné využít také *uvm_mem*, ta má ale oproti *uvm_reg* nevýhodu v tom, že nemá mirrored a desired hodnoty. To znamená že správnost hodnot v *uvm_mem* není možné ověřovat pomocí metod *mirror* a *predict*, ale je nutné hodnotu z paměti přečíst pomocí metody *read* a následně ji zkontrolovat ručně. Třídu *uvm_mem* je vhodné využívat pouze v případě, že je potřeba číst souvislé bloky paměti najednou (tam by u *uvm_reg* vznikala příliš velká režie) – to ale nebyl můj případ, já čtu z paměti vždy jen 4 byty.

4.4.3 Struktura simulačního prostředí

Strukturu simulačního prostředí pro verifikaci jednocyklového procesoru můžete vidět na obrázku 4.5. Opět se jedné o klasickou strukturu UVM testbenche, která je v tomto případě navíc doplněna o registrový model. Funkce jednotlivých komponent bude popsána v následujících kapitolách.

4.4.4 My_item

Třída *my_item* reprezentuje datovou položku transakce, dědí od třídy *uvm_sequence_item* a obsahuje pouze jedno datové pole a to *instruction*. *My_item* je využit při komunikaci monitoru se scoreboardem, kdy monitor předává scoreboardu informaci o aktuálně vykonávané instrukci v DUT. Tato třída je odpovědná i za kontrolu pokrytí zpracovaných instrukcí (ke vzorkování dochází ve scoreboardu).



Obrázek 4.5: Struktura simulačního prostředí pro jednocyklový procesor

4.4.5 My_instruction

Třída *my_instruction* dědí od třídy *uvm_sequence_item* a využívám ji jako pomocnou třídu pro generování náhodných instrukcí. Součástí této třídy jsou randomizační omezení (*constraint*), která zajišťují to, že bude vždy vygenerována validní instrukce. Dále tato třída slouží i pro kontrolu pokrytí vygenerovaných instrukcí (to je důvodem proč jsem vytvořil novou třídu místo využití třídy *my_item* – potřeboval jsem odděleně kontrolovat pokrytí vygenerovaných a zpracovaných instrukcí).

4.4.6 Ral_item

Třída *ral_item* reprezentuje datovou položku transakce, dědí od třídy *uvm_sequence_item* a obsahuje 4 datové položky:

```
rand bit [31:0] addr;    // adresa registru
rand bit [31:0] rdata;   // přečtená data
rand bit [31:0] wdata;   // data pro zápis
rand bit write;          // čtení/zápis (WE)
```

Tato třída slouží pro frontdoor přístup registrového modelu k registrům v DUT (komunikace registrového modelu a driveru).

4.4.7 Ral_driver

Třída *ral_driver* dědí od třídy *uvm_driver* a slouží pro frontdoor přístup registrového modelu k registrům v DUT. Driver přijímá transakce typu *ral_item* od sekvenceru a na jejich základě budí rozhraní *if_reg_acc*, které slouží k přímému přístupu k registrům v DUT. Aby driver mohl přistupovat k tomuto rozhraní, musí si na něj ve fázi build vyzvednout referenci z konfigurační databáze UVM.

4.4.8 Ral_agent

Třída *my_agent* dědí od třídy *uvm_agent* a slouží k zapouzdření driveru a sekvenceru (sekvencer zde nepopisuji, protože jsem opět využil *uvm_sequencer* přímo z knihovny UVM) a tím odstiňuje zbytek testbenche od implementace rozhraní DUT. Tato třída je ve fázi build odpovědná za inicializaci sekvenceru a driveru a ve fázi connect je odpovědná za jejich propojení.

4.4.9 Ral_adapter

Třída *ral_adapter* dědí od třídy *uvm_reg_adapter* a je odpovědná za převod registrových transakcí na sběrnicové a naopak – za tímto účelem obsahuje metody *reg2bus* a *bus2reg*.

4.4.10 My_reg

Třída *my_reg* dědí od třídy *uvm_reg* a je základním stavebním kamenem mého registrového modelu – reprezentuje jeden 32bitový registr. U této třídy jsem přepsal metodu *do_check* a to proto, že původní metoda v knihovně UVM nedovolovala upravit chybovou hlášku, která se zobrazuje na konzoli v případě, že se při kontrole hodnota v DUT neshoduje s mirrored hodnotou v registrovém modelu (kód metody je zkopírovaný ze zdrojových kódů knihovny UVM, pouze jsem upravil chybovou hlášku). Aby bylo možné v chybové hlášce zobrazit i aktuální prováděnou instrukci, přidal jsem do této třídy i členskou proměnnou *instr* a metodu *SetInstr* pomocí, které je možné do této proměnné uložit aktuální instrukci.

4.4.11 My_reg_model_gr

Třída *my_reg_model_gr* dědí od třídy *uvm_reg_block* a seskupuje 32 GPR registrů a PC. Ve fázi build je tato třída odpovědná, za inicializaci všech členských registrů, jejich začlenění do adresní mapy a nastavení HDL cest ke korespondujícím registrům v DUT.

4.4.12 My_data_mem_gr a My_instr_mem_gr

Třídy *my_data_mem_gr a my_instr_mem_gr* dědí od třídy *uvm_reg_block* a reprezentují registrové modely datové a instrukční paměti. Každá z těchto tříd obsahuje *MEM_SIZE* registrů typu *my_reg* a ve fázi build je odpovědná za inicializaci těchto registrů a nastavení HDL cesty k pamětem v DUT.

4.4.13 My_reg_model

Třída *my_reg_model* dědí od třídy *uvm_reg_block* a reprezentuje celkový registrový model procesoru – obsahuje objekty typu *my_reg_model_gr, my_data_mem_gr a my_instr_mem_gr*. Ve fázi build všechny tyto objekty inicializuje, a navíc ještě nastavuje HDL cestu k top modulu testbenche a seskupuje adresní mapy jednotlivých bloků do jedné nadřazené mapy (aby měl každý registr unikátní adresu).

4.4.14 Ral_env

Třída *ral_env* dědí od třídy *uvm_env* a slouží jako pomocné prostředí seskupující registrový model a jeho adaptér. Ve fázi build tyto dva komponenty inicializuje a následně ukládá referenci na registrový model do konfigurační databáze UVM.

4.4.15 My_monitor

Třída *my_monitor* dědí od třídy *uvm_monitor* a slouží ke čtení instrukce, kterou procesor aktuálně zpracovává. Ke čtení instrukce dochází vždy po sestupné hraně hodin, následně monitor tuto instrukci uloží do nové transakce typu *my_item* a čeká na další náběžnou hranu hodin (je třeba počkat, než budou do registrů uloženy výsledky této instrukce). Poté monitor odešle vytvořenou transakci do scoreboardu pro kontrolu. Aby mohl monitor číst aktuálně prováděné instrukce z DUT, musí si ve fázi build vyzvednout referenci na monitorovací rozhraní z konfigurační databáze UVM.

4.4.16 My_scoreboard

Třída *my_scoreboard* dědí od třídy *uvm_scoreboard* a je odpovědná za kontrolu správné funkce DUT. Vzhledem k tomu, že v této simulaci veškerá kontrola probíhá s využitím registrového modelu, musí si scoreboard na počátku fáze run z konfigurační databáze UVM vyzvednout referenci na registrový model. Po vyzvednutí této reference už fáze run obsahuje pouze volání metody *scoreboard_match*.

Metoda *scoreboard_match* obsahuje nekonečnou smyčku, ve které vždy čeká na přijetí nové instrukce od monitoru, následně tuto instrukci dekóduje a zavolá odpovídající metodu pro kontrolu (pro každý typ instrukce existuje

samostatná metoda, která provádí kontrolu: metody *EvalALUOP, EvalLw, EvalSw, EvalBeq, EvalJal, EvalAddu, EvalJr, EvalJ*). Po úspěšné kontrole výsledků instrukce, je provedeno vzorkování transakce přijaté od driveru pro kontrolu pokrytí. Na závěr je ještě kontrolováno, zda již nebylo dosaženo požadovaného pokrytí – pokud ano je vypsána statistika simulace (počet vykonaných instrukcí a počet nalezených chyb) a simulace je ukončena.

Metody pro kontrolu instrukcí fungují tak, že si nejprve pomocí registrového modelu načtou potřebné operandy, vypočítají správné výsledky instrukce a následně pomocí registrového modelu ověří správnost hodnot v DUT. Jako příklad uvádím kontrolu správného výsledku instrukce Addi:

```
// vytvoření popisu instrukce
instr = $sformatf("Addi (%b -> s%0d = s%0d + %0d)",
    item.instruction, item.instruction[20:16],
    item.instruction[25:21],
    $signed(item.instruction[15:0]));
// čekání dokud je cílový registr zaneprázdněn
wait (!m_reg_model.m_reg_model_gr.my_reg_inst[item.
    instruction[20:16]].is_busy);
// nastavení mirrored hodnoty cílového registru
m_reg_model.m_reg_model_gr.my_reg_inst[item.instruction[20:16]]
    .predict(res, -1, UVM_PREDICT_DIRECT, UVM_BACKDOOR);
// nastavení popisu instrukce (pro chybové hlášení)
m_reg_model.m_reg_model_gr.my_reg_inst[item.instruction[20:16]]
    .SetInstr(instr);
// kontrola hodnoty v DUT
m_reg_model.m_reg_model_gr.my_reg_inst[item.instruction[20:16]]
    .mirror(status, UVM_CHECK, UVM_BACKDOOR);
// zápis správné hodnoty do registru v DUT
m_reg_model.m_reg_model_gr.my_reg_inst[item.instruction[20:16]]
    .poke(status, res);
```

Pro kontrolu správných hodnot v DUT využívám vždy backdoor přístup a to proto, že nespotřebovává žádný simulační čas – v případě, že by byl využit frontdoor přístup, docházelo by zde ke zpoždění. Největší problém by představovala poslední funkce z příkladu výše, při frontdoor zápisu by se totiž muselo čekat na náběžnou hranu hodin. Při náběžné hraně hodin už ale musí docházet k zápisu výsledků následující instrukce.

4.4.17 My_env

Třída *my_env* dědí od třídy *uvm_env* a slouží k zapouzdření objektů *my_scoreboard, my_monitor, ral_env* a *ral_agent* – ve fázi build všechny tyto objekty inicializuje a ve fázi connect propojuje monitor se scoreboardem a adaptér se sekvencerem.

4.4.18 My_test

Třída *my_test* dědí od třídy *uvm_test* a reprezentuje testovací scénář. Ve fázi build tato třída inicializuje *my_env* a poté si vyzvedává z konfigurační databáze UVM referenci na rozhraní DUT (pouze kvůli hodinovému signálu). Ve fázi run si vyzvedává z konfigurační databáze UVM referenci na registrový model a následně začíná testování, to je rozděleno do 2 fází.

V první fázi testuji pouze to, zda správně fungují GPR registry – pomocí frontdoor přístupu jsou do registrů zapisovány náhodné hodnoty a následně jsou tyto hodnoty čteny pomocí frontdoor i backdoor přístupu a je kontrolováno, zda bylo přečtena stejná hodnota, která byla zapsána.

Ve druhé fázi pak testuji funkci celého procesoru – v nekonečné smyčce (simulaci ukončuje scoreboard) zde inicializuji instrukční i datovou paměť a následně čekám MEM_SIZE / 2 hodinových cyklů na zpracování instrukcí.

Kromě toho obsahuje *my_test* také metodu *DoSomethingBad*, volání této metody je v přiložených zdrojových kódech zakomentováno. Tato funkce je zde pro kontrolu, že je simulace schopna detekovat chyby v DUT – v náhodných intervalech injektuje do procesoru chyby (do PC, GPR registrů nebo do instrukční paměti). Injekce chyb spočívá v invertování jednoho náhodného bitu a pro přístup do DUT jsou využity metody registrového modelu.

4.4.19 Uvm_top

Modul *uvm_top* je hlavním modulem testbenche a slouží pro instanciaci DUT a jeho rozhraní, pro generování hodinového signálu a pro spuštění testu *my_test*. Tento modul také musí do konfigurační databáze UVM uložit reference na obě rozhraní DUT (rozhraní pro monitoring a rozhraní pro přímý přístup k registrům).

4.4.20 Výsledky simulace

V této variantě musí procesor pro dosažení požadovaného pokrytí 95% zpracovat 473 097 instrukcí (nárůst oproti simulaci bez registrového modelu je zde způsoben tím, že nyní byly do simulace zahrnuty i instrukční a datová paměť). Simulační logy můžete vidět na obrázcích 4.6 (bez injekce chyb) a 4.7 (s injekcí chyb – funkce *DoSomethingBad*). Na druhém zmíněném obrázku si můžete všimnout, že přesto, že bylo vygenerováno pouze 1 406 chyb, detekováno jich bylo 8 177. To je způsobeno tím, že injekce chyby do instrukční paměti způsobí více než jen jednu chybu. Na obrázku 4.8 můžete vidět příklad chybové hlášky při detekci špatné hodnoty v PC po instrukci Addi.

```
Number of instructions: 473097
Number of errors: 0

Errors generated:              0
```

Obrázek 4.6: Simulační log – bez injekce chyb

```
Number of instructions: 241680
Number of errors: 8177

Errors generated:           1406
```

Obrázek 4.7: Simulační log – s injekcí chyb

```
# UVM_ERROR ral_regs.sv(36) @ 731871000: reporter [RegModel] Error - instruction
  Addi (00100000101111010111011110001110 -> s29 = s5 + 30606)
# Wrong value in DUT - m_reg_model.m_reg_model_gr.PC
# Expected value:00000000000000000000000000000000110001101101100101100010011111
             got:00000000000000000000000000000000110001101101100101100010011011
```

Obrázek 4.8: Příklad chybové hlášky

4.5 Simulace zřetězeného procesoru

Finální částí mé práce byla aplikace všeho, co jsem se dosud naučil na jednodušších příkladech na simulaci zřetězeného procesoru. Vzhledem k tomu, že jsem měl již hotovou simulaci jednocyklového procesoru, se opět projevila výhoda znovupoužitelnosti, kterou přináší knihovna UVM – tentokrát jsem mohl využít v podstatě celé simulační prostředí pro jednocyklový procesor a musel jsem upravit pouze instrukční sadu, registrový model a vyhodnocování ve scoreboardu.

Kromě těchto úprav jsem navíc ještě přešel na využívání pouze backdoor přístupu, a to z toho důvodu, že zřetězený procesor obsahuje přes 70 registrů a využití frontdoor přístupu by znamenalo pro každý registr vyvést na rozhraní procesoru signály pro zápis z testbenche, čtení z testbenche, povolení zápisu z testbenche a dále pak přidání multiplexorů na tyto signály – to mi přišlo zbytečně složité, a proto jsem přešel na backdoor.

4.5.1 Úpravy procesoru

U zřetězeného procesoru jsem upravil instrukční a datovou paměť – v originále paměti načítaly data ze souboru. Stejně jako u předchozího procesoru bylo i zde třeba omezit velikost obou pamětí, proto jsem opět použil parametr *MEM_SIZE*, který udává velikost paměti v 4bytových slovech.

4.5.2 Registrový model

U tohoto procesoru bylo nutné do registrového modelu zahrnout:

- PC: 1 32bitový registr
- GPR registry: 32 32bitových registrů
- instrukční paměť: MEM_SIZE * 4 bytů
- datovou paměť: MEM_SIZE * 4 bytů
- registry oddělující stupně procesoru: 11 32bitových registrů a 27 1bitových registrů

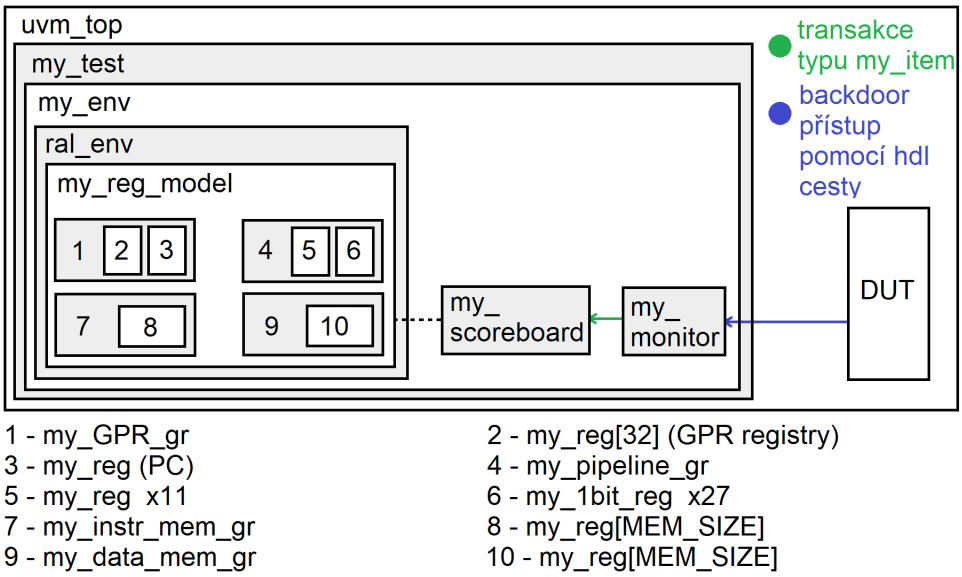
Všechny tyto položky opět modeluji pomocí tříd, které dědí od *uvm_reg*.

4.5.3 Struktura simulačního prostředí

Strukturu simulačního prostředí pro verifikaci zřetězeného procesoru můžete vidět na obrázku 4.9. Oproti simulaci jednocyklového procesoru zde chybí adaptér a agent – to proto, že nyní využívám pouze backdoor přístup. Navíc v registrovém modelu přibyl jeden blok s registry oddělujícími stupně procesoru. Vzhledem k tomu, že převážná část simulačního prostředí je stejná jako u jednocyklového procesoru, popíšu v následujících kapitolách pouze ty třídy, u kterých došlo k nějakým změnám.

4.5.4 My_reg

Třída *my_reg* dědí od třídy *uvm_reg* a stejně jako u simulace jednocyklového procesoru reprezentuje jeden 32bitový registr v registrové modelu. Oproti předchozí simulaci jsem do této třídy přidal členskou proměnnou *my_val* a metody *GetVal a SetVal*, které slouží k přístupu k této proměnné. Tuto proměnnou využívám u GPR registrů pro ukládání hodnot, které by v těchto registrech měly být při jednocyklovém zpracování – tím odpadá nutnost řešit během simulace forwarding (přeposílání výsledků mezi stupni procesoru).



Obrázek 4.9: Struktura simulačního prostředí pro zřetězený procesor

4.5.5 My_1bit_reg

Třída *my_1bit_reg* dědí od třídy *uvm_reg* a reprezentuje jeden 1bitový registr v registrové modelu (jednobitové registry jsou využity pro oddělení stupňů procesoru). Stejně jako ve třídě *my_reg* jsem i zde přepsal metodu *do_check* za účelem upravení chybové hlášky (viz kapitola My_reg u simulace jednocyklového procesoru).

4.5.6 My_pipeline_gr

Třída *my_pipeline_gr* dědí od třídy *uvm_reg_block* a seskupuje 11 32bitových (*my_reg*) a 27 1bitových (*my_1bit_reg*) registrů v registrovém modelu, které slouží pro oddělení stupňů zřetězeného procesoru. Ve fázi build tato třída inicializuje všechny své registry a nastavuje jim nastavuje HDL cesty ke korespondujícím registrům v DUT.

4.5.7 My_reg_model

Do této třídy oproti simulaci jednocyklového procesoru přibyl navíc jeden registrový blok a to *my_pipeline_gr*.

4.5.8 My_monitor

Třída *my_monitor* dědí od třídy *uvm_monitor* a stejně jako v simulaci jednocyklového procesoru slouží ke čtení aktuálně prováděné instrukce v DUT. Oproti předchozí simulaci pro to ale nevyužívá klasický přístup přes rozhraní procesoru, ale backdoor přístup pomocí HDL cesty k signálu v DUT.

4.5.9 My_scoreboard

Třída *my_scoreboard* dědí od třídy *uvm_scoreboard* a kontroluje správnost funkce DUT. Stejně jako v případě jednocyklového procesoru i v tomto scoreboardu je kontrolována pouze správnost výsledků instrukcí (není kontrolováno, zda jsou uloženy správné hodnoty v registrech oddělujících stupně procesoru). Ve fázi run si scoreboard vyzvedává z konfigurační databáze UVM reference na registrový model a na rozhraní DUT (pouze kvůli hodinovému signálu), následně je opět spuštěna funkce *scoreboard_match*.

Funkce *scoreboard_match* zůstala v podstatě stejná jako v předchozí simulaci, byly sem potřeba doplnit pouze proměnné *prev_target* a *ignore*. V proměnné *prev_target* je vždy uložen index GPR registru, do kterého bylo zapisováno v předchozí instrukci (nebo 32 v případě, že instrukce do registru nezapisovala) – tuto proměnnou využívám pro detekci hazardů, které je třeba řešit pozastavením (STALL). Proměnná *ignore* udává počet následujících instrukcí, které mají být ignorovány (nemají být vyhodnoceny) – tato proměnná slouží pro ošetření řídících hazardů po skokových instrukcích.

Metody pro vyhodnocení jednotlivých instrukcí také zůstaly převážně stejné jako u jednocyklového procesoru – opět dochází nejprve k načtení potřebných operandů pomocí registrového modelu a poté jsou vypočítány správné výsledky dané instrukce. Ke změně došlo v kontrole správnosti hodnot v DUT – vzhledem k tomu, že ve scoreboardu jsou instrukce vyhodnocovány najednou a v DUT jsou vyhodnocovány postupně v jednotlivých stupních, je třeba ve scoreboardu vždy počkat, než bude instrukce vyhodnocena i v DUT.

Scoreboard ale nemůže zastavit svoji činnost a čekat na DUT – v následujícím taktu už musí vyhodnocovat novou instrukci, proto jsem čekání vyřešil tak, že jsem do scoreboardu doplnil metody *CheckReg, CheckPC a CheckMem* (postupně kontrola GPR registrů, PC a datové paměti), které spouštím v novém simulačním vlákně. Tyto metody dostanou vždy jako parametr kolik hodinových cyklů je třeba počkat, adresu registru/paměti, očekávanou hodnotu a popis instrukce, která byla vykonávána (pro chybové hlášky). Samotná kontrola hodnot už probíhá stejně jako v předchozí simulaci – pomocí registrového modelu.

4.5.10 My_second_scoreboard

Třída *my_second_scoreboard* dědí od třídy *uvm_scoreboard* a také kontroluje správnost funkce DUT. Tento druhý scoreboard jsem navrhl proto, že

jsem měl příležitost si vyzkoušet, že s využitím předchozího scoreboardu je prakticky nemožné lokalizovat chyby v návrhu simulovaného procesoru – třída *my_scoreboard* dokáže poměrně rychle zkontrolovat, zda DUT funguje správně, ale v případě detekování chyby je obtížné tuto chybu lokalizovat. Při lokalizaci chyb (v procesoru byly 2 – viz kapitola Odhalené chyby v procesoru) jsem musel místo nástrojů UVM knihovny využít grafické průběhy jednotlivých signálů uvnitř DUT.

My_second_scoreboard jsem proto navrhl tak, aby kontroloval nejen výsledky instrukcí, ale i všechny registry oddělující stupně procesoru – tím se lokalizace chyby podstatně zjednodušila (chyba se neprojeví až ve výsledku, ale už v registrech za stupněm, ve kterém vznikla). *My_second_scoreboard* si ve fázi run stejně jako *uvm_scoreboard* vyzvedává reference na registrový model a na rozhraní DUT z konfigurační databáze UVM, poté se spouští metoda *scoreboard_match*.

Metoda *scoreboard_match* obsahuje nekonečnou smyčku, ve které vždy čeká na přijetí nové instrukce od monitoru – tuto instrukci následně pomocí metody *InstrToString* přeloží na textový popis pro případ potřeby výpisu chybového hlášení. Poté jsou již pomocí metod *EvalIF, EvalID, EvalEX, EvalMEM a EvalWB* kontrolovány výsledky v jednotlivých stupních (vyhodnocování je prováděno od stupně WB ke stupni IF – v „opačném pořadí“). Po vyhodnocení jednotlivých stupňů je ještě volána metoda *EvalPC*, která kontroluje správnost hodnot v PC. Na závěr je ještě provedeno vzorkování pro kontrolu pokrytí a kontrola, zda již nebylo dosaženo požadované úrovně pokrytí.

Metody pro kontrolu výsledků v jednotlivých stupních si vždy načtou desired hodnoty z registrů před daným stupněm (některé hodnoty je nutné přeposílat mezi jednotlivými metodami pomocí jejich parametrů, protože pro vyhodnocení aktuálního stupně jsou potřeba hodnoty z jiného stupně), následně vypočítají správné výsledky, které by měly být v registrech za daným stupněm a výsledky pomocí registrového modelu ověří. Po ověření jsou pro jistotu správné hodnoty do registrů ještě zapsány (pro případ, že by v registrech byla špatná hodnota).

Kromě metod pro kontrolu jednotlivých stupňů obsahuje *my_second_scoreboard* také metodu *UpdatePipeline*, která slouží k načtení hodnot všech registrů v DUT za účelem aktualizace desired hodnot – tato funkce je volána vždy na začátku testovací fáze, protože před tím bylo testování na chvíli pozastaveno za účelem nahrání nových hodnot do obou pamětí a GPR registrů. Vzhledem k tomu, že bylo testování pozastaveno, ale v DUT pokračovala činnost, desired hodnoty v registrovém modelu už nejsou správné.

4.5.11 Srovnání my_scoreboard a my_second_scoreboard

Jak třída *my_scoreboard* tak i *my_second_scoreboard* souží ke kontrole správnosti výsledků v DUT a obě tyto třídy dokáží všechny chyby v DUT dete-

kovat. Rozdíl spočívá v tom, že *my_scoreboard* kontroluje pouze správné výsledky instrukcí, které jsou ukládány do GPR registrů, popř. do datové paměti. *My_second_scoreboard* oproti tomu kontroluje i správnost hodnot ve všech registrech oddělujících stupně procesoru – tím je dosažena mnohem jednodušší lokalizace chyb v procesoru, ale za cenu delšího simulačního času (*my_second_scoreboard* ověří správnost stejného množství instrukcí jako *my_scoreboard* za přibližně pětinásobek času).

4.5.12 Odhalené chyby v procesoru

Během této simulace se mi podařilo ve zřetězeném procesoru odhalit 2 chyby:

- u modulu s GPR registry byl překlep na řádku 279 – místo proměnné g10 zde byla pouze hodnota 10 a v důsledku toho nebylo možné číst z GPR registru 27 (přečtená hodnota byla vždy 0)
- v aritmeticko-logické jednotce byl překlep na řádku 40 – v 27. řádu ALU byl použit 26. bit druhého operandu

Kromě těchto chyb, jsem odhalil ještě jednu méně závažnou chybu, která sice nezpůsobovala výpočet špatných hodnot, ale zbytečně zpomalovala procesor. Tato chyba spočívala v tom, že činnost procesoru byla pozastavována (STALL) i v případě, že ve stupni ID byla instrukce j nebo jr – ani jedna z těchto instrukcí ale nevyužívá operand RT a instrukce j nevyužívá dokonce ani operand RS.

4.5.13 Výsledky simulace

Při využití třídy *my_scoreboard* musí procesor pro dosažení požadovaného pokrytí 95% zpracovat 33 433 instrukcí – simulační log bez injekce chyb (metoda *DoSomethingBad* třídy *my_test)* můžete vidět na obrázku 4.10 a simulační log s injekcí chyb na obrázku 4.11. Můžete si všimnout, že pro dosažení požadovaného pokrytí je v případě zřetězeného procesoru třeba zpracovat mnohem méně instrukcí než u jednocyklového – to je způsobeno tím, že zřetězený procesor má menší instrukční sadu.

Při využití třídy *my_second_scoreboard* musí procesor pro dosažení požadovaného pokrytí 95% zpracovat 33 448 instrukcí. Simulační logy můžete opět vidět na obrázcích 4.12 (bez injekce chyb) a 4.13 (s injekcí chyb).

```
Number of instructions: 33433
Number of errors: 0

Errors generated:           0
```

Obrázek 4.10: Simulační log – *my_scoreboard*

```
Number of instructions: 28075
Number of errors: 412

Errors generated:         298
```

Obrázek 4.11: Simulační log – *my_scoreboard* s injekcí chyb

```
Number of instructions: 33448
Number of errors: 0

Errors generated:           0
```

Obrázek 4.12: Simulační log – *my_second_scoreboard*

```
Number of instructions: 26352
Number of errors: 493

Errors generated:         178
```

Obrázek 4.13: Simulační log – *my_second_scoreboard* s injekcí chyb

Závěr

Hlavním cílem této práce bylo navržení verifikačních prostředí pro jednocyklový a zřetězený procesor s využitím knihovny UVM a jejího registrového modelu. Dalšími cíli pak bylo seznámení se s registrovým modelem a sepsání stručného nápomocného textu k využívání RAL vrstvy pro začínající vývojáře. Všech těchto cílů se mi podařilo dosáhnout.

V první části této práce (Kapitola Knihovna UVM) jsem vytvořil text, který čtenáře seznamuje s knihovnou UVM a popisuje funkci všech jejích základních komponent. Poté následuje popis registrového modelu (RAL), jeho komponent a metod pro přístup k registrům.

Kapitola Analýza popisuje zvolené procesory, pro které jsem navrhoval simulační prostředí pro jejich verifikaci – u obou procesorů je zde popsána jejich struktura a instrukční sada, u zřetězeného procesoru jsou zde navíc popsány také hazardy, ke kterým v procesoru dochází a způsob jejich řešení. Kapitolu Analýza uzavírá stručný popis využitého vývojového prostředí QuestaSim.

Kapitola Realizace se zabývá popisem samotných navržených simulačních prostředí a všech jejich komponent. U obou procesorů jsou zde uvedeny také chyby v návrhu, které se mi během verifikace podařilo odhalit.

Pomocí navržených verifikačních prostředí se mi v obou procesorech podařilo odhalit chyby – v jednocyklovém procesoru bylo možné zapisovat do registru r0 (který by měl být vždy nulový) a ve zřetězeném procesoru byla chyba v ALU, v modulu s GPR registry a v jednotce, která řídí pozastavování činnosti procesoru (STALL).

Všechny zdrojové kódy napsané v rámci této práce jsou přístupné na git stránce projektu [29] a v budoucnu mohou být využity pro verifikaci dalších procesorů, či v rámci výuky.

Literatura

[1] Harrison, J. *Formal Verification at Intel* [online prezentace]. [cit. 2022-03-13]. Dostupné z: https://www.cl.cam.ac.uk/~jrh13/slides/nijmegen-21jun02/slides.pdf

[2] UVM Tutorial *chipverify* [online]. [cit. 2022-01-19]. Dostupné z: https://www.chipverify.com/uvm/uvm-tutorial

[3] UVM TestBench architecture *Verification Guide* [online]. [cit. 2022-01-20]. Dostupné z: https://verificationguide.com/uvm/uvm-testbench-architecture/

[4] UVM Sequence item *Verification Guide* [online]. [cit. 2022-01-21]. Dostupné z: https://verificationguide.com/uvm/uvm-sequence-item/

[5] UVM Sequence [uvm_sequence] *chipverify* [online]. [cit. 2022-01-20]. Dostupné z: https://www.chipverify.com/uvm/uvm-sequence

[6] UVM Sequencer [uvm_sequencer] *chipverify* [online]. [cit. 2022-01-20]. Dostupné z: https://www.chipverify.com/uvm/uvm-sequencer

[7] UVM Driver [uvm_driver] *chipverify* [online]. [cit. 2022-01-20]. Dostupné z: https://www.chipverify.com/uvm/uvm-driver

[8] UVM Monitor [uvm_monitor] *chipverify* [online]. [cit. 2022-01-20]. Dostupné z: https://www.chipverify.com/uvm/uvm-monitor

[9] UVM Agent [uvm_agent] *chipverify* [online]. [cit. 2022-01-20]. Dostupné z: https://www.chipverify.com/uvm/uvm-agent

[10] UVM Scoreboard [uvm_scoreboard] *chipverify* [online]. [cit. 2022-01-20]. Dostupné z: https://www.chipverify.com/uvm/uvm-scoreboard

[11] UVM Environment [uvm_env] *chipverify* [online]. [cit. 2022-01-21]. Dostupné z: https://www.chipverify.com/uvm/uvm-environment

[12] UVM Test [uvm_test] *chipverify* [online]. [cit. 2022-01-21]. Dostupné z: https://www.chipverify.com/uvm/uvm-test

[13] UVM Phases [uvm_test] *Verification Guide* [online]. [cit. 2022-03-12]. Dostupné z: https://verificationguide.com/uvm/uvm-phases/

[14] Introduction to UVM RAL *Verification Guide* [online]. [cit. 2022-01-22]. Dostupné z: https://verificationguide.com/uvm-ral/introduction-to-uvm-ral/

[15] uvm_reg_field *Verification Academy* [online]. [cit. 2022-01-27]. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files/reg/uvm_reg_field-svh.html

[16] uvm_reg *Verification Academy* [online]. [cit. 2022-01-23]. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_reg-svh.html

[17] uvm_mem *Verification Academy* [online]. [cit. 2022-01-27]. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_mem-svh.html

[18] uvm_reg_block *Verification Academy* [online]. [cit. 2022-01-27]. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_reg_block-svh.html

[19] uvm_reg_map *Verification Academy* [online]. [cit. 2022-01-27]. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1b/html/files/reg/uvm_reg_map-svh.html

[20] Registers/Adapter *Verification Academy* [online]. [cit. 2022-01-27]. Dostupné z: https://verificationacademy.com/cookbook/registers/adapter

[21] UVM Register Backdoor Access *chipverify* [online]. [cit. 2022-01-22]. Dostupné z: https://www.chipverify.com/uvm/uvm-register-backdoor-access

[22] ŠTĚPANOVSKÝ, Michal *Návrh jednocyklového RISC procesoru* [online prezentace]. 2019 [cit. 2022-01-30]. Dostupné z: https://courses.fit.cvut.cz/BI-APS/@B191/media/lectures/BI-APS-Prednaska04-SingleCycleCPU.pdf

[23] ŠTĚPANOVSKÝ, Michal *Semestrální projekt č.1: Jednocyklový procesor* [online]. 2019 [cit. 2022-01-30]. Dostupné z: https://courses.fit.cvut.cz/BI-APS/@B191/tutorials/05/semester_project_cz.html

[24] Pipelined MIPS Processor in Verilog (Part-3) *FPGA 4 Student* [online]. 2017 [cit. 2022-01-31]. Dostupné z: https://www.fpga4student.com/2017/06/32-bit-pipelined-mips-processor-in-verilog-3.html

[25] Pipelined MIPS Processor in Verilog (Part-1) *FPGA 4 Student* [online]. 2017 [cit. 2022-01-31]. Dostupné z: https://www.fpga4student.com/2017/06/32-bit-pipelined-mips-processor-in-verilog-1.html

[26] HENNESSY, J. L. a D. A. PATTERSON. *Computer Architecture: A Quantitative Approach.*Waltham: Morgan Kaufman/Elsevier, 2017. ISBN 978-01-281-1905-1

[27] Siemens EDA (Formerly Mentor Graphics) *Semiconductor Engineering* [online]. 2020 [cit. 2022-02-7]. Dostupné z: https://semiengineering.com/entities/mentor-a-siemens-business/

[28] Questa Advanced Verification *SIEMENS* [online]. [cit. 2022-02-7]. Dostupné z: https://eda.sw.siemens.com/en-US/ic/questa/

[29] JÍLEK, V. *SCE - Simulace procesoru* [online]. Dostupné z: https://gitlab.fit.cvut.cz/jilekvoj/sce---simulace-procesoru

Příloha **A**

Seznam použitých zkratek

ALU Arithmetic-Logic Unit

DUT Device Under Test

EX Execute (3. stupeň zřetězeného procesoru)

GPR General Purpose Register

ID Instruction Decode (2. stupeň zřetězeného procesoru)

IF Instruction Fetch (1. stupeň zřetězeného procesoru)

LSB Least Significant Bit

MEM Memory (4. stupeň zřetězeného procesoru)

PC Program Counter

RAL Register Abstraction Layer

RAW Read After Write

TB Testbench

TLM Transaction-level Modeling

UVM Universal Verification Methodology

WB Write Back (5. stupeň zřetězeného procesoru)

WAR Write After Read

WAW Write After Write

WE Write Enable

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `Pipeline_Procesor` ..... zřetězený procesor
    - `Pipeline_RAL` ..... simulace zřetězeného procesoru
    - `Registers_RAL` ..... simulace samotných registrů
    - `Single_Cycle_Processor` ..... jednocyklový procesor
    - `Single_Cycle_no_RAL` simulace jednocylkového procesoru bez RAL
    - `Single_Cycle_RAL` ..... simulace jednocylkového procesoru s RAL
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `RAL_tips.pdf` .... tipy k používání registrového modelu pro začínající vývojáře

Příloha C

Ukázkový kód jednoduchého registrového modelu

Registr:

```
class my_reg extends uvm_reg;
    // registrace do factory
    `uvm_object_utils(my_reg)

    // datové pole registru
    rand uvm_reg_field data;

    // konstruktor
    function new (string name = "my_reg");
        super.new(name, 32, UVM_NO_COVERAGE);
    endfunction

    // build fáze
    function void build;
        data = uvm_reg_field::type_id::create("data");
        // konfigurace datového pole
        data.configure(this, 32, 0, "RW", 0, 0, 0, 1, 0);
    endfunction
endclass
```

Blok registrů:

```
class my_reg_model extends uvm_reg_block;
    // registrace do factory
    `uvm_object_utils(my_reg_model_gr)

    // registr
    rand my_reg m_reg;

    // konstruktor
    function new (string name = "");
        super.new(name, build_coverage(UVM_NO_COVERAGE));
    endfunction

    // build fáze
    function void build;
        // vytvoření adresní mapy
        default_map = create_map("my_map", 0, 4,
                                 UVM_LITTLE_ENDIAN);
        m_reg = my_reg::type_id::create("m_reg");
        m_reg.build();
        m_reg.configure(this);
        // přidání registru do adresní mapy
        default_map.add_reg(m_reg, 0, "RW");
        // nastavení HDL cesty k registru v DUT
        m_reg.add_hdl_path_slice("cesta", 0,
                                 m_reg.get_n_bits());
        add_hdl_path("dut_top");
    endfunction
endclass
```

Adaptér:

```
class ral_adapter extends uvm_reg_adapter;
    // registrace do factory
    `uvm_object_utils (ral_adapter)

    //konstruktor
    function new (string name = "ral_adapter");
        super.new (name);
    endfunction

    // převod RAL transakce na sběrnicovou transakci
    function uvm_sequence_item reg2bus(
                                const ref uvm_reg_bus_op rw);
        bus_item item = bus_item::type_id::create("item");
        item.write = (rw.kind == UVM_WRITE);
        item.addr  = rw.addr;
        if (item.write)  item.wdata = rw.data;
        if (!item.write) item.rdata = rw.data;
        return item;
    endfunction

    // převod sběrnicové transakce na RAL transakci
    function void bus2reg(uvm_sequence_item bus_it,
                          ref uvm_reg_bus_op rw);
        bus_item item;
        assert( $cast(item, bus_it) )
            else `uvm_fatal("", "Adapter ERROR!")
        rw.kind = item.write ? UVM_WRITE : UVM_READ;
        rw.addr = item.addr;
        rw.data = item.rdata;
        rw.status = UVM_IS_OK;
    endfunction
endclass
```