FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Pokročilé metody simulace v jazyce SystemVerilog
Student:	Bc. Miroslav Kallus
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 (interface, clocking block, randomizace, asserty, coverage, ...).
- Seznamte se s konstrukcemi knihovny UVM (object, component, komunikační prostředky, TLM, RAL).
- Otestujte podporu výše zmíněných konstrukcí ve vývojových a simulačních nástrojích.
- Vytvořte nápomocný text k používaní SystemVerilogu a UVM knihovny včetně ukázkových zdrojových kódů.
- Vytvořte vzorové úlohy pro předmět Simulace číslicových obvodů.

Elektronicky schválil/a doc. Ing. Hana Kubátová, CSc. dne 10. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Pokročilé metody simulace v jazyce SystemVerilog

Bc. Miroslav Kallus

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

6. května 2021

Poděkování

Rád bych poděkoval vedoucímu práce, panu Ing. Martinu Kohlíkovi, za jeho cenné rady, které byly pro tuto práci velkým přínosem. Dále bych chtěl poděkovat své rodině a přátelům za poskytnutí opory.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů 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 předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, 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 způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). 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 způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2021 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Miroslav Kallus. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Kallus, Miroslav. *Pokročilé metody simulace v jazyce SystemVerilog.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato práce se zabývá prací s nástroji pro simulaci číslicových obvodů. V první části se práce zabývá seznámením se s konstrukcemi jazyka SystemVerilog a knihovny UVM. Dále práce otestuje nástroje pro vytváření testbenchů, jako jsou textové editory a IDE, a simulátory. Nakonec vznikne pomocný text k používání SystemVerilogu a knihovny UVM včetně ukázkových zdrojových kódů a zároveň vzniknou vzorové úlohy pro předmět Simulace číslicových obvodů.

Klíčová slova SystemVerilog, DVT Eclipse, Questa, simulace

Abstract

This thesis deals with digital circuit simulation tools. In research part I am going to learn basic constructs of SystemVerilog language and UVM library. Then, I am going to test testbench design tools, like text editors and IDEs, and simulation tools. In the end, I'll create helpful text for using SystemVerilog and the UVM library including examples of source code. There will also be model projects for the Digital Circuit Simulation course.

Keywords SystemVerilog, DVT Eclipse, Questa, simulation

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešním světě se každý z nás setkává s desítkami, ne-li stovkami, přístrojů, kterým bezmezně věříme a očekáváme jejich bezchybný provoz. A právě za tímto pocitem stojí lidé, kteří stráví často i roky vyvíjením takového přístroje a vychytáváním každé jeho chybičky. Jedním z kroků takového návrhu je i verifikace. Verifikace ovšem není prospěšná jen pro uživatele těchto přístrojů. Pokud se ve výrobku objeví chyba, projeví se to na zisku, který z prodeje výrobce má. A čím dříve výrobce chybu v procesu vývoje objeví, tím nižší jsou náklady na opravu takové chyby a případné ztráty z prodeje.

Výsledek této práce je určen pro lidi, kteří by se rádi stali jedním z těchto hledačů chyb a živili se verifikací číslicových obvodů. Ze zkušenosti vím, že naučit se něco v tomto oboru může být složité, hlavně pokud nejste dostatečně tvrdohlaví a vzdáváte se po prvním neúspěchu, podobně jako já. Právě takovým jedincům by měl dopomoci výsledek této práce k jednodušším začátkům s jazykem SystemVerilog.

Na začátku se práce věnuje úvodu do Verilogu a základním konstrukcím SystemVerilogu. Zde se pokusí osvětlit úlohu těchto konstrukcí a jak je tvořit. Další kapitola se věnuje knihovně UVM a jejímu použití. Dále se zabývá nástroji pro práci s konstrukcemi z předchozích kapitol. Na konci práce obsahuje proces vytváření pomocného textu ke konstrukcím jazyka SystemVerilog a vzorových úloh do předmětu Simulace číslicových návrhů.

Po této kapitole jsou vytyčeny cíle práce. Pak následuje kapitola zabývající se úvodem do jazyka Verilog.

Kapitola 1

Cíle práce

Cílem této práce je nastudovat základní funkce konstrukcí jazyka SystemVerilog a knihovny UVM a následně vyzkoušet podporu nástrojů těchto konstrukcí. Dále vzniknou nápomocné texty k jednotlivým konstrukcím a k nim i ukázkové zdrojové kódy s příklady použití těchto konstrukcí. Na závěr se vytvoří příklady vzorových úloh pro předmět Simulace číslicových obvodů.

Práce je rozdělena do tří částí, seznámení se s jazykem SystemVerilog a knihovnou UVM, otestování podpory těchto konstrukcí nástroji a vytváření textů a úloh.

Cílem první části je osvojení si znalostí o jednotlivých konstrukcích jazyka SystemVerilog a knihovnou UVM a jejich funkcích, naučit se tyto konstrukce využívat při verifikaci číslicových obvodů.

Cílem části otestování podpory je seznámení s nástroji podporujícími jazyk SystemVerilog a knihovnu UVM, tyto nástroje se naučit používat a následně zhodnotit moji zkušenost s nimi. Dále na základě tohoto hodnocení budou vybrány nástroje které se použijí pro vytvoření vzorových kódů a úloh.

Cílem poslední části je vytvoření pomocných textů pro konstrukce jazyka SystemVerilog a knihovny UVM a vzorových kódů pro tyto texty. Dalším cílem je vznik vzorových úloh pro předmět Simulace číslicových obvodů.

Kapitola 2

Úvod do Verilogu

Pro pochopení této práce je potřeba základní znalost jazyka Verilog. Proto se tato kapitola zaměří na porovnání dvou kódů, jednoho psaného ve VHDL a druhého ve Verilogu. Pokud čtenář tyto znalosti má, může kapitolu přeskočit. Na příkladu klopného obvodu budou ukázány ekvivalentní konstrukce mezi těmito jazyky.

K porovnání budou využity zdrojové kódy 2.1 a 2.2. Na první pohled jsou kódy zcela rozdílné, ale přesto mají spoustu společného.

VHDL rozděluje návrh na dvě části, *entity*, kde se nachází portlist[1] návrhu a *architecture*, která řeší vnitřní chování návrhu. Verilog obsahuje portlist v hlavičce bloku *module* a vnitřní chování je uvnitř téhož bloku. U portlistu ve VHDL je důležité ke každému signálu uvést datový typ, nejčastěji buď *std_logic* nebo *std_logic_vector* vektory. Verilogovým ekvivalentem *std_logic* jsou *reg* a *wire*. *Reg* je možné chápat jako paměťový prvek který si pamatuje hodnotu až do následujícího zápisu. *Wire* se chová jako drát a používá se pro výstupy anebo při propojování bloků. Vektory pro tyto typy vzniknou pomocí hranatých závorek, podobně jako třeba pole v jazyce C.

Chování návrhu se ve VHDL řeší uvnitř bloku *architecture*. Zde se mohou objevit deklarace komponent, proměnných a signálů. Po deklaracích začíná blok ve kterém se objevují jednotlivé procesy, vytváření instancí a podobně. Verilog při popisu chování návrhu není tak striktní. Deklarace, vytváření instancí a procesy zde mohou být libovolně smíchané, i když to není ideální pro čitelnost kódu. Při vytváření procesu se využívá *always* blok který se chová téměř totožně jako *process* blok u VHDL. Snad jediný rozdíl se skrývá v takzvaném *sensitivity listu* [2]. Ve Verilogu se dá reagovat na specifikovanou hranu, zatímco VHDL tuto možnost nemá a musí správnost hrany kontrolovat uvnitř procesu. V případě, že je do sensitivity listu potřeba napsat všechny signály, ekvivalentem k VHDL zápisu pomocí *all* je ve Verilogu použití *.

[1] Portlist je seznam vstupů a výstupů návrhu.

[2] Sensitivity list obsahuje seznam signálů jejichž změna provede blok.

```
module DFF_8 (input clk, reset, input [7:0] D, output reg [7:0] Q);
  always @ (posedge clk)
  begin
    if (rst == 1'b1) begin
      Q <= 8'h00;
    else
      Q <= D;
    end
  end
endmodule
```

Zdrojový kód 2.1: Klopný obvod D napsaný pomocí Verilogu

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity DFF_8 is
  port(
    clk, reset : in std_logic;
    D : in std_logic_vector (7 downto 0);
    Q : out std_logic_vector (7 downto 0);
  );
end DFF_8;

architecture Behavioral of DFF_8 is
-- place for declaration
begin
  process(clk)
  begin
    if(rising_edge(clk)) then
      if(reset = '1') then
        Q <= x"00";
      else
        Q <= D;
      end if;
    end if;
  end process;
end Behavioral;
```

Zdrojový kód 2.2: Klopný obvod D napsaný pomocí VHDL

Kapitola 3

Seznámení se SystemVerilogem

Podle standardu IEEE 1800-2017 [1] je SystemVerilog jazyk určený k specifikaci, verifikaci a návrh hardwaru. Tento standard byl vytvořen sdružením společností Accellera a jeho základem bylo rozšíření HDL jazyka Verilog. Následující podkapitoly se budou věnovat základním konstrukcím tohoto jazyka a změnám oproti Verilogu.

3.1 Datové typy

S vytvořením SystemVerilogu se oproti Verilogu o dost zjednodušilo používání typů. Nejviditelnější změnou je přidání datového typu *logic*. Ten se dá použít na místo Verilogových typů *reg* nebo *wire*, čímž odpadá rozhodování, který z těchto dvou je vlastně správný.

Dále se v SystemVerilogu objevil výčtový typ *enum*. Díky tomu je jednodušší například práce se stavy automatu. Ve Verilogu byla jediná možnost jak reprezentovat stavy pomocí binárních hodnot, případně vytvořením parametrů s těmito hodnotami. Na následujícím ukázkovém kódu je ukázáno jak moc *enum* zjednodušší reprezentaci stavů automatu pro semafor:

```
// SystemVerilog
 enum {RED, YELLOW, GREEN} traffic_light;

//Verilog
parameter RED = 2'b00;
parameter YELLOW = 2'b01;
parameter GREEN = 2'b10;
```

Rozšíření počtu stavů takového automatu v SystemVerilogu je na první pohled mnohem jednodušší.

Dalším přídavkem SystemVerilogu byl datový typ String a metody pracující s tímto typem. Mnoho metod funguje stejně jako v C, jako například

len(), *compare()* nebo *atoi()*. K tomu má SystemVerilog definované operátory podobné práci se signály, například spojování a replikování řetězců pomocí složených závorek nebo porovnávání pomocí rovnítek.

S příchodem SystemVerilogu se také objevilo několik velice užitečných operátorů. V první řadě přibyly operátory pro inkrementaci a dekrementaci čím se zjednodušil například zápis hlavičky for cyklu. Dalším užitečnými operátory jsou porovnání s takzvanými zástupnými znaky, neboli "wildcard". Díky ním je možné při porovnávání řešit jen důležité pozice signálu, ostatní pozice se nastaví buď na hodnotu x nebo z.

3.2 Třídy

Asi jednou z nejvýznamějších novinek týkajících se typů, je přidání tříd. Třída je datový typ definovaný uživatelem, obsahuje data a funkce a tasky pracující s těmito daty. Třídy umožňují použití principu objektově orientovaného programování, v případě SystemVerilogu to jsou zapouzdření, dědičnost a polymorfismus. Stejně jako ostatní konstrukce v SystemVerilogu, i třídy se dají parametrizovat.

Nejdůležitější funkcí ve třídě je konstruktor, který by se měl starat o inicializaci při vytvoření instance. V SystemVerilogu se tato funkce značí názvem *new*. Stejně jako v jiných jazycích může konstruktor použít vstupních parametrů. Konstruktor může být použit k vytvoření mělké kopie instance, ale protože SystemVerilogové třídy neznají takzvané "copy constructory", pro hlubokou kopii instance je potřeba vytvořit si vlastní funkci.

Data uvnitř třídy jsou defaultně viditelná zvenčí. Pokud to není přípustné, je potřeba prvek označit klíčovým slovem *local*, pak k němu budou mít přístup pouze členské funkce. V případě že se z třídy bude dědit třída další, je potřeba na místo *local* využít *protected* které způsobí že zděděné objekty tato data uvidí a budou s nimi moct pracovat. To jde vidět na příkladu 3.1, kde třída B dědí od třídy A. Ve třídě A je členská proměnná *m_addr* definována jako protected, takže s ní mohou pracovat metody z B. Pokud by ovšem byla třída která by dědila od třídy B, proměnnou *m_data* by použít nemohla, protože je definovaná jako local.

Dědění samotné probíhá obdobně jako u dalších jazyků podporujících OOP. Za zmínku stojí klíčové slovo *super* které umožňuje volat funkce svého předka. Nejčastěji se s tímto lze setkat v konstruktoru, kde se pomocí super volá konstruktor předka, jak je vidět i na příkladu 3.1.

Třídy v SystemVerilogu také mohou být abstraktní. Toho lze docílit použitím slova *virtual*. Stejným způsobem můžeme deklarovat virtuální funkci. Z takové třídy nelze vytvořit instanci, je potřeba vytvořit další třídu oddědenou od této, která bude obsahovat další funkcionality a podobně. Tohoto se dá využít například pro polymorfismus, kdy abstraktní třída bude obsahovat základní

metody a data a třídy děděné od této přidají data a metody specifická pro svou funkci, případně dodefinují virtuální metody z abstraktní třídy.

```
class B extends A;

        local logic [7:0] m_data;
        function new (logic [7:0] data, logic [31:0] addr);
                super.new(addr);
                m_data = data;
        endfunction

endclass

virtual class A;
        protected logic [31:0] m_addr;
        function new (logic [31:0] addr);
                m_addr = addr;
        endfunction
endclass
```

Zdrojový kód 3.1: Příklad dědění tříd.

3.3 Interface

Interface, neboli rozhraní, slouží k spojení více signálů do jednoho bloku, a jak už název napovídá, spojující dvě či více komponent. Výhodou použití této konstrukce je obrovská redukce portů v portlistu komponenty a také mnohem nižší potřeba tento portlist udržovat. Veškeré změny se totiž dějí uvnitř samotného rozhraní.

Deklarace rozhraní je velmi podobná deklaraci klasického modulu, jak je vidět na příkladu níže:

```
interface mult_if #(WIDTH = 4)(input clk);

logic [WIDTH-1:0] a,b;
logic [2*WIDTH-1:0] s1, s2;

modport multIN (input a, b, output s2);
modport testIN (output a, b, input s2);

endinterface
```

V hlavičce se nachází portlist a výčet vstupních parametrů. V samotném těle je seznam signálů které chceme využívat pro propojení komponent. Dalšími nepovinnými částmi jsou modporty, časovací bloky anebo tasky a funkce.

Modport v rozhraní určuje ke kterým signálům bude mít komponenta přístup a směr[3] kterým data z rozhraní do komponenty jdou. Vzhledem k tomu, že výchozí nastavení signálů v rozhraní je inout může se stát že na signálu nastane konflikt když na něj zapíše více modulů najednou. Pokud není vhodné, aby na stejný signál mohlo zapisovat více modulů, předejde se tomu nastavením směru uvnitř modportu. Zároveň se dá pomocí modportu i vybrat signály, ke kterým má mít modul přístup. K signálům neuvedeným v modportu pak modul nebude mít vůbec přístup.

Uvnitř rozhraní se taktéž dají vytvářet funkce a tasky, které pak mohou volat připojené moduly. To jde udělat několika způsoby. Prvním z nich je definovat task přímo v rozhraní, pak modul může task bez problému zavolat, aniž by se v modulu něco změnilo. Druhá možnost je uvnitř modportu. Zde se musí k seznamu signálu přidat i jméno tasku s klíčovým slovem *import*. Tělo tasku je pak znovu umístěno uvnitř rozhraní a modul připojený přes modport může tento task využít. Poslední možností je definovat task uvnitř modulu. Tady se musí využít klíčového slova *extern*, v případě používání modportu *export*. Tělo tasku je pak umístěno přímo do těla modulu.

3.4 Randomizace

Randomizací je myšleno generování náhodných hodnot pro signály a zároveň vytváření omezení pro toto generování. Randomizace se využívá hlavně během testování návrhu, omezení se tedy využívají k tomu, aby vygenerovaná data dávala smysl pro daný test.

Obvykle se k generování testovacích dat využívají třídy. Na příkladu níže je vidět jednoduchá třída obsahující dvě proměnné označené k randomizaci. Následuje krátký kus kódu, ve kterém se vytvoří instance třídy a následně se pomocí metody *randomize()* vygenerují náhodné hodnoty pro obě proměnné.

```
class my_rand;
...
randc int m_x;
rand int m_y;
...
endclass

...
my_rand item;
 item = new();
item.randomize()
...
```

K označení proměnných k randomizaci se používají klíčová slova *rand* a

[3]Možnosti jsou input (dovnitř), output (ven) a inout (obousměrný přenos)

randc. *Rand* znamená vygenerování hodnoty na základě uniformního rozdělení. *Randc* se chová maličko jinak, zde se vytvoří permutace obsahující všechny možné hodnoty. Následně se pak při volání metody *randomize()* postupně vrací hodnoty z této permutace. Ve chvíli, kdy metoda vrátí poslední prvek této permutace, vygeneruje se permutace nová. Toto však může být jak výkonostně tak i paměťově náročné, proto by se modifikátor *randc* neměl používat pro signály s velkým počtem bitů.

Generování náhodných dat můžeme usměrňovat pomocí omezení. Ta se tvoří pomocí klíčového slova *constraint*. Důležitá konstrukce při vytváření omezen *solve ... before ...*, která určuje pořadí generování. Pokud mezi omezeními žádná taková podmínka není, generují se všechny hodnoty najednou dokud, nejsou splněna všechna omezení. V případě, že hodnota jednoho signálu závisí na hodnotě druhého signálu a vytvoříme omezení s konstrukcí *solve ... before ...*, můžeme tím zrychlit generování hodnot. Jinak se dá jako omezení používat různá porovnávání, logické podmínky, testování, jestli je hodnota v rozsahu a podobně. Dají se také použít operátor *inside*, *dist* a *unique*. *Inside* zajistí, že hodnota bude vybrána z určené množiny, *dist* zajistí totéž, ale dodrží váhy určené u hodnot v množině a *unique* zajistí, že seznam signálů bude mít různé hodnoty.

V případě práce s polem se dá využít operátoru *foreach*. Ten aplikuje pravidlo na každý prvek pole, je tedy možnost generovat pro každý prvek pole hodnotu s jiným omezením, zavislým například na indexu.

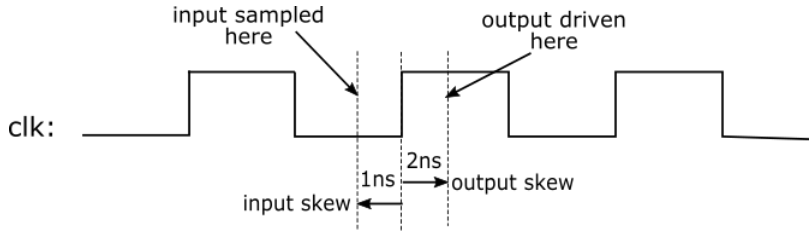
Omezení se při běhu programu dají vypínat a zapínat pomocí metody *constraint_mode()*. Ta pokud dostane jako parametr 1, omezení zapne, a pokud dostane 0 omezení vypne. Pokud parametr nedostane, metoda vrátí stav omezení.

3.5 Clocking bloky

Díky clocking blokům se dají nastavit předstihy, případně přesahy, pro jednotlivé signály. To umožňuje tuto část přesunout z testbenche do vlastního bloku, čímž se testbench zpřehlední a zároveň bude toto časování jednodušší na údržbu.

V hlavičce clocking bloku musí být jméno a takzvaný clocking event neboli event, vůči kterému se ostatní signály budou synchronizovat. Event v tomto případě je hrana signálu, náběžná, závěrná nebo obě. Uvnitř bloku se pak nachází definice směrů signálů, výchozí předstih pro vstupní signály a výchozí přesah pro výstupní signály. Na obrázku 3.1 je znázorněno kdy se signál mění při nastavení spoždění oproti eventu a kdy se vzorkuje hodnota signálu při nastavení předstihu.

Dále se definují předstihy a přesahy pro jednotlivé signály, pokud jsou odlišné od výchozího nastavení. Krom posunu o časovou konstantu lze nastavit



Obrázek 3.1: Ukázka předstihu a přesahu u signálu [2]

že se signál bude měnit při hraně synchronizujícího signálu. Zde se dá využít možností *negedge* pro závěrnou hranu nebo *posedge* pro náběžnou.

Pokud nějaký takový clocking block v testu existuje, dá se využít několika dalších funkcí. Pokud je potřeba, aby program počkal na event clocking bloku, lze toho docílit pomocí konstrukce @{*jmeno_blocku*}. To je ekvivalentní použití konstrukce @{*signal*} kde signal je synchronizační signál z hlavičky bloku. Druhá funkce je zpoždění o počet eventu. To se děje pomocí operátoru ## x, kde x může být číslo nebo výraz. V tu chvíli test čeká, než proběhne x eventů a potom dál pokračuje.

3.6 Pokrytí

Coverage, neboli pokrytí, umožňuje zjistit, jestli je verifikace dostatečná. Na základě nastavených podmínek a vstupních testovacích vektorů SystemVerilog sám hlídá a počítá procentuální pokrytí. Tyto podmínky se vytvářejí pomocí konstrukce *covergroup*. Ta v hlavičce obsahuje seznam eventů, při kterých se hodnoty signálů snímají. V těle pak najdeme jednotlivé podmínky, takzvané *coverpoints*. Uvnitř coverpointů se nalezají takzvané *biny* neboli intervaly hodnot proměnné ke které se coverpoint váže. Tyto biny se buď vygenerují samy tak, že pro každou hodnotu vznikne jeden, anebo si je uživatel vytváří sám. Pokud biny nejsou defaultně vygenerovány, pak je potřeba říct, kdy hodnota do binu patří. Pomocí binů lze také hlídat jestli, signál provedl přechod z jedné hodnoty do jiné. Příkladem využití může být třeba hlídání využití hrany v konečném automatu. Následující úryvek kódu například ukazuje vytvoření

covergroup pro kontrolu změn světel na semaforu:

```
enum {RED, YELLOW, GREEN} traffic_light;
...
covergroup cg @(light)
{
  cp1 : coverpoint light
  {
    bins b [] = (RED => YELLOW), (YELLOW => GREEN),
                (GREEN => YELLOW), (YELLOW => RED);
  }
}
```

Tato vzniklá covergroup kontroluje, jestli nastanou všechny přechody, které nastat mají. V případě že nastanou, pokrytí tohoto bodu je 100%.

Při vytváření binů lze využít několika zjednodušení. S pomocí klauzule *with* můžeme definovat bin pomocí funkce. Případně lze využít klíčového slova *wildcard*, díky němuž můžeme definovat interval binu ignorováním nějakého bitu signálu.

Pro vytváření binů pro hlídání přechodů existuje několik konstrukcí. Pomocí konstrukce $x[*n{:}m]$ lze hledat n až m výskytů hodnoty x za sebou. Konstrukce $x[{=}n]$ hledá n výskytů x, tentokrát ale můžou mezi hodnotami x být libovolné množství jiných hodnot.

Na konec stojí za to zmínit konstrukci *cross*, která dokáže vytvořit kartézský součin binů dvou či více coverpointů. V těle konsturkce *cross* lze biny vzniklé součinem seskupit pomocí konstrukce *binsof(x)*, kterou lze kombinovat s *intersect* $\{y\}$. První z nich seskupí všechny biny, které mají společný bin x, při použití průřezu se navíc bin x omezí na množinu y. Vzhledem k tomu, že při kartézském součinu může vzniknout spoustu nepotřebných binů, dají se tyto biny pomocí klíčového slova *ignore_bins* ignorovat.

3.7 Aserce

SystemVerilog také přidal možnost aserce, což jsou podmínky, které musí být během simulace splněny. Toho lze využívat jak pro jednoduchou kontrolu rovnosti hodnot signálů, čemuž se říká okamžitá aserce a využívá se konstrukce *assert*, tak pro kontrolu průběhů a změn signálů během nějakého období, té se říká sekvenční aserce a využívá konstrukce *assert property*. Aserce funguje velice podobně jako v ostatních programovacích jazycích, základní konstrukce je velice podobná konstrukci if-else. Zároveň s tím se dají využívat systémové

funkce pro chybová hlášení lišící se podle závažnosti nesplnění aserce, ta jsou následující:

- *$fatal* - simulace by měla být ukončena,
- *$error* - simulace by měla upozornit na chybu, ale pokračovat dál,
- *$warning* - simulace vypíše varování že aserce, nebyla splněna
- *$info* - signalizuje, že aserce nebyla splněna, bez dalších následků.

Pokud u aserce chybí větev else, simulátor automaticky hlásí chybu pomocí funkce *$error*.

V případě použití sekvenční aserce lze využít bloku *property*. Ten slouží k zabalení podmínky pro assert, kam se následně napíše jen jméno property bloku. Tím se třeba stejná podmínka může opakovat na více místech a stále se udržovat jen na jednom místě.

Další zajímavou použitelnou konstrukcí je implikace, které byly přiřazeny operátory |–> a |=>. Implikace funguje tak, jak se očekává, v případě že je levá strana vyhodnocena jako pravdivá, testuje se pravá strana. Jinak je celá podmínka vyhodnocena jako pravdivá. Rozdílem mezi operátory je ten, že u druhého operátoru se testuje pravá strana až při dalším tiku řídícího signálu.

Často využívané funkce u asercí jsou také *$rose*, *$fell* a *$stable*, které vrací informaci o tom jestli se na signálu objevila náběžná hrana, závěrná hrana anebo se hodnota nezměnila.

Kapitola 4

Seznámení s knihovnou UVM

Podle [3] je UVM metodologie pro funkční verifikaci hardwaru, primárně za pomoci simulací. K tomuto účelu vznikla knihovna UVM, což je soubor bloků psaných v jazyce SystemVerilog, určených k sestavení znovupoužitelných modulárních testbenchů. Metodologie UVM byla vytvořena, stejně jako SystemVerilog, sdružením společností Accellera a jejím vzorem byla zejména metodologie OVM. Knihovna UVM umožňuje rozdělit vytváření jednotlivých bloků mezi více lidí. Jednotlivým stavebním blokům knihovny UVM jsou věnovány následující podkapitoly.

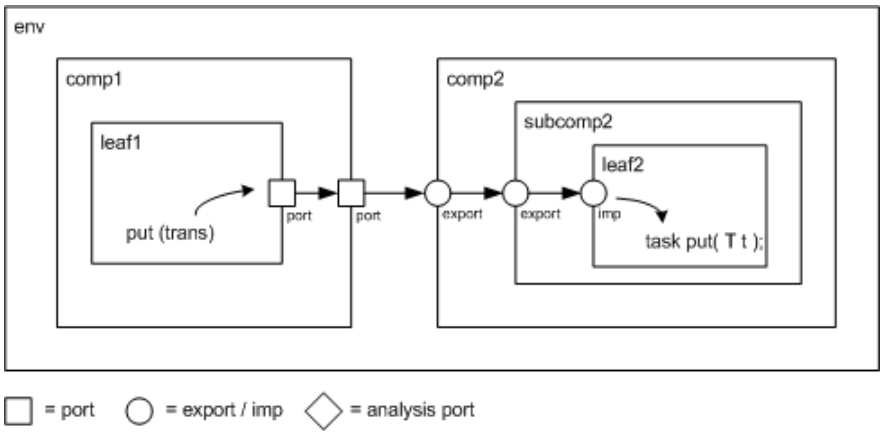
4.1 TLM

Než se práce bude moct věnovat blokům UVM, je potřeba nastínit co TLM znamená a hlavně jak ho používat s UVM. Zkratka TLM znamená Transaction Level Modeling. Podle [4] je to modelovací styl pro vytváření vysoce abstraktních modelů systému. TLM modeluje výměnu informací mezi komponentami jako objekt, narozdíl od RTL kde se používají přímo signály.

UVM obsahuje knihovnu s prvky používanými k propojování komponent pomocí TLM. Obsahem této knihovny jsou rozhraní pracující s transakcemi a různé druhy portů a soketů. Porty a sokety se pak dělí na blokující a neblokující, podle způsobu příjímání respektive odesílání dat. Porty se pak ještě dělí na porty, exporty a impy. Dále se v knihovně nalézá implementace fronty a analysis portu.

Při propojení a komunikaci pomocí portů je důležité buď, aby odesílatel implementoval task *put* nebo příjemce implementoval task *get*. Na základě toho, kde je task implementovaný, se rozhoduje, který port se použije u příjemce a který u odesílatele. Komponenta, ve které je task implementovaný obsahuje imp, druhá komponenta je pak připojena přes port. Export se využije v případě komponenty, která task implementovaný nemá, ale obsahuje jinou komponentu, která implementaci obsahuje. Toto je vyobrazeno na obrázku 4.1. Použití analysis portu je podobné–komponenta která, ho využívá, funkci

write jen volá, zatímco komponenty připojené pomocí analysis impů každá implementuje svou funkci *write*. Dalším možným způsobem propojení je pomocí FIFO. Zde jsou obě komponenty propojené pomocí portů a neimplementují žádný task, jen volají *put* nebo *get* podle toho, jestli do fronty vkládají nebo z ní vybírají. Je to proto, že FIFO oba tasky implementované už má.



Obrázek 4.1: Použití portů, exportů a impů podle toho kde je implementován task *put*. [5]

V případě použití neblokujících portů se místo tasku *put* využívají tasky *can_put* a *try_put*, podobně i pro task *get*. Task *can_put* slouží ke kontrole, jestli je přijímající strana připravena přijmout transakci. Toto se obvykle dělá v cyklu, dokud task nedostane kladnou odpověď. V tu chvíli se použije task *try_put*, který transakci odešle.

Podrobnější informace k TLM portům, soketům a použití se nachází na [4] a [6], odkud tato podkapitola čerpala.

4.2 Továrna

Továrna v UVM slouží k vytváření objektů a komponent. Pro vytváření objektů a komponent se nevyužívá volání funkce *new*, ale statická funkce *create*. Důvodem je následná možnost záměny objektů mezi sebou bez změny kódu.

UVM továrna obsahuje tabulku všech objektů a komponent, které se do ní zaregistrují. To se děje pomocí takzvaného *utility makra*. Ta se liší podle toho, jestli se jedná o třídu děděnou od *uvm_object* nebo *uvm_component* a vypadají následovně:

```
`uvm_object_utils(my_obj)
`uvm_component_utils(my_comp)
```

Argumentem pro makra je jméno třídy. Pro vytvoření se pak používá funkce *create*. Továrna zároveň obsahuje override list, ve kterém se nachází seznam

pravidel pro nahrazování tříd mezi sebou. Pravidlo může být vytvořeno pouze pokud třídy mají mezi sebou vztah třída a podtřída. Pak lze pravidlo přidat následující funkcí:

```
set_type_override_by_type(orig_type, override_type);
```

kde první argument je jméno typu který bude změněn a druhý argument je jméno typu, za který bude změněn. Přepisování instancí funguje podobně, jen s jinou funkcí, další funkce s vysvětlením chování se nalézají zde [7].

V případě zaregistrování všech objektů a komponent pomocí maker lze také zapisovat do override listu pomocí argumentů simulátoru při spuštění simulace.

4.3 Sequence item

Objekt *uvm_sequence_item*, neboli transakce, obsahuje data, která jsou potřeba k vytvoření stimulu pro testovaný obvod, zkráceně DUT. Tento objekt by měl být zaregistrován u továrny, obsahovat konstruktor a makra pro automatické vytvoření metod pro výpis obsahu instance, kopírování instance a podobně. Tato makra nemusí být použita, pokud si programátor tyto funkce vytvoří sám. Tyto funkce jsou *do_copy*, *do_clone*, *do_print*, *do_pack* a *do_unpack*. Tyto funkce mají jasně dané prototypy, které je potřeba dodržet.

4.3.1 Field makra

Pokud si programátor sám funkce napsat nechce, musí továrně říct, jak jednotlivé členské proměnné reprezentovat a které operace s nimi půjde provádět. Toho docílí pomocí field maker, kterými pokryje všechny členské proměnné. Na příkladu 4.1 je ukázáno, jakým způsobem se field makro používá. Makro přijímá dva argumenty, třetí argument určující typ členské proměnné, je schován ve jméně použitého makra. Prvním argumentem v závorkách je jméno členské proměnné, druhý označuje, které funkce s danou proměnnou nejdou provádět, případně u čísel je možné nastavit číselná soustava, ve které se proměnná má vypisovat. Pro druhý argument lze použít více než jednu možnost, možnosti se spojují bitovým orem.

```
`uvm_object_utils_begin(Packet_Macro)
  `uvm_field_int(m_data, UVM_DEFAULT)
  `uvm_field_int(m_addr, UVM_DEFAULT)
`uvm_object_utils_end
```

Zdrojový kód 4.1: Příklad použití field makra.

4.4 Sequence, Sequencer

Sekvence se dědí od objektu *uvm_sequence* a slouží ke generování položek s náhodnými hodnotami. Sekvence má jako svůj parametr typ generované transakce, což je jméno objektu vytvořeného jako sequence_item. Stejně jako ostatní objekty je potřeba sekvenci registrovat u továrny, k registrování se používá objektové makro. Sekvence musí obsahovat svůj konstruktor a task *body*. Uvnitř konstruktoru se obvykle jen zavolá konstruktor předka za pomoci klíčového slova *super*. Task body slouží k vygenerování položky a odeslání driveru v následujících šesti krocích:

1. Vytvoření položky
2. Čekání na požadavek
3. Randomizace položky
4. Odeslání položky
5. Čekání na potvrzení o dokončení
6. Vytvoření položky

Poslední dva kroky jsou nepovinné. Protože jsou tyto kroky téměř vždy stejné, obsahuje knihovna makra, která zjednodušují vytváření tohoto tasku. Tato makra nahradí některé či všechny funkce provádějící kroky výše. V následujícím seznamu jsou uvedeny příklady některých maker, další makra lze najít na [8].

- `` `uvm_do(item) `` - provedou se všech šest kroků,
- `` `uvm_create(item) `` - provede pouze vytvoření,
- `` `uvm_send(item) `` - přeskočí vytvoření a randomizaci,
- `` `uvm_rand_send(item) `` - přeskočí vytvoření a provede zbylé kroky,
- `` `uvm_rand_send_with(item,constraints) `` - přeskočí vytvoření a k randomizaci použije omezení z druhého parametru.

Sekvencer se dědí od objektu *uvm_sequencer*, jeho účelem je generovat transakce pomocí sekvence a odesílat je driveru. Parametrem sekvenceru je znovu jméno generované transakce. U továrny se registruje svým vlastním makrem `` `uvm_sequencer_utils(xxx) ``. Pokud k sekvenceru není připojeno více než jedna sekvence, obsahuje pouze svůj konstruktor. O propojení se stará driver.

4.5 Driver

Driver dědí od komponenty *uvm_driver* a slouží k převedení transakce na jednotlivé signály, které posílá do testovaného návrhu. Driver obsahuje logiku jakým způsobem data DUTu předat. I driver je parametrizovaný, jeho parametrem je typ transakce. U továrny se registruje pomocí makra pro komponenty.

Driver je propojený se sekvencerem pomocí TLM portů a s DUTem skrz interface. Vytvoření interface je na programátorovi, uvnitř driveru je pak virtualní interface, do kterého se uloží handle[4] pomocí konfigurační databáze. Toho se docílí následujícím řádkem v kódu:

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

Přiřazení probíhá ve funkci *build_phase*. Port pro připojení k sekvenceru se nedefinuje, už je nadefinován v nadtřídě.

Samotná funkce driveru je v tasku *run_phase*. Zde si v nekonečné smyčce driver vyžádá transakci od sekvenceru. Toho může docílit buď pomocí blokující funkce *get_next_item* nebo neblokující funkce *try_next_item*. Pak následuje kód, který z data z transakce přepošle do testovaného designu. Tato část je práce programátora a bude se měnit v závislosti na testovaném designu. Když je řízení signálu do DUTu dokončené, driver potvrdí dokončení sekvenceru, to má na starosti funkce *item_done*. Ta by se měla volat vždy v případě blokující komunikace, v případě neblokující jen při úspěšném přijmutí transakce. Poslední, nepovinný, krok je shromáždění odpovědi od DUTu a poslání do sekvenceru.

4.6 Monitor

Monitor dědí od komponenty *uvm_monitor*. Jeho úkolem je odposlouchávání signálů testovaného designu a vytváření transakce z těchto odposlechnutých hodnot. Tyto hodnoty pak poskytuje dalším komponentám testbenche. Monitor navíc může obsahovat i kontroly pokrytí testovacích vektorů. U továrny se také registruje pomocí makra pro komponenty.

Monitor může být připojen k testovanému designu pomocí interface kterým spolu komunikují driver a DUT. Do této komunikace vůbec nezasahuje. Další možností je připojit monitor na čistě výstupní porty testovaného designu, taktéž pomocí interface. V obou případech monitor musí obsahovat handle na tento interface. Dále monitor obsahuje TLM analysis port, kterým přeposílá vytvořené transakce dalším komponentám, například *scoreboardu*. Stejně jako u driveru, task *build_phase* slouží k přiřazení interface do handle pomocí konfigurační databáze. Zároveň se zde vytváří instance analysis portu.

[4] Handle je ekvivalent pointerům z jiných jazyků

Nejdůležitější část monitoru je task *run_phase*, kde se odehrává všechna činnost. V tomto tasku se vzorkují signály DUTu, to se děje při eventu který značí, že jsou data validní. Dále se tato data zabalí do transakce, ta je následně odeslána přes analysis port pomocí funkce *write*. Uvnitř monitoru se nesleduje správnost výsledků, pouze pokrytí vstupů, případně správnost komunikace.

4.7 Agent

Agent dědí od komponenty *uvm_agent*. Uvnitř agenta se nalézají dříve popsané komponenty monitor, driver a sekvencer, pokud je agent nastavený jako aktivní. Pokud je nastavený jako pasivní, uvnitř je pouze komponenta monitor a agent negeneruje pro testovaný design testovací transakce. U továrny se registruje pomocí makra pro komponenty.

Task *build_phase* uvnitř agenta obsahuje instance vnitřních komponent. To, jestli se bude jednat o aktivního nebo pasivního agenta, určuje parametr *is_active*, jeho hodnota se dá zjistit pomocí funkce *get_is_active*. Defaultně je agent aktivní. Pokud je potřeba nastavit agenta jako pasivního, docílí se toho následující částí kódu:

```
uvm_config_int::set(this, "path_to_agent", "is_act", UVM_PASSIVE);
```

Uvnitř tasku *connect_phase* se pak jen propojí driver se sekvencerem, a to jen v případě, že se bude jednat o aktivního agenta.

4.8 Scoreboard

Scoreboard dědí od komponenty *uvm_scoreboard*. Úloha scoreboardu je porovnávat výsledky testovaného designu proti očekávaným hodnotám. Uvnitř se obvykle nachází referenční model testovaného designu, který očekávané hodnoty vytváří. Scoreboard je propojen s monitorem, odkud mu přichází výstupy DUTu. Registrace u továrny se provádí pomocí makra pro komponenty.

Pro spojení s monitorem musí scoreboard obsahovat analysis imp port. Dále musí obsahovat implementaci funkce *write*, kterou používá monitor pro odeslání transakce. Další, pro programátora jednodušší, možností je použít *TLM analysis FIFO*. Její výhodou je, že není potřeba vytářet funkci *write*, ta už je implementována ve frontě. Při vytváření fronty lze nastavit, aby mohla obsahovat maximálně jednu transakci. Tím se oba způsoby propojení budou na venek chovat totožně.

V případě využití nevyužití fronty se kontrola správnosti výsledků provádí v implementaci funkce *write*. Port pro propojení je také potřeba vytvořit uvnitř funkce *build_phase*. Pokud se fronta použije, kontrola správnosti se přesouvá do funkce *run_phase*. Předtím, než začne kontrola, se musí transakce vytáhnout z fronty pomocí funkce *get*. V tomto případě je potřeba vytvořit frontu v konstruktoru scoreboardu.

4.9 Environment

Environment dědí od komponenty *uvm_env*. Environment slouží jako kontejner pro všechny ostatní komponenty, případně i další environmenty. Environment také může obsahovat propojení komponent pomocí TLM portů. Lze proto využít možností hierarchického přístupu, tím se dají propojení shromáždit na jednom místě. I environment se u továrny registruje pomocí makra pro komponenty. Vytváření komponent se provádí v funkci *build_phase*, propojování probíhá ve funkci *connect_phase*.

4.10 Test

Test dědí od komponenty *uvm_test*. Test obsahuje testovací scénáře a pravidla nahrazování tříd pro továrnu. Z komponent obsahuje environment a sekvenci, takže vlastně celý testbench. Stejně jako ostatní komponenty UVM, i tato se u továrny registruje pomocí makra pro komponenty.

Během funkce *build_phase* se v testu vytvoří instance pro sekvenci a environment. Případně se sem vkládají i nastavení testbenche. Pokud se tak nestane v top modulu, může se zde objevit vložení interface do konfigurační databáze. To se dělá stejnou funkcí jako je ukázáno v podkapitole 4.11.

Funkce *run_phase* pak má za úkol takzvaně spustit sekvenci na sekvenceru. K tomu slouží funkce *start* sekvence, která jako parametr přijímá sekvencer, kterému má posílat vygenerované transakce.

Na závěr se ještě vyplatí zmínit funkci *print_topology*. Ta při spustění simulace vypíše topologii testbenche, je tedy možné si zkontrolovat, že se vše propojilo a vytvořilo tak, jak mělo. Funkce se volá pomocí globální proměnné *uvm_top*. Volání této funkce je dobré použít ve funkci *end_of_elaboration_phase*, která se volá těsně před tím než začne samotná simulace, ale až po vytvoření všech komponent a jejich propojení. Více informací o fázích a jejich pořadí lze najít na [9].

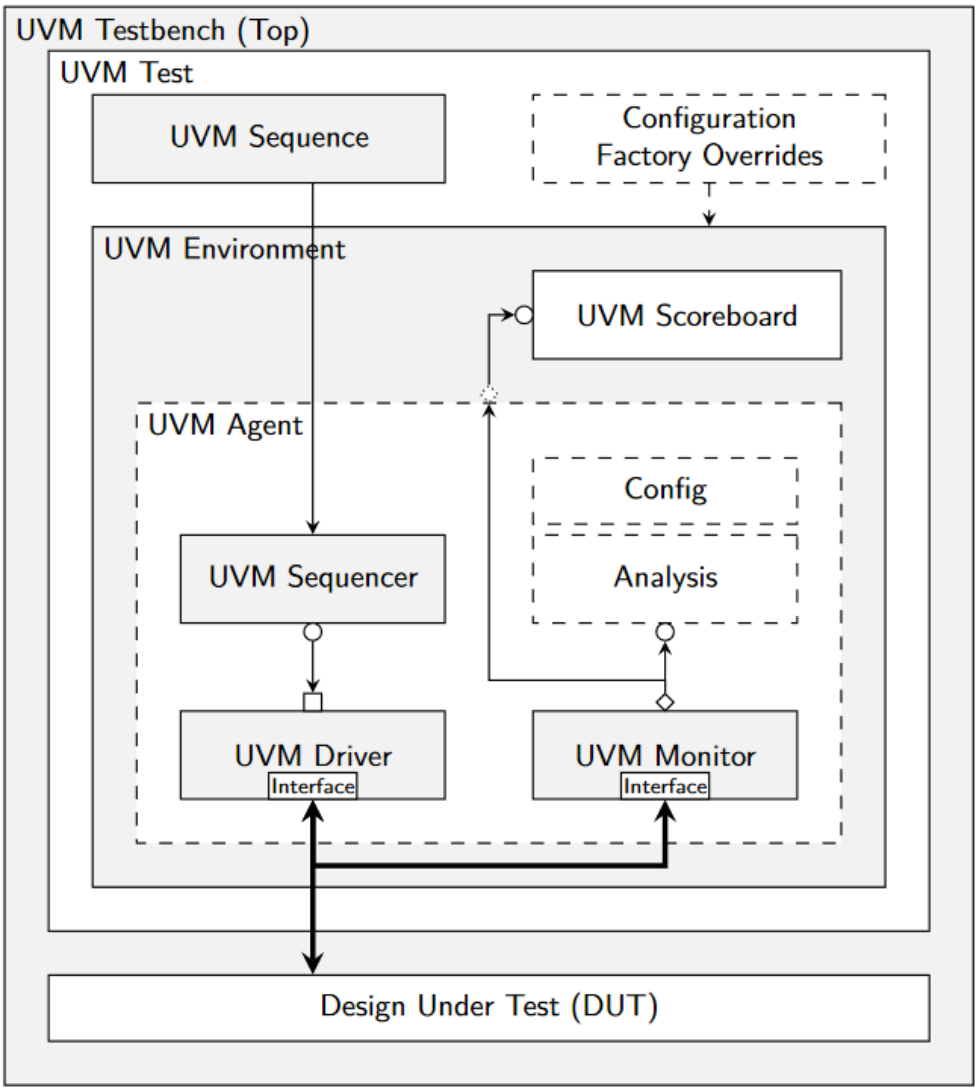
4.11 Top

Top modul je klasický SystemVerilogový modul, který slouží k propojení testovaného designu a testbenche. Stejně jako u normálního top modulu určeného k testování se zde nalézá generování hodinového signálu a signálu reset, instanci testovaného designu, případně i vytvoření instance rozhraní. Navíc se zde musí přidat interface do konfigurační databáze, aby se skrz ní k němu ostatní komponenty mohly připojit, a zavolání testů. Přidání do konfigurační databáze provádí následující řádek kódu:

```
uvm_config_db#(virtual if_name_type)::
        set(uvm_root::get(),"*","vif",if_name);
```

Je potřeba dodržet aby třetí parametr byl stejný pro funkce *get* i *set*, podle tohoto parametru se v databázi vyhledává. U volání testů je možné buď zavolat všechny testy, tehdy se použije funkce *run_test* bez parametrů, pokud je potřeba volat testy v určitém pořadí nebo provést jen jeden test, funkce bere jako parametr jméno komponenty test. Toto lze řešit i přímo při spuštění simulace pomocí parametru.

Pokud je vše v pořádku, při spuštění by se měl vytvořit testbench s hierarchií téměř totožnou jako na obrázku 4.2.



Obrázek 4.2: Hierarchie zapojení jednotlivých komponent v testebenchi za použití knihovny UVM [10].

Kapitola 5

Simulační nástroje

Pro vytvoření testbenche pro design není potřeba jen znalost SystemVerilogu, ale také prostředí, ve kterém testbench vznikne a simulátor, ve kterém se test spustí. Tato kapitola se tedy bude těmto nástrojům věnovat. Bude zde uvedeno několik nástrojů a způsobů jak psát kód a následně několik možností jak svůj design odsimulovat. U každého nástroje budou uvedeny přednosti a následně i můj názor na práci s ním.

5.1 Textové editory a IDE

Ačkoliv Verilog ani SystemVerilog není tak objemný co se týče velikosti kódu, při používání knihovny UVM kód začne bobtnat. Proto není dobrý nápad psát celý kód vlastnoručně a raději využít něco, co při tvorbě pomůže. V tomto případě jsou to různá IDE, textové editory s možností generování textů a podobně. Ty umožní programátorovi zaobírat se pouze kódem specifickým pro testovaný design, jako například řízení signálu v driveru.

5.1.1 DVT Eclipse IDE

DVT Eclipse je IDE vytvořené společností AMIQ podporující návrh hardware a jeho verifikace pomocí HDL jazyků. Jak už název napovídá, IDE bylo postaveno na platformě Eclipse [11]. Instalace je trochu zdlouhavější, jelikož je potřeba žádat o licenci skrz formulář, která během několika dní dorazí na email. Poté je potřeba cestu k licenci pomocí systemové proměnné předat Eclipse a program je připravený.

IDE se zdá být ze začátku velice zmatené, obsahuje spoustu tlačítek a možností. Naštěstí tu je v základu projekt pro začátečníky, na kterém se vše celkem rychle ukáže a vysvětlí. AMIQ také vytvořila dokumentaci[5], playlist na youtube[6] a další pomůcky pro používání jejich produktu. IDE pro-

[5] `https://www.dvteclipse.com/docs#dvt-eclipse-ide`

[6] `https://www.youtube.com/playlist?list=PLBx64n-99iZNiUJj_I8WkLLbRl8u0mi94`

gramátorovi nabízí spoustu nástrojů pro urychlení práce, ten asi největší z nich je obrovská zásoba templatů pro prakticky každý module SystemVerilogu i knihovny UVM. Programování se tak změní z psaní na klikání na templaty a přepisování jmen signálů a proměnných. IDE zvládá rozumně zvýrazňovat syntaxi, barvy je možné si změnit v nastavení IDE. Funguje zde velmi chytrý našeptávač, ale to se dá u IDE očekávat. IDE také zvládá už při psaní kódu upozorňovat na chyby v syntaxi nebo na případná varování. Využití klávesových zkratek je zde také možné, ale nakonec to není tak potřeba vzhledem k způsobu psaní kódu. IDE také obsahuje možnost propojení se simulátorem, což znamená že se simulace spustí přímo z IDE v simulátoru, bez potřeby vytvářet projekt, kompilovat a spouštět simulaci.

DVT Eclipse je velice užitečný nástroj pro tvoření testbenchů. Vytvořit testbench se mi díky němu povedlo velice rychle, i když klávesové zkratky často dělaly něco jiného, než jsem zvyklý. Při používání mě moc nepotěšil tmavý mód IDE, na který jsem při programování zvyklý a který bohužel moc nefungoval. Ale jinak na IDE není asi nic, co bych mu mohl vytknout.

5.1.2 Sublime Text a balíček suni_uvm

Sublime text je klasický textový editor od společnosti Sublime HQ, který nabízí spoustou funkcí. Instalace je velmi přímá a samotný editor nezabírá moc místa. Sublime text obsahuje velké množství klávesových zkratek a nastavení, které mohou velice zefektivnit vytváření kódu. Velice užitečným prvkem se ukázala být minimapa souboru na pravém kraji obrazovky, díky které se dá efektivně vyhledávat v kódu, pokud uživatel neví co hledat, ale ví kde to hledat. Formátování kódu je velice jednoduché. Odsazení si programátor může nastavit podle svých preferencí a i u už existujícího kódu jde toto velice jednoduše změnit. Editor využívá automatického odsazení, kdy při použití konstrukcí, po kterých se odsazení obvykle zvětší, jako je třeba vnitřek bloku funkce, jej editor sám zvětší. Nastavení ovšem není moc dobré pro začátečníky, místo okna s možnostmi editor obsahuje pouze konfigurační soubor. Naštěstí se uživatel s tímto nastavením vůbec nemusí setkat, pro používání to není vůbec potřebné.

Co se týče programování, sám o sobě toho moc neumí, maximálně zvýraznění syntaxe u často používaných jazyků. Mezi ty samozřejmě nepatří ani Verilog a ani SystemVerilog. Toto se ovšem dá vyřešit, a to pomocí přídavných balíčků. Těch má tento editor spoustu a řeší prakticky jakýkoliv problém co se týče obarvení kódu, vzhledu editoru či vytváření snippetů[7]. Bohužel, našeptávání, které editor obsahuje, není nejinteligentnější. Obvykle při psaní našeptává jakékoliv slovo, které se zatím v souboru objevilo, včetně kusů kometářů.

[7] snippet je šablona pro znovupoužitelný kus kódu

Sublime jako takový samozřejmě nepomáhá s vytvořením testbenche s knihovnou UVM. K tomu vznikl balíček *suni_uvm*. Ten obsahuje snippety pro vygenerovnání komponent a dalších konsturkcí knihovny UVM. Výhodou tohoto balíčku je i to, že se snippety dají upravit k obrazu svému. Možnosti instalace jsou dvě, buď se balíček stáhne z githubového repozitáře [12] a následně se vloží do složky editoru se snippety. Druhou možností je stáhnout balíček přímo pomocí manažera balíčků uvnitř editoru. Balíček je bohužel již několik let neudržovaný. To sice v tuto chvíli nevadí, v případě změny v knihovně UVM pak pravděpodobně dál používat nepůjde a musela by se najít nebo vytvořit alternativa.

5.1.3 Vivado

Vivado je návrhový software od firmy Xilinx. Vivado obsahuje, kromě spousty dalších nástrojů pro návrh designu, také textový editor určený pro psaní kódu. I když není zrovna příjemné v něm psát delší kód. Editor sám od sebe nenašeptává, programátor si návrhy musí zobrazit sám pomocí klávesové zkratky. A ani zvýrazňování syntaxe není nejlepší. Obojí však jde předefinovat v nastavení editoru. Celkově je podpora knihovny UVM špatná. Často se stává že Vivado tvrdí o UVM komponentách že nejsou deklarované i když jsou součástí knihovny. Orientace mezi soubory byla také velice obtížná. Ale to je problém Vivada jako celku, ne jen textového editoru. Pokud ale toto vše pomineme, editor ve Vivadu není zas tak špatný. Funguje zde několik klávesových zkratek pro jednodušší ovládání, formátování kódu zde není zas tak obtěžující. Dokonce je zde pár šablon pro generování kódu, i když jich je velice málo a jsou dost primitivní. Bohužel, šablony pro knihovnu UVM chybí. V nastavení si programátor může upravit editor podle svých představ, alespoň co se týče fontu, barev a odsazení.

5.1.4 Questa

Ačkoliv je Questa v první řadě simulátor, obsahuje i jednoduchý textový editor. Ten zvládá základní práci se SystemVerilogovým kódem, jako je zvýrazňování syntaxe a její kontrola, nicméně již po chvilce používání musí každému dojít, že editor není určený k vytváření kódu, ale spíše k jeho úpravám. Questa, podobně jako Vivado, neobsahuje žádnou nápovědu nebo našeptávání a snaha o formátování je zde většinou ztráta času. Naštěstí Questa obsahuje možnost automatického otevírání zdrojových kódu v externím programu. Zdejší editor jsem obvykle využil pouze k úpravám a vylepšením existujícího kódu, ve chvíli kdy vše fungovalo, jak jsem potřeboval a s kódem jsem byl spokojený, přesunul jsem se do jiného programu, abych kód zformátoval a okomentoval.

5.2 Simulátory

Po vytvoření testbenche je potřeba design otestovat a k tomu slouží simulátory. Při spuštění simulace se začne provádět kód testbenche a tedy posílat signály do testovaného designu. V průběhu simulace simulátor zaznamenává hodnoty všech možných signálů a transakcí jak v testovaném designu tak i v testbenchi. V dnešní době už simulátory mají GUI, v těchto simulátorech se tyto zaznamenané hodnoty vykreslují na takzvaný waveform[8]. Následující podkapitoly popisují některé známé simulátory, jejich funkce a hodnotí práci s nimi.

5.2.1 EDA Playground

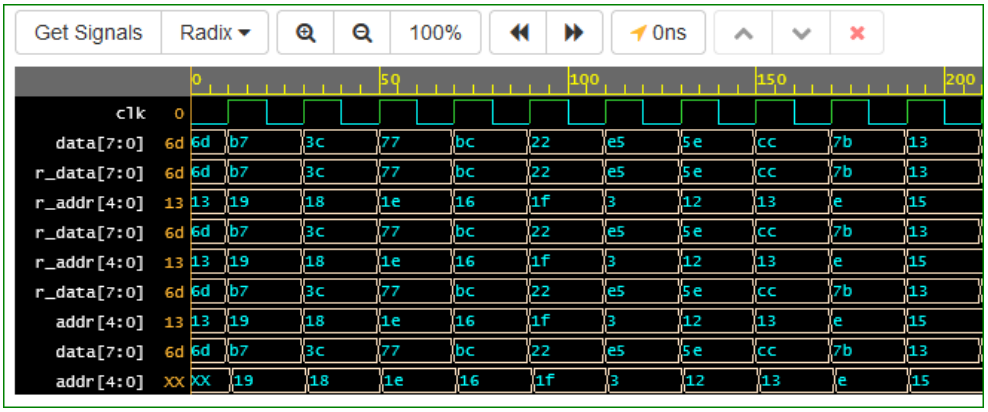
EDA Playground není úplně simulátor, ale webová stránka[9] určená k psaní a simulování HDL kódu. Web podporuje velké množství jazyků, jako Verilog, VHDL, SystemVerilog nebo SystemC. Stejně tak podporuje velké množství nástrojů k simulaci a syntéze. Bohužel, k většině nástrojů podporujících SystemVerilog je potřeba validace emailu, ke které je potřeba vyplnit krátký registrační formulář, včetně služebního či školního emailu. Nicméně, pro všechny své testy jsem využíval simulátoru Aldec Riviera Pro, ke kterému tato validace nebyla potřeba.

Tento simulátor podporuje všechny konstrukce SystemVerilogu i UVM, někdy je ovšem moc vlídný a bez problémů vezme kód, který by měl skončit chybou. Některé konstrukce Verilogu fungují dokonce špatně, příkladem třeba výpis pomocí *$monitor*. Tyto chyby naštěstí nejsou tak závažné, aby se simulátor nedal použít.

Co se týče prostředí webu, tak slavné to není. Dá se zde odsimulovat kdejaký kód v SystemVerilogu, je zde podporována knihovna UVM, ale psaní kódu, i jakékoliv malé úpravy, jsou zde velice nepříjemné. Webová stránka má svůj waveform, ale práce s ním je spíš za trest. Při pokusu o přidání většího množství signálů najednou stránka obvykle zamrzla a bylo potřeba jí aktualizovat. Pokud se signály na více místech jmenují stejně, tak se nedají rozeznat, protože na waveformu je jen jméno signálu. Používání kurzoru ve waveformu je také složité, bohužel se s ním nedá jezdit, jen klikat. Trefit se do správného místa je tedy někdy obtížné. Zoomování je tu taktéž složité, musí se tak dít skrz tlačítka, na kolečko myší i kombinaci tohoto s tlačítkem ctrl reaguje samotný prohlížeč a ne waveform. Hodnoty signálů jsou zde viditelné, možnosti pro soustavu výsledků jsou omezené jen na binární a hexadecimální. Posouvání signálů je také řešené pomocí tlačítek, tudíž také není nejlepší. Signály bohužel nejde obarvovat, čtení z waveformu je tak docela obtížné. To lze vidět na obrázku 5.1. Pro větší projekty je toto jako výstup nedostatečné. Pro menší projekty to dostatečné je, ale osobně bych dal přednost spíš výpisu do konzole.

[8] waveform znamená průběh signálu, v kontextu simulátoru je to okno s těmito průběhy

[9] `https://www.edaplayground.com/`



Obrázek 5.1: Signály zobrazené pomocí waveformu na EDA Playground. Signály se stejnými jmény jsou tu k nerozeznání.

V případě možnosti výběru mezi tímto webem a prakticky jakýmkoliv jiným simulátorem bych rozhodně volil jiný simulátor. Web EDA playground není stvořen pro vytváření velkých návrhů a testbenchů, spíše pro učení se a zkoušení si. I tak je ale možné zde vytvořit rozsáhlý projekt, jak je možné vidět na některých veřejných „hřištích“. Je ovšem důležité počítat i s tím že simulace má omezený simulační čas, je tedy možné že větší simulace bude násilně ukončena bez jakéhokoliv výstupu.

5.2.2 Vivado

Práce se simulátorem uvnitř Vivada je celkem jednoduchá. Ve chvíli, kdy se povede nastavit správný soubor k simulaci, se prakticky nedá nic zkazit a to obvykle není problém, dokud projekt neobsahuje více jak jeden testbench. Pokud se simulace zapne bez problémů, objeví se na obrazovce několik přehledných oken rozdělených na waveform, hierarchii modulů a konzoli obsahující mimo jiné výstupy testbenche, pokud nějaké obsahuje. Vzhled signálů zobrazených ve waveformu lze různě obarvovat a skládat do skupin, pro rozdělení skupin se používají dividery, které se taktéž dají obarvit. Defaultně se do waveformu vkládají buď vnitřní signály nejvyššího modulu v hierarchii nebo vstupy a výstupy tohoto modulu. Pokud je potřeba přidat další signály, musí se najít v okně s hierarchií modulů a následně přetáhnout do waveformu. Okno waveformu také zobrazuje hodnotu signálů v čase, kde je zrovna nastavený kurzor. U těchto hodnot se dá změnit číselná soustava.

Vivado podporuje používání SystemVerilogu v simulacích a dokonce i pro syntézu již několik let, podpora pro UVM tu je od verze 2019.2. Pokud je součástí projektu něco z knihovny UVM, musí se Vivadu dát vědět aby používal svou předkompilovanou knihovnu pomocí přepínače *-L UVM*. Ačkoliv by toto mělo být vše, ani tak se mi nepovedlo ve Vivadu spustit svůj testovací projekt, který v jiných testovaných simulátorech bez problému fungoval.

Ovšem největší nevýhodou simulátoru je jeho rychlost. Spouštění simulace je někdy i v řádu minut, a to i pro menší projekty. Další obrovská nevýhoda, která se začla objevovat v posledních letech, je sken antiviru, který se spouští při každém spuštění simulátoru. Nejen že to zpomalí už tak pomalé spouštění simulace, ale díky některým antivirům se stalo Vivado zcela nefunkční[10]. Řešení tohoto problému je naštěstí celkem jednoduché, a to antivirus při používání Vivada vypínat.

5.2.3 Questa

Questa je simulátor původně od firmy Mentor Graphics kterou v roce 2017 převzala firma Siemens. Tento simulátor se z počátku tváří velmi user unfriendly, nepomáhá tomu ani fakt, že k simulátoru není moc tutoriálů, jak s ním pracovat. Ale když si člověk dá tu práci a se simulátorem se chvíli trápí, nakonec celkem brzy zjistí že Questa je velmi silný nástroj.

Po vytvoření projektu jsou obvykle vidět dvě okna, jedno se soubory projektu a jedno s konzolí simulátoru. Okno s kódem se objeví při otevření nějakého ze souborů projektu. Kompilace, spuštění simulace a další činnosti se dají spustit buď pomocí konzole a nebo pomocí tlačítek a menu v horní části obrazovky.

Po spuštění simulace se objeví waveform a okno s objekty. V okně se soubory projektu přibude další záložka s moduly. Při klikání na různé moduly v tomto okně se mění obsah okna s objekty, kde se mění hodnoty signálů a proměnných uvnitř modulu. Okno s waveformy je hodně podobné jako u Vivada, i s podobnými funkcemi. Signály zde jde obarvovat a oddělovat pomocí dividerů, navíc tu jde signály seskupovat. Takto sloučené signály jde skrýt. Tím se waveform stává mnohem přehlednější pokud vykresluje velké množství signálů. Dále je zde velké množství nástrojů, které kontrolují různé prvky testbenche i testovaného designu, jako je třeba obsah pamětí, pokrytí použitých testů nebo třeba hierarchii.

Jak už bylo řečeno na začátku, s Questou je velice těžké začít pracovat. Ale jakmile se člověk dostane přes počáteční nevědomí, rychle se z Questy stává velice silný nástroj pro simulaci a těžko se přechází k jiným, méně obsáhlým nástrojům.

[10] `https://forums.xilinx.com/t5/Design-Entry/Vivado-2018-3-simulation-won-t-run-with-AntiVirus-turned-on/td-p/938151`

Kapitola 6

Vytváření textů

Součástí této práce je také vytvoření pomocného textu. Tento text by měl sloužit jako pomůcka programátorovi, pokud si nemohl vzpomenout, jak použít anebo vytvořit nějakou konstrukci jazyka SystemVerilog a knihovny UVM. Zároveň k tomuto textu vznikl doprovodný zdrojový kód, kde jednotlivé konstrukce byly použity, případně připraveny ke spuštění v simulátoru. Tato kapitola popisuje průběh vytváření tohoto textu a doprovodných zdrojových kódů.

6.1 Příprava

Než začla tvorba textů samotných, bylo nutné provést přípravu. Jako první krok přípravy bylo průzkum již existujících tutoriálů a návodů k SystemVerilogu a knihovně UVM. Samozřejmě se na internetu dají najít, ale ne v tak hojném zastoupení jako pro jiné programovací jazyky. Z existujících jsem pro své studium hojně využíval tutoriály na ChipVerify[11], Verification Guide [12] a Asic World[13]. Tyto zdroje obvykle byly dostatečné k vytvoření základní představy o konstrukcích a použití, bohužel se občas stávalo že každý tvrdil něco jiného. Další výtkou je, že text, ačkoliv obvykle obsahoval všechny důležité informace, nebyl často lehce pochopitelný. Stávalo se tedy, že jsem nad některými příklady z těchto stránek musel trávit několik desítek minut až hodin, než jsem pochopil obsah textu. A nakonec zde byla navigace mezi různými tématy na těchto webech. Ta nebyla ani trochu jednoduchá, pokud jsem potřeboval informace z části, která přímo nepředcházela ani nenásledovala tu aktuální, bylo nejrychlejší proklikat se od domovské stránky. Tato tři fakta, ačkoliv nejsou nijak závažná, mě utvrdila že tato práce není zbytečná.

Dalším zdrojem, který jsem používal zároveň s výše uvedenými, byl standard jazyka SystemVerilog [1]. Spíše než k učení jsem tento dokument používal

[11] `https://www.chipverify.com/`
[12] `https://verificationguide.com/`
[13] `http://www.asic-world.com/systemverilog/tutorial.html`

jako kontrolu nebo upřesnění informací z ostatních zdrojů. Hlavním důvodem je to, že text je ještě více technický než ostatní zdroje, takže se četl ještě hůř. Měl jsem ale jistotu, že informace v tomto dokumentu budou pravdivé.

Ve chvíli, kdy jsem si myslel, že studovanou část zvládám použít, přišlo na řadu volba nástrojů a následné učení se s nimi. Z nástrojů uvedených v kapitole 5 jsem potřeboval vybrat pomocí čeho si své příklady vytvořím a pomocí čeho si je odsimuluji. Nakonec jsem využil více nástrojů, pro psaní kódu jsem použil textový editor Sublime a pro simulaci jsem nejdříve využíval webové stránky EDA Playground, následně jsem však přešel na simulátor Questa. Důvodem k tomuto přechodu byla zvyšující se obtížnost pochopení konstrukcí SystemVerilogu, kdy při simulaci bylo mnohem jednodušší si výsledky zobrazit v Questě než na EDA Playground. Volba Sublime byla téměř jasná, jediným konkurentem bylo IDE Eclipse, na který jsem ovšem nebyl zvyklý.

Protože jsem Sublime používal již dříve, nebylo těžké si zvyknout. Aby byla práce s ním jednodušší, musel jsem stáhnout balíčky pro zvýrazňování syntaxe SystemVerilogu a UVM. Poté už si stačilo jen chvilku ošahat nové našeptávání a zjistit, které zkratky vedou ke generování kódu, pak už byla příprava textového editoru hotová.

V případě simulátoru jsem nejdříve zvolil cestu jednoduchosti. Používání simulátoru na EDA Playground je velice jednoduché a není potřeba žádné velké nastavování. Z nabídky se jen zvolí jazyk, dále simulátor, který se použije a na konec jestli se po dokončení simulace otevře waveform. Pro spuštění pak stačí stisk tlačítka a stránka provede vše sama. Po chvilce mi trochu začalo vadit, že kód doplněný na stránkách simulátoru se formátuje trošku jinak, než jsem měl nastavené v Sublime a nebylo možné to změnit. To jsem prozatimně vyřešil, tím že jsem všechny úpravy prováděl nejdříve v textovém editoru a následně jsem kód překopíroval. Začaly se však objevovat další nedostatky, jak je popsáno v kapitole 5.2.1. Protože pro mě byl waveform jednou z hlavních cest k pochopení některých problémových částí, rozhodl jsem se dát přednost raději simulátoru Questa.

Používání Questy bylo na začátku velmi obtížné. Pro spuštění je potřeba licence, kterou jsem získával ze školního serveru. Při prvním spuštění jsem se v programu vůbec nevyznal a trvalo několik hodin něž jsem zvládl základní úkony, jako je vytvoření projektu tam, kde jsem ho chtěl mít uložen, zkompilování jednotlivých zdrojových kódů, vypsání chyb ve zdrojovém kódu, který se nepovedlo zkompilovat anebo spuštění simulace bez toho, aby se mi díky optimalizaci nezobrazily některé signály. Jakmile jsem přišel jak všeho docílit, sám jsem uznal, že je práce s tímto simulátorem mnohem příjemnější. Dokonce jsem zjistil, že některé konstrukce se v těchto dvou simulátorech chovaly odlišně, jako například příkaz $monitor, a chování v Questě bylo to očekávané. Navíc Questa nabízí spoustu dalších nástrojů pro sledování obsahů paměti, pokrytí a podobně, které mi pomohly s pochopením některých částí mnohem rychleji než při používání jen výpisů do konzole a waveformu.

Ve chvíli kdy jsem měl ujasněno jak fungují důležité části SystemVerilogu

a knihovny UVM, jaké nástroje používat jak s nimi pracovat, vrhl jsem se na psaní pomocných textů a ukázkových zdrojových kódů k nim.

6.2 Vznik zdrojových kódů

Jako první začaly vznikat zdrojové kódy. Při vytváření vzorových zdrojových kódů jsem se rozhodl co nejvíce rozdělit jednotlivé konstrukce tak, aby se nové funkčnosti konstrukce neobjevovaly u kódů, které se touto konstrukcí nezabývaly. Tím prakticky vzniklo pořadí, ve kterém následně vznikly texty k těmto konstrukcím.

Při vzniku kódů pro SystemVerilogové konstrukce byla snaha o to, aby kód byl dostatečně okomentovaný tak, aby byl samovysvětlující. Záměrem bylo, aby zkušenější člověk nemusel procházet jak text tak kód a hledat, co vlastně potřebuje, ale stačilo mu otevřít jen jedno a vyčíst odtud vše potřebné. Kódy neobsahují vše, co konstrukce zvládnou, ale vybral sem pouze ty funkčnosti o kterých jsem usoudil, že by mohly být často používané. Kódy také obsahují velké množství výpisů do konzole, které mohou pomoct k vysvětlení jisté funkčnosti. Díky tomu si méně zkušení lidé mohli přečíst, co funkčnost má udělat a následně v kódu zkusit odhadnout, jaký bude výsledek po použití této funkčnosti. Potom si zdrojový kód mohli pustit a zkontrolovat, jestli funkčnost pochopili, anebo ne a je potřeba se nad ní více zamyslet. Některým lidem takový příklad může pomoci funkčnost pochopit.

První kód se věnuje randomizaci. Kód obsahuje třídu, ve které je několik členských proměnných, množství constraintů omezujících možné hodnoty těchto proměnných a funkce pro výpis. Dále obsahuje modul s několika instancemi této třídy a s funkcí, která tyto instance různě randomizuje.

Kód věnující se interface obsahuje dva moduly reprezentující násobičku a program, který generuje čísla k pronásobení. Tyto komponenty jsou propojené pomocí interface. Vše je instancováno v modulu top, který vše obaluje.

Další kód ukazuje použití pokrytí. Uvnitř kódu se nachází třída s dvěmi členskými proměnnými, constraint, který zabraňuje, aby jedna z členských proměnných nabývala všech hodnot a jedna covergroup. Uvnitř ní je několik coverpointů reprezentující různé způsoby vytváření binů. Nakonec se zde nachází program uvnitř kterého je instance třídy, jejíž obsah se několikrát randomizuje a následně se vypíše stav pokrytí.

Dále vznikl kód věnující se funkci clocking bloku. Zde je propojený pomocí interface program generující vstupy a modul, který se chová jako klopný obvod typu D. Vše je vytvořeno uvnitř modulu top. Do konzole se v průběhu simulace vypisují všechny změny na vstupu a výstupu klopného obvodu i se simulačním časem.

U kódů zabývajících se knihovnou UVM jsem zvolil mírně odlišný přístup. Zde jsem vytvořil projekt k vytvoření testbenche pro jednoduchý design, v mém případě sčítačky. Ještě před vytvořením prvních komponent testbenche

jsem vytvořil kódy zabývající se prací s funkcemi třídy *uvm_object* knihovny. Zde jsem se zabýval UVM makry, dále výpisy, kopírování a porovnávání objektů. Po tomto následovalo několik ukázkových kódů k propojování komponent pomocí TLM portů. Po rozebrání těchto základů přišly na řadu komponenty tvořící testbench. Pořadí jsem zvolil z vnitřku výše, tedy od sekvence k testu. Přišlo mi že je to tak jednodušší k pochopení toho, co komponenty mají na starost. Navíc jsem se ani v kódu a ani v textu nemusel odkazovat na budoucí části, jen na předchozí. V kódu pak nechyběly okomentované důležité části komponent, stejně jako výpisy do konzole. Ty hlavně sledují cestu transakce po testbenchi, případně ohlašují, co jednotlivé komponenty provádějí. Výsledkem této části je nakonec funkční UVM testbench pro sčítačku.

Po dokončení zdrojového kódu ke konstrukci jsem rovnou vytvořil pomocný text vysvětlující funkci této konstrukce.

6.3 Vznik textů

Texty k jednotlivým konstrukcím vznikaly hned po vzniku ukázkového kódu. Text se nesnaží vysvětlit, jakým způsobem operace probíhají, jen to, jaký je výsledek operace. Tím se značně zredukoval obsah textů o informace, které by stejně v takovém textu moc lidí nevyhledávalo. Stejně tak byla snaha o to, aby text byl co nejméně technický a aby mu rozuměl každý s alespoň základní znalostí Verilogu. Při vytváření textů vznikl malý problém s tím, jak z textu do zdrojových kódů odkazovat. Nejprve se tak dělo podle čísla řádků na kterém se funkčnosti které text projednával, to se velice brzy ukázalo jako špatný nápad ze zřejmých důvodů. Rozhodl jsem se tedy vytvořit v kódu pomocí komentářů záchytné body. Ty obsahovaly klíčové slovo podle kterého se v kódu dalo vyhledávat pomocí funkce najít. V textu jsem pak jen funkčnost vysvětlil a pod jakým záchytným bodem ji v kódu najít. I toto řešení má svá úskalí, třeba pokud použiji toto klíčové slovo jako jméno signálu, vyhledávání nebude tak účinné. Ale myslím si, že toto se dá jednodušše ohlídat, případně je možné na začátek vzorových zdrojových kódů uvést seznam těchto záchytných bodů. Tím se šance k použití stejného jména sníží.

Pro každou z konstrukcí jazyka vznikl jeden text, který se jí zabýval. Jak bylo zmíněno v kapitole výše, při vytváření zdrojových kódů vzniklo jisté pořadí vzniku textů pro konstrukce jazyka SystemVerilog. Bylo tomu tak kvůli snaze zabránit použití nezmíněných konstrukcí nebo jejich funkčností

v předchozích testech. Toto pořadí bylo následovné:

1. Randomizace
2. Interface
3. Pokrytí
4. Clocking bloky
5. Aserce

V těchto textech je vysvětleno jak konstrukce syntakticky vypadá a k čemu slouží. Následně je rozebráno tělo konstrukcí, texty se věnují jednotlivým funkčnostem na jejichž příklady pak odkazují do pomocných kódů.

U konstrukcí knihovny UVM je to víceméně stejné. Na začátek je zde úvodní, text který se zaměřuje na funkce třídy *uvm_object*. Je zde vysvětleno, jak používat uvm makra a funkce, které pak továrna vygeneruje, případně jak si napsat tyto funkce vlastní. Po tomto úvodu následuje část zabývající se propojování objektů pomocí TLM. Zde je na příkladu producenta a konzumenta ukázáno vytvoření portů, je tu stanoveno, který objekt obsahuje implementace funkcí a který je volá a jsou zde rozebrány téměř všechny možnosti propojení pomocí těchto portů. Než se text věnuje komponentám užívaným v testbenchi, objevuje se zde krátká kapitola pro přípravu jednoduchého modulu sčítačky, pro kterou se pak bude testbench stavět. Pak už následují komponenty testbenche, zde je pořadí stanoveno, stejně jako při vzniku zdrojových kódů, od vnitřku ven. U jednotlivých komponent se probírájí funkce které mají plnit, věnuje se zde důležitým součástem jako jsou funkce, případně porty.

Kapitola 7

Vytváření úloh

Druhým výstupem této práce mělo být vytvoření vzorových úloh pro předmět Simulace číslicových obvodů. Tyto úlohy měly být zaměřeny na vytvoření vlastního designu, ke kterému následně student vytvořil testbench, který jeho design otestoval. Jazyk pro vytvoření designu byl Verilog, testbenche pak vznikaly v jazyce SystemVerilog s případným použitím knihovny UVM.

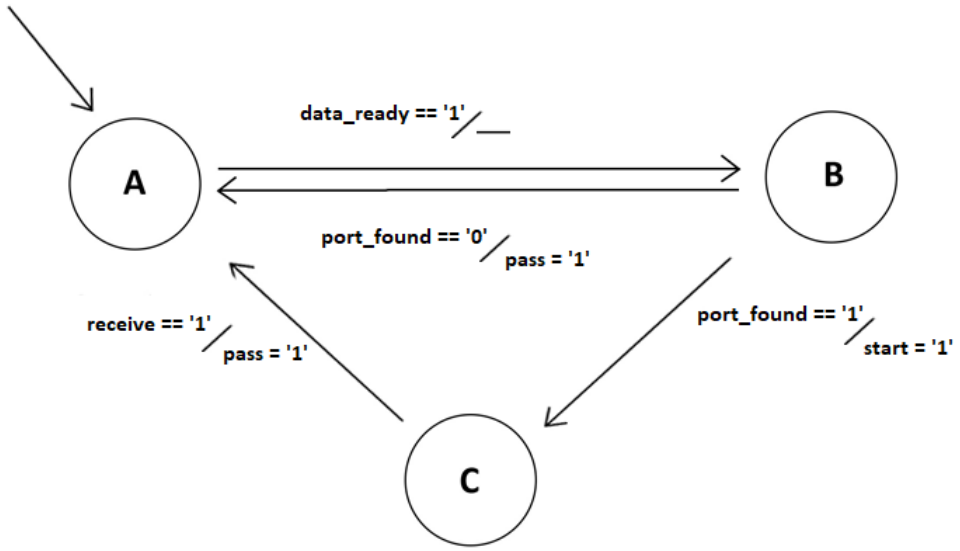
Pro tento předmět nakonec vznikly dvě úlohy, z nichž jedna má dvě různé varianty. Těmto úlohám se věnují následující podkapitoly.

7.1 Síťový přepínač

Úloha síťový přepínač se inspiruje příkladem ze starších slidů předmětu Simulace číslicových obvodů [13], kde vystupuje jako příklad pro rozhraní mezi UVM testbenchem a testovaným designem. Přepínač obsahuje dvě skupiny signálů, nebo pro jednoduchost porty, jednu pro nastavení samotného přepínače a druhou pro přijímání dat. Zároveň je zde pole výstupních portů, počet se zde udává parametrem. Funkce přepínače je přeposílat pakety na svém datovém vstupu na jeden, případně více, ze svých výstupů. Děje se tak podle adresy, která je taktéž na datovém vstupu přepínače a udává který výstup bude použit. Port pro nastavování se používá pro změny adres uložených v paměti přepínače, čímž se mění to, který výstup odpovídá které adrese.

Uvnitř přepínače, krom jednoduchých hradel a několika multiplexorů, jsou v mém návrhu tři komponenty: komparátor, paměť adres a konečný automat pro řízení přepínače. Nejjednodušší z nich je komparátor, ten na vstupu prijímá dvě adresy a jeho výstup odpovídá tomu, jestli se adresy rovnají. Uvnitř přepínače se takových komparátorů nachází stejně jako je v přepínači výstupů, jelikož jeden komparátor porovnává jednu adresu.

Další, podobně složitá, komponenta je paměť pro adresy. V té se tedy skladují adresy, které odpovídají jednotlivým výstupním portům. Zde adresa na i-té pozici odpovídá i-tému výstupnímu portu. Malou změnou od klasické



Obrázek 7.1: Graf konečného automatu řídící síťový přepínač

paměti je zde její výstup, kde pro každou buňka paměti je vyvedena ven. Jinak ale tento paměťový blok funguje stejně jako normální paměť.

Poslední komponenta uvnitř přepínače je kontroler. Jedná se o konečný automat, který řídí funkci přepínače. Na začátku automat čeká, než na datový port přepínače přijdou data. Jakmile se tak stane, na základě výstupů z komparátorů přijde do automatu signál značící nalezení odpovídající adresy té žádané. Pokud tato adresa nalezená byla, data ze vstupu se objeví na odpovídajícím výstupním portu a jakmile se na tomto portu objeví potvrzení o přijmutí, automat se přesouvá do prvního stavu a čeká na další data. Pokud v paměti žádná taková adresa není, přepínač data ignoruje a automat rovnou čeká na nová data. Graf automatu je zobrazen na obrázku 7.1, který je také součástí zadání.

Dalším krokem bylo vytvoření top komponenty, která reprezentuje celý síťový přepínač. Jediné, co zde zbývá, je zapojit komponenty zmíněné výše, k tomuto účelu vznikl obrázek 7.2. Podle mého názoru je vytvořit obrázek zapojení mnohem jednodušší než toto zapojení slovně popsat a i pro studenta to je lépe pochopitelné.

Druhou částí úlohy bylo vyrobit testbench pro takto vzniklý návrh. Zde není řečeno, jak má takový testbench vypadat, je tedy zcela na studentově uvážení jak ho vyrobí, jedinou podmínkou je, aby jeho návrh otestoval dostatečně. Vzorový testbench, který jsem vytvořil, sestává ze stínové kopie paměti z přepínače a kopie výstupů přepínače, tasků pro změnu dat v paměti a pro odeslání dat skrz přepínač a tasků kontrolující obsah paměti a výstup

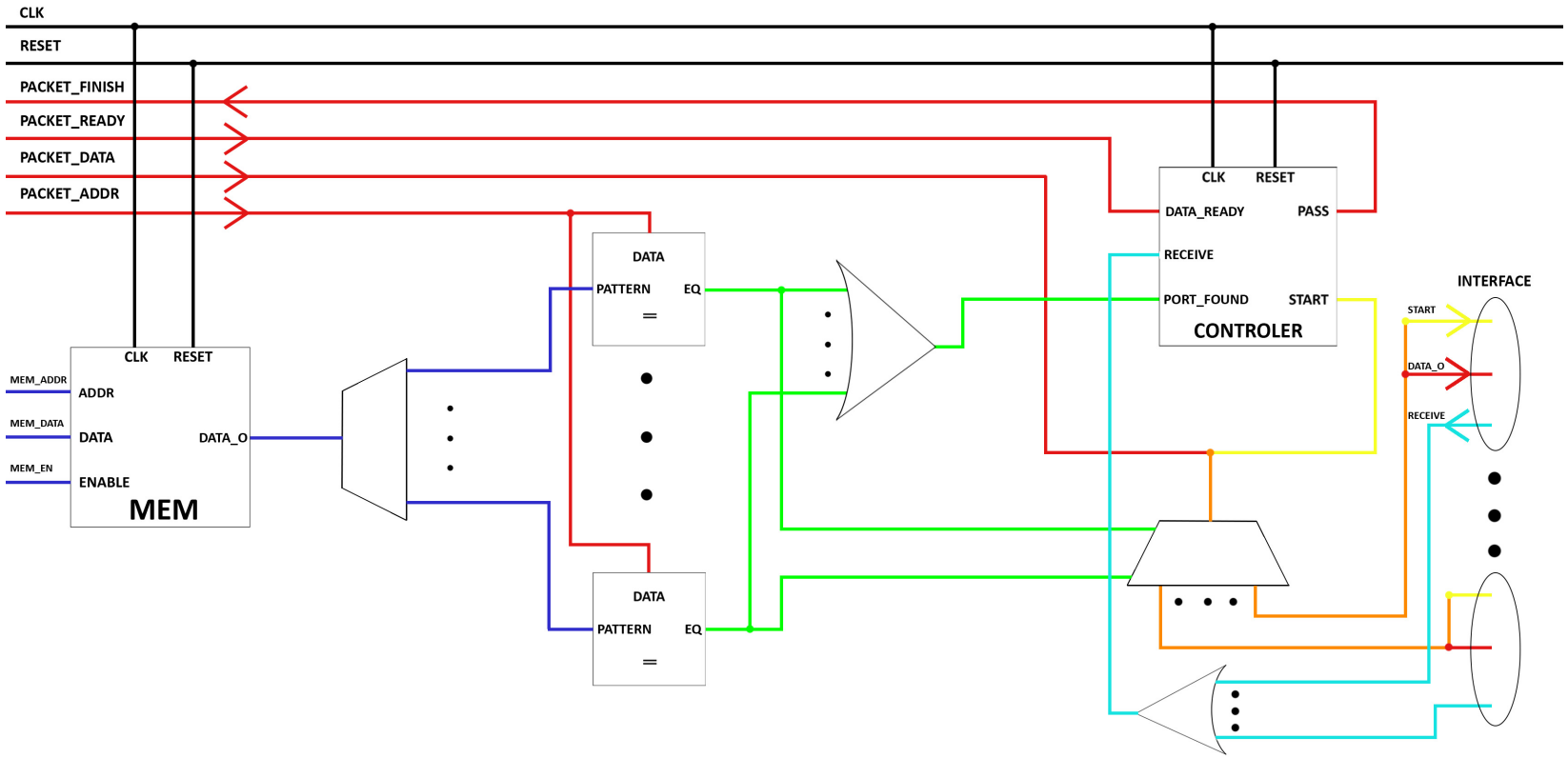
přepínače. Hlavní test probíhá v následujících krocích:

1. Inicializace, vyplnění paměti náhodnými hodnotami.
2. Test náhodného měnení adres v paměti.
3. Test posílání dat na adresy z paměti.
4. Test posílání dat na adresy které v paměti nejsou.
5. Test všech operací.

Během každého z těchto testů probíhá kontrola oproti stínové kopii, případně oproti patřičnému výstupu. V případě jakékoliv neshody se test zastaví, vypíše do konzole, která data se liší a ukončí se.

Tato úloha se také měnila v průběhu jejího vzniku. Úplně první verze nebyla parametrizovatelná, výstupy modulů obvykle tvořil jeden velký signál, který bylo potřeba rozsekat a výstupní porty přepínače tvořil dlouhý seznam signálů. Samozřejmě v takovém stavu nešlo úlohu prezentovat, proto bylo potřeba jí později předělat, ale jako vzor pro vytvoření testbenche a potvrzení funkčnosti návrhu prozatím takto stačila. První úpravou bylo přidání čtyř parametrů, ty řešily počet výstupních portů, délku adresy, délku dat a délku adresy do paměti přepínače. Poslední parametr se ukázal být nepodstatný a proto je v poslední verzi obvykle rovný $log\lceil port\_count \rceil$, kde $port\_count$ značí počet výstupních portů. Parametr v kódu ovšem dál zůstává a je možné ho měnit. S přidáním parametrů se tak změnila většina portlistů komponent, kde se objevily vektory místo dlouhých seznamů signálů. Další, celkem úsměvnou, změnou bylo předělání výstupních portů na interface a zpět. Předělání těchto portů na interface se ze začátku zdálo jako dobrý nápad, zkrátil se portlist síťového přepínače, celkově se pak kód stal přehlednější. Předělání testbenche bylo pak trochu složitější, bylo zde potřeba používání virtuálních interface pro testování správnosti výstupů, zde tato změna naopak kód trochu znepřehlednila. Nakonec se ale vše vrátilo zpět, protože bylo rozhodnuto že tato úloha nebude používat konstrukce z jazyka SystemVerilog. Další změny už úlohu nepotkaly, alespoň ne tak významné jako tyto.

Jako poslední pro úlohu vznikla stránka se zadáním. To obsahuje popis komponent, seznam vstupních a výstupních signálů a jména parametrů. Dále pro každou komponentu obsahuje popis funkce a taktéž seznam vstupních a výstupních signálů. Na konec je zde popsaný top modul, zde je znovu seznam vstupních a výstupních signálů, na úplném konci zadání je obrázek popisující propojení jednotlivých komponent. V obrázku jsou barevně odlišeny různé skupiny signálů, jako třeba signály pro nastavování přepínače, vstupní datové signály nebo signály označující nalezení adresy v paměti. Pokud student dodrží pojmenování vstupních a výstupních signálů a komponent, mělo by být možné k otestování využít vzorový testbench, bohužel to ale není vyzkoušené.



Obrázek 7.2: Popis zapojení síťového přepínače podle kterého mají studenti propojit komponenty. Signály jsou pro přehlednost barevně odlišeny.

7.2 Testbench pro jednoduché CPU

Druhá úloha se věnuje vytvoření testbenche pro jednocyklový CPU. Návrh toho CPU je zadávaný jako úloha v předmětu Architektura počítačových systémů v bakalářském studiu. Úloha je rozdělena do dvou částí, obě tyto části spočívají ve vytvoření testbenche pro CPU, každá ale používá jiné prostředky. V první části je k vytvoření testbenche použito konstrukcí jazyka SystemVerilog, v té druhé se k němu přidají konstrukce knihovny UVM. V následujících kapitolách se objeví krátký popis procesoru a instrukční sady následovány popisem obou variant úlohy.

7.2.1 Procesor

Testovaný design je pro tuto úlohu jednocyklový procesor vytvořený jako semestrální projekt v předmětu Architektura počítačových systémů. Procesor má rozdělenou paměť pro data a instrukce. Tento navržený procesor obsahuje registrové pole o 32 registrech, ALU jednotku a řídicí jednotku, která ze zakódované instrukce vygeneruje kontrolní signály pro ostatní komponenty. Návrh zapojení je na obrázku 7.3. Procesor při každém taktu hodinového signálu zpracuje jednu instrukci a její výsledek uloží do registru nebo paměti, v případě skoku nastaví adresu další instrukce.

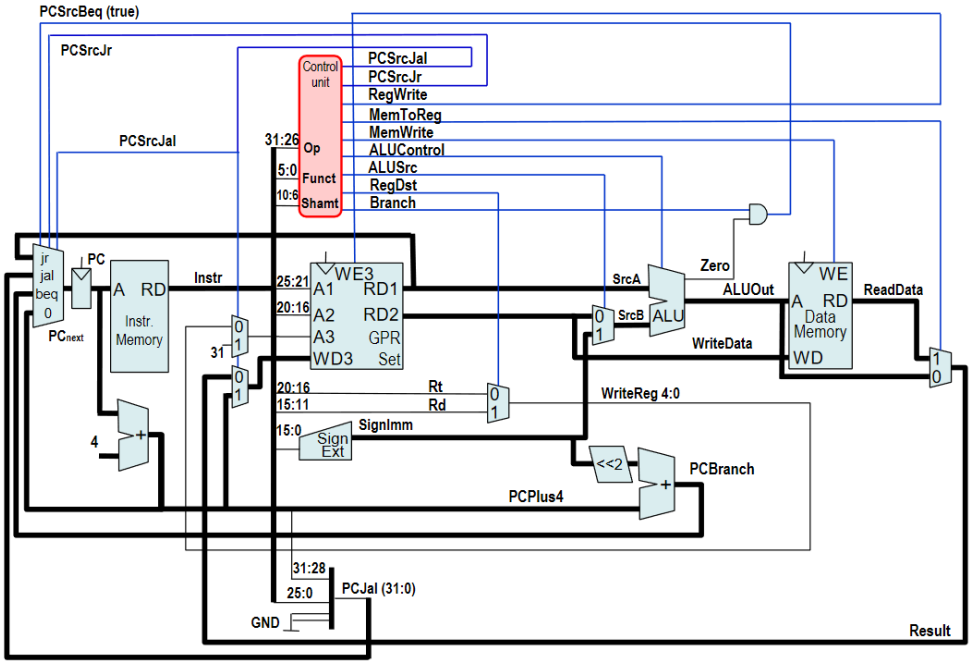
Registrové pole obsahuje 32 bitové registry. Pole umožňuje najednou číst dva registry a do jednoho zapisovat. Zápis se zde provádí při náběžné hraně hodinového signálu, vystavení hodnoty na výstup je okamžité. Zvláštní chování má nultý registr, ten je nepřepisovatelný a jeho hodnota je vždy nulová.

Aritmeticko-logická jednotka slouží k provádění aritmetických a logických operací anebo výpočtů adres při práci s pamětí dat. Jejím úkolem je také generování příznaků, a to Zero, Carry a Overflow. Činnost jednotky řídí řídicí jednotka která do ALU posílá informace o prováděné instrukci.

Řídicí jednotka slouží k nastavení dalších komponent procesoru. Z částí zakódované instrukce generuje kontrolní signály, kterými řídí multiplexory, ALU, registrové pole a paměť dat v registru.

7.2.1.1 Instrukce

Úloha z předmětu Architektura počítačových systémů obsahuje seznam instrukcí, které je potřeba, aby procesor podporoval. Seznam těchto instrukcí je v tabulce 7.1, která obsahuje jméno, popis operace a zakódování instrukce. Takto zakódované instrukce pak v případě obou testbenchů vstupují do procesoru. Instrukce se skládají z jednotlivých částí podle druhu instrukce, všechny obsahují svůj vlastní opcode, což je prvních šest bitů instrukce. Ve zbytku instrukce se pak nacházejí informace podle typu instrukce. Těmito informacemi mohou být například čísla registrů, se kterými pracuje, informace pro řídící



Obrázek 7.3: Popis zapojení komponent uvnitř jednocyklového procesoru. Signály obsahující data jsou obarvené černě, řídicí signály jsou modré[14].

jednotku k nastavení signálů nebo přímá hodnota pro uložení do registrů anebo manipulaci s programovým čítačem v případě skoku.

7.2.2 Testbench 1

První verzí úkolu bylo vytvořit testbench v jazyce SystemVerilog, ale bez použití knihovny UVM. Při vytváření zadání jsem se tak snažil o použití co největšího množství novinek v jazyce. Celý test se nacházel v top modulu, který obsahoval instance procesoru, testovací modul pojmenovaný driver a interface spojující tyto dva moduly. Krom toho modul generuje hodinový signál pro oba tyto moduly.

Interface je jednoduchý modul, obsahuje pouze definici dvou modportů pro obě komponenty které spojuje. Jediným vstupem do interfacu je hodinový signál, který pak interface rozvádí do obou komponent.

Protože jsem nechtěl zdrojový kód samotného procesoru předělávat, ale chtěl jsem použít interface pro propojení, součástí řešení je vytvořit wrapper pro procesor. Tento wrapper obsahuje instanci procesoru popsaného výše a namapování signálu z interfacu na vstupy a výstupy procesoru.

Před vytvářením samotného modulu bylo potřeba vytvořit nějakou reprezentaci intstrukce. Pro tento testbench tak vznikla třída, která instrukci představuje. Pro reprezentaci typu instrukce vznikl výčtový typ, jehož pro-

Instr	Operace	Zakódování
add	d = s + t;	0000 00ss ssst tttt dddd d000 0010 0000
sub	d = s - t;	0000 00ss ssst tttt dddd d000 0010 0010
and	d = s & t;	0000 00ss ssst tttt dddd d000 0010 0100
or	d = s | t;	0000 00ss ssst tttt dddd d000 0010 0101
slt	d = (s<t) ? 1 : 0;	0000 00ss ssst tttt dddd d000 0010 1010
addi	t = s + imm;	0010 00ss ssst tttt iiii iiii iiii iiii
lw	t = MEM[s + offset];	1000 11ss ssst tttt iiii iiii iiii iiii
sw	MEM[s + offset] = t;	1010 11ss ssst tttt iiii iiii iiii iiii
jr	PC = s;	0001 11ss sss0 0000 0000 0000 0000 1000
sllv	d = t <<s;	0000 00ss ssst tttt dddd d000 0000 0100
srlv	d = (unsigned)t >>s;	0000 00ss ssst tttt dddd d000 0000 0110
srav	d = (signed)t >>s;	0000 00ss ssst tttt dddd d000 0000 0111
beq	s == t potom PC=PC+4+(offset<<2); jinak PC=PC+4;	0001 00ss ssst tttt iiii iiii iiii iiii
jal	$31=PC+4; PC=(PC&0xf0000000) | (target<<2)	0000 11ii iiii iiii iiii iiii iiii iiii
addu	$d_{31:24}$=$s_{31:24}$+$t_{31:24}$ $d_{23:16}$=$s_{23:16}$+$t_{23:16}$ ...	0111 11ss ssst tttt dddd d000 0001 0000
addu_s	$d_{31:24}$=sat($s_{31:24}$+$t_{31:24}$) $d_{23:16}$=sat($s_{23:16}$+$t_{23:16}$) ...	0111 11ss ssst tttt dddd d001 0001 0000

Tabulka 7.1: Tabulka instrukcí které podporuje vytvořený jednocyklový počítač a jejich zakódování. Písmena *i* v zakódování reprezentují přímou hodnotu, pro poslední operaci funkce *sat* znamená součet se saturací. Poslední dvě instrukce pracují po bytech. Tabulka převzata z [15] a upravena.

měnná je uvnitř třídy. Dále zde jsou proměnné pro každou část instrukce, těmi jsou opcode, použité registry, přímá hodnota, adresa a hodnoty shamt a funct určené pro řídící jednotku procesoru. Ve třídě jsou také constrainty které slouží pro randomizaci třídy. Jeden z constraintů řeší, aby typ instrukce byl randomizován před všemi ostatními hodnotami, ostatní constrainty pak vyplňují ostatní proměnné na základě vygenerovaného typu. Pro studenty nepovinnou, nýbrž velice užitečnou při debugování, funkcí v třídě je její výpis. V mém podání tento task vypsal simulační čas jeho volání, typ instrukce a významné hodnoty pro tuto instrukci. Task tedy nevypisoval hodnotu adresy v případě kdy se jednalo o insturkci sčítání. Třída obsahuje jeden parametr, tím je počet bitů adresy do paměti dat. Parametr se používá ve spojení s instrukcemi načítání a ukládání do této paměti a je jím omezena hodnota

vygenerovaného offsetu pro tyto instrukce.

Poslední modul, který bylo potřeba vytvořit byl driver. Ten měl za úkol generovat instrukce podle typu testu. Jak bude finální driver vypadat, je ponecháno na studentovi, důležité bylo aby splňoval následující podmínky:

- Z třídy instrukce složit zakódovanou instrukci,
- Ze zakódované instrukce vyčíst typ instukce a operandy, případně ji celou rozkódovat,
- Vyplnit programovou paměť náhodnými instrukcemi podle typu testu
  - Použít jen instrukce ADDI,
  - Použít jen instrukce využívající ALU,
  - Použít pouze instrukce skoku,
  - Vygenerování smyčky (pomocí instrukcí ADDI, SLT, BEQ),
  - Použít pouze instrukce JAL a JR (ekvivalenty instrukcí call a return),
  - Použít všechny známé instrukce.
- Na základě vygenerovaných instrukcí posílaných do procesoru udržovat hodnoty ve své kopii registrů,
- Průběžně kontrolovat rovnost vlastní kopie registrového pole s tím v procesoru.

V mém podání tato komponenta tedy obsahuje instanci třídy reprezentující instrukci, paměť pro instrukce a stínovou kopii registrového pole procesoru. Driver obsahuje tasky, které umožňují vyplnit paměť instrukcí všemi způsoby které jsou vypsány výše, v případě generování smyčky task potřebuje znát počáteční adresu smyčky, počet instrukcí ve smyčce včetně instrukcí tvořící cyklus a dva registry použité pro cyklení. V případě, že se objeví problém bránící vytvoření smyčky, jako třeba délka cyklu je menší než počet instrukcí potřebných k vytvoření nebo použití nultého registru, smyčka se nevytvoří. Driver obsahuje dva tasky pracující s instancí instrukce, jeden z 32 bitové proměnné vyplní informace do instance třídy, druhý task naopak z informací této instance vyplní 32 bitovou proměnnou reprezentující tuto instrukci. Dále jsou zde tasky které posílají do procesoru instrukce, tasky měnící a kontrolující registrové pole a task pro kontrolu změn programového čítače procesoru. Tento task se volá po každém odeslání instrukce do procesoru, task čeká do změny programového čítače anebo 100 ns simulačního času, podle toho co nastane dřív. V případě změny čítače se pokračuje bez problému dál, pokud uběhlo 100 ns simulačního času, tak se kontroluje hodnota program counteru. Jestliže je na nejvyšší možné hodnotě, test úspěšně proběhl, v opačném případě to znamená zacyklení procesoru, test je ukončen a je vypsána instrukce, na které se chyba vyskytla.

7.2.3 Testbench 2

Druhou verzí bylo pro ten samý procesor vytvořit testbench za pomocí bloků z knihovny UVM. Stejně jako předchozí verze úlohy, i zde byly stanoveny podmínky, které testbench musel splnit. Vzhledem k podobnosti zadání jsem zvolil podobný přístup k řešení této úlohy. Nejprve jsem si vygeneroval kódy pro všechny potřebné komponenty testbenche. V těchto komponentách jsem nastavil vše tak, aby se testbench správně vytvořil a propojil. Dalším krokem bylo přetvořit testbench tak, aby otestoval vytvořený procesor. Obvykle se tak dělo přesouváním tasků z minulé úlohy do komponent UVM testbenche.

Třída reprezentující instrukci se přeměnila na uvm_object, členské proměnné i constrainty zůstaly stejné, navíc přibyla proměnná pro programový čítač a zakódovanou instrukci. Objekt se tak bude moct dát použít jak pro vytváření testovacích instrukcí tak pro tranksakci vytvořenou monitorem. Task pro výpis bylo potřeba přeměnit, aby využíval funkci *do_print* knihovny. Taktéž bylo potřeba přidat makro pro registraci do továrny a konstruktor.

Funkce pro vytvoření instrukce z instance třídy se přesunula do uvm_driveru, který takto vytvořenou instrukci posílal do procesoru. Task pro dekódovaní instrukce se přesunul do uvm_monitoru, který také z dat vycházejících z procesoru vytváří transakci. Ta obsahuje hodnotu programového čítače, zakódovanou instrukci a informace skrývající se v této instrukci. V komponentě uvm_scoreboard se nalézá stínová kopie registrů a provádí se zde kontrola rovnosti registrových polí. Ostatní komponenty neobsahovaly nic zajímavého tykajícího se tohoto projektu.

Při následném sepisování zadání pro tuto úlohu jsem se snažil o co nejpodrobnější popis výsledku, pokusem o co nejjasnější zadání jsem ale spíše vytvořil postup. Zadání prochází postupně komponenty testbenche a popisuje co se uvnitř nachází, co do které komponenty patří a v jaké funkci to má být. Bohužel, při pohledu zpětně je jak v zadání tak i v řešení několik chyb, takže je dobře, že zadání nebylo použito v předmětu jako úloha pro studenty. V případě, že student neměl hotový procesor z bakalářského studia by dle mého názoru byla úloha i docela časově náročná. I když to může být jen domněnka z důvodu toho, že mi vytvoření zadání a vzorového řešení trvalo nejdéle a povedlo se z těch tří úloh nejméně.

Závěr

Cílem této práce bylo nastudovat konstrukce jazyka SystemVerilog a knihovny UVM, s těmito znalostmi následně vytvořit nápomocný text s doprovodnými zdrojovými kódy věnující se těmto konstrukcím. Dalšími cíli bylo otestovat podporu nástrojů pro jazyk SystemVerilog a knihovnu UVM a vytvoření vzorových úloh pro předmět Simulace číslicových obvodů.

V první části práce jsem vytvořil jednoduchý úvod do jazyka SystemVerilog pro čtenáře kteří používají spíše jazyk VHDL. Dále jsem vytvořil popis některých konstrukcí, které SystemVerilog přináší oproti Verilogu. Tento popis je občas doplněn o příklady kódu. Následně vznikl popis komponent knihovny UVM. Součástí této kapitoly byl úvod do používání TLM v kódu, krátký popis, co je továrna a k čemu slouží. Pak už jsem se věnoval jednotlivým komponentám testbenche, kde jsem obvykle zmínil co má komponenta za úkol, které členské funkce jsou důležité a co by měly obsahovat.

V druhé části práce jsem se věnoval nástrojům pro vytváření a simulování kódu v SystemVerilogu. Kapitolu jsem rozdělil na textové editory a simulátory. V textových editorech jsem popsal práci pár vybraných programů ve kterých jsem si vyzkoušel vytvořit zdrojový kód, na základě této zkušenosti jsem pak program ohodnotil, jak se mi s ním pracovalo, případně silné a slabé stránky práce s ním. U simulátoru jsem byl postup obdobný, zde jsem se pokusil odsimulovat jednoduchý projekt obsahující kód v jazyce SystemVerilog a komponenty knihovny UVM. Následně jsem v práci shrnul své pocity při práci s těmito nástroji a jejich nedostatky.

V poslední části práce popisuji jak vznikal pomocný text s doprovodnými kódy a vzorové úlohy. Zmiňuji zde jak jsem postupoval, co jsem zkusil a případně na jaké problémy jsem během vzniku narazil. Výsledkem této části jsou tři vzorové úlohy pro předmět Simulace číslicových obvodů, jedna z nich se ukázala moc komplikovaná pro použití, ale úloha Síťový přepínač se v předmětu použila v letním semestru v tomto roce. Pomocné texty se vzorovými kódy se stále upravují, jejich současná verze je k nalezení na git stránce

projektu [14].

V době odevzání vznikla základní verze textů, ta bude dále vylepšována na základě zpětné vazby od lidí, kteří text reálně použijí. V budoucnu by se na tuto práci dalo navázat aktualizací textů ve chvíli, kdy vyjdou nové standardy jazyka SystemVerilog anebo knihovny UVM. V případě velkých změn bude potřeba aktualizovat i vzorové úlohy, případně vymyslet úplně nová zadání.

[14] V době vzniku práce `https://gitlab.fit.cvut.cz/kallumir/mi-sce`

Literatura

[1] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, 2018: s. 1–1315, doi:10.1109/IEEESTD.2018.8299595.

[2] Harvey, M.: *Input and output skew timing for cb_axi clocking block*. Jan 2017. Dostupné z: http://www.markharvey.info/rtl/clkblk_08.01.2017/clkblk_08.01.2017.html

[3] Aynsley, J.: Jun 2010. Dostupné z: https://www.doulos.com/knowhow/systemverilog/uvm/uvm-verification-primer/

[4] UVM TLM. Dostupné z: https://www.chipverify.com/uvm/tlm-preface

[5] *TLM1 Interfaces, Ports, Exports and Transport Interfaces*. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files2/tlm1-txt.html

[6] UVM TLM Nonblocking Put Port. 2015. Dostupné z: https://www.chipverify.com/uvm/uvm-tlm-nonblocking-put-port

[7] UVM Factory. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files/base/uvm_factory-svh.html

[8] UVM Sequence. Mar 2020. Dostupné z: https://verificationguide.com/uvm/uvm-sequence/#UVM_Sequence_macros

[9] UVM Common Phases. Dostupné z: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1c/html/files/base/uvm_common_phases-svh.html

[10] Kohlík, M.: Universal Verification Methodology. https://courses.fit.cvut.cz/MI-SIM/media/lectures/11-UVM.pdf, Mar 2021.

[11] Promocrat: We are creative, dynamic, and friendly professionals with extensive experience in hardware design and verification. Dostupné z: https://www.amiq.com/about

[12] Hiiaka: hiiaka/suni_uvm_for_subl. Jul 2014. Dostupné z: https://github.com/hiiaka/suni_uvm_for_subl

[13] Douša, J.; Kohlík, M.: UVM (Universal VerificationMethodology). https://courses.fit.cvut.cz/NI-SIM/@B181/media/lectures/13/13-knihovna-uvm-2017.pdf, 2018.

[14] Štěpánovský, M.; Tvrdík, P.: Návrh jednocyklového RISC procesoru. https://courses.fit.cvut.cz/BI-APS/@master/media/lectures/BI-APS-Prednaska04-SingleCycleCPU.pdf, 2020.

[15] Štěpánovský, M.: Semestrální projekt č.1: Jednocyklový procesor. Dostupné z: https://courses.fit.cvut.cz/BI-APS/@B181/tutorials/05/semester_project_cz.html

Příloha A

Seznam použitých zkratek

HDL Hardware description language

OOP Object-oriented programming

UVM Universal Verification Methodology

OVM Open Verification Methodology

TLM Transaction Level Modeling

DUT Design under test

Příloha **B**

Obsah přiloženého CD

- `DP_Miroslav_Kallus_2021.pdf` ..... text práce ve formátu PDF
- `src`
  - `DP_Miroslav_Kallus_2021_src` zdrojová forma práce ve formátu LaTeX
  - `kod_ulohy` ..... zdrojové kódy k vzorovým úlohám
    - `SIM-CPU_tester` ..... zdrojové kódy k testeru 1 CPU
    - `SIM-UVM_CPU_tester` ..... zdrojové kódy k testeru 2 CPU
    - `SIM-packet_sorter` ..... zdrojové kódy k síťovému přepínači
  - `kod_texty` ..... doprovodné zdrojové kódy k textům
  - `UVM_test` . projekt k otestování podpory nástrojů pro knihovnu UVM