Tento článek je původním rukopisem textu publikovaného v časopise DPS Elektronika A-Z: J. Šťastný. Verifikace pomocí assertions: jak začít, DPS Elektronika od A do Z, no 4, pp. 4-9, 2013.

Verifikace pomocí assertions: jak začít?

Jakub Šťastný
ASICentrum, s.r.o.
FPGA Laboratoř, Katedra teorie obvodů FEL ČVUT Praha

1 Úvod

Předchozí články [1,2,3] představily jak základní koncepci, tak výhody nasazení verifikace pomocí assertions (*Assertions Based Verification*, ABV) spolu se základy jazyka PSL a řadou případových studií. V posledním dílu seriálu ukážeme, jak s ABV začít. Budeme se věnovat metodice použití assertions a dotkneme se také základních přístupů k verifikaci číslicových obvodů. Předkládaná pravidla a rady jsou z části získána z literatury citované v závěru příspěvku (zejména [4,5,6,7]), z části pak pochází z autorových vlastních zkušeností.

2 Nasazení do praxe

Nejsložitějším krokem při nasazení ABV do každodenního použití je naučit se na assertions myslet už v průběhu tvorby HDL kódu a správně identifikovat klíčové vlastnosti návrhu, které je nutné kontrolovat. Rychlejšímu přijetí a snažší integraci ABV by čtenáři mohla pomoci následující pravidla.

Identifikujte bloky, které často používáte. Víme, že jsou ideálními kandidáty pro assertions. V blocích určených k opětovnému použití používejte assertions co nejintenzivněji. Jejich použití se tak nejvíce vyplatí, protože mohou ověřit správnou integraci bloku a zabránit chybám v časné fázi návrhového cyklu.

Najděte/definujte podmínky pro správnou funkci jednotlivých bloků; to obnáší mimo jiné potřebu nalézt odpovědi na tyto otázky:

1. **Očekává blok specifickou sekvenci signálů na svém rozhraní?** Typickým příkladem může být blok děličky prezentovaný v [3]. U děličky má například smysl kontrolovat, že před každým startem dělení jsou do bloku zapsány dělenec a dělitel. Implementovaný assertion pak ověří správnou integraci bloku do okolního systému.
2. **Používá blok nějaký standardní protokol na svém rozhraní?** Pokud ano, zjistěte zda je pro něj dostupný automatický „*checker*“. Pokud použijete standardní sběrnice a protokoly, existují pro ně často už předpřipravené checkery a monitory, které kromě kontroly správné podoby transakcí na sběrnicích také měří funkční pokrytí a hledají výskyt mezních stavů (viz [4] – monitor AMBA sběrnice, ale dostupný je i například *assert_handshake* v [8]). Implementovaný assertion zde bude ověřovat správnou integraci bloku do okolního systému.
3. **Mají mít vnitřní signály bloku specifické časování?** Pěkným příkladem může být například použití synchronizátoru asynchronních signálů tvořeného dvěma registry v sérii (viz [9], kapitola 11) uvnitř bloku. Synchronizátor by neměl mít na vstupu pulzy kratší, než je dvojnásobek periody hodinového signálu a je ideálním místem pro implementaci assertion. Implementovaný assertion pak bude ověřovat bezchybnost implementace bloku.
4. **Jsou v bloku sběrnice, na kterých se nesmí objevit určité hodnoty, nebo jsou na nich hodnoty kódované specifickým kódem? Očekáváte na sběrnicích nějaké přesně definované sekvence hodnot?** Příkladem může být například použití binárního čítače modulo 5. Takový blok bude mít tříbitový výstup na kterém se nesmí objevit hodnoty 101_2, 110_2 a 111_2 a po hodnotě 100_2 se na výstupu musí objevit jako další stav 000_2. Je snadné implementovat assertion kontrolující správnou sekvenci stavů a opět se bude jednat o ověření bezchybné implementace bloku.
5. **Konečně, za jakých podmínek je třeba navržené assertions blokovat**? Musí být příslušné assertions splněny při resetu bloku? Správné blokování assertions nelze podceňovat, zabrání falešným chybovým hlášením za běhu simulace a ušetří tak čas návrháři při ladění návrhu.

Assertions pište hned. Výhody assertions vyniknou jen pokud jsou použity od samého začátku implementace RTL kódu; pište proto assertions dohromady s RTL kódem, který ověřují a **pište je hned**. První RTL kód nikdy nefunguje a assertions implementované na samém začátku návrhových prací mají největší přínos pro produktivitu práce. Naproti tomu implementace assertions do existujících a použitých bloků nepřinese takové výhody, neboť příslušné bloky jsou typicky dobře odladěné dlouhým používáním [5].

Připojujte assertions ke komentářům. Jako vodítko k implementaci assertions je šikovné použít komentáře zapisované do HDL kódu. Komentáře typu „*tady se nikdy nestane, že…*“ a „*tady vždycky nastane, že*“ přímo volají po

připojení assertion [5]. Na druhou stranu, implementovanými assertions nemá smysl duplikovat elementární fukcionalitu RTL návrhu; například zápis

```
x_d <= x_q + 1;
```

bude vždy fungovat správně, nemá tedy smysl pomocí assertions kontrolovat zda se *x_d* opravdu rovná *x_q+1*.

Zamýšlejte se nad chybami ve Vašem návrhu. Analyzujte chyby, které jste objevili ve Vašem návrhu, a doplňujte assertions, které zabrání jejich opakování. Poučte se z chyb a při příštím návrhu podobné číslicové struktury vložte assertion kontrolující správnost implementace ihned.

Používejte knihovny assertions. Nejlépe je začít přímo používat knihovnu OVL [8]. Pokud si budete psát vlastní assertions, často používané assertions implementujte v samostatných entitách/blocích a použijete je stejným způsobem, jako je tomu u knihovny OVL. A budete-li používat například PSL, lze s výhodou použít parametrizované konstrukce.

Implementaci assertions plánujte. Autorovi tohoto příspěvku se osvědčilo k tomu používat tabulky, jako je tabulka 1 (koncepce je převzata z publikace [6] a upravena na základě používání). Udržujte si formální plán toho, co se má verifikací pokrýt a revidujte jeho naplnění v průběhu projektu; pamatujte na pravidlo 5P – *Prior Planning Prevents Poor Performance*.

Pojmenovávejte všechny assertions i vlastnosti. Používejte konzistentní jmennou konvenci [6]. Usnadní to orientaci v návrhu a případné odkazování na assertions v dalších skriptech. Vhodnou konvencí může být například *asrt_xxxx*, *cover_xxxx*, *prop_xxxx* pro pojmenování kontrolních assertions, assertions pro měření pokrytí a definice vlastností.

Jméno	Popis	Stav	Typ	Návrhář
cover_fifo_full	Detekce plného FIFA.	tbd	coverage	Matěj
cover_fifo_empty_rd	Detekce výskytu čtecí operace z prázdného FIFA (nic se nesmí stát).	tbd	coverage	Matěj
asrt_sys_clk_req	Zkontroluj, že *sys_clk* signál mění svoji hodnotu jen pokud je signál *sys_clk_reg*='1'. **Assertion je zakázán**: resetem bloku	✓	assert	Jakub
asrt_sys_dac_data	Vstupní sběrnice *sys_dac_data* musí být stabilní alespoň 3 hodinové cykly před sestupnou hranou *dig_dac_follow*. **Assertion je zakázán**: resetem bloku	✓	assert	Matěj

Tabulka 1: Příklad plánu assertions pro jeden blok v návrhu. Tabulku je možné buď udržovat pro každý blok zvlášť v jeho specifikaci, nebo jako jeden dokument v tabulkovém kalkulátoru kde bude plán všech assertions v obvodu; pak ji lze i filtrovat po blocích, atp.

3 Implementace assertions

Na jednu stranu, pro implementaci assertions je praktické mít simulátor s jejich podporou; jen tak lze využít plného potenciálu této technologie. Na druhou stranu, pořízení takového nástroje není zanedbatelná investice a managementu většiny firem zaměstnávajích čtenáře asi nepřijde jako dobrý nápad utrácet peníze, pokud není užitečnost ABV pro práci návrhářů zcela jistá.

Chcete-li začít s assertions pracovat, můžete s nimi začít experimentovat tak, že je budete implementovat v jazyce, který používáte pro RTL návrh. Implementace assertions v HDL jazyce vyžaduje o málo více času a ani ladění nebude tak komfortní, avšak autor tohoto příspěvku se i přesto na základě svých zkušeností domnívá, že v konečném výsledku dojde k úspoře času. Například v jazyce VHDL lze jednoduché kombinační assertions implementovat snadno a rychle pomocí příkazu *ASSERT*, složitější potom pomocí dedikovaných procesů zapsaných na behaviorální úrovni na co nejvyšší úrovni abstrakce; nejlepší je ovšem v maximální možné míře využít předpřipravených assertions v knihovně OVL [8]. V tabulce 2 jsou shrnuty jednotlivé výhody verifikace pomocí assertions, jak jsou uvedeny v článku [1] a je uvedeno, které z nich lze využít i bez specializovaných nástrojů.

Assertions snadno umožňují...		Poznámka
... kontrolovat očekávané chování na vstupech a výstupech bloku.	✓	Bez omezení, jen mírně časově náročnější implementace assertions.
... kontrolovat očekávané chování logických struktur uvnitř obvodu.	✓	Bez omezení, jen mírně časově náročnější implementace assertions.
... monitorovat chování bloku a počítat výskyty definovaných podmínek, měřit funkční pokrytí.	✓	Bez použití simulátoru s podporou pro assertions není podpora pro automatické generování zpráv o funkčním pokrytí.

... sloužit jako komentář k funkci bloku a pomoci jeho pochopení.	✓	Bez omezení, jen mírně časově náročnější implementace assertions.
... sloužit jako komentář k funkci bloku, který upozorňuje na potřebu ještě něco dokončit.	✓	Bez omezení, jen mírně časově náročnější implementace assertions.
... poskytnout informace nástroji pro formální verifikaci.	✗	Nástroje pro formální verifikaci dokážou zpracovat jen assertions ve specializovaných jazycích.
... pomoci nalézt chyby v systému v prototypu.	✗	Konverze behaviorálního popisu assertion do číslicového obvodu v emulátoru není možná.

Tabulka 2: Využitelnost jednotlivých výhod použití assertion bez simulátoru s podporou pro specializované jazyky pro zápis assertions.

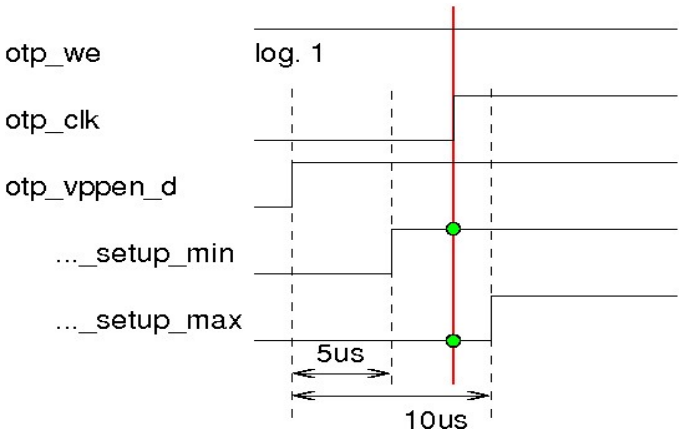
Implementace kontrolních assertions je snadná, jen implementujete v HDL jazyce nesyntetizovaný kód v příslušném bloku, který tyto podmínky kontroluje a vyhlásí chybu pomocí příkazu *ASSERT* kdykoliv jsou porušeny. Kód obklopte direktivami *translate_off* a *translate_on*. Příklad takového assertion je v rámečku 1; zde se jedná o assertion kontrolující správnou funkci čítače do 255 se saturací. Všimněte si, že stav čítače se v každém cyklu buď nezmění, zvětší o jedničku, resetuje do 0x00, nebo je trvale saturován na 0xFF.

1

```
--pragma translate_off
sample_cnts_reg : PROCESS(mcu_res, mcu_clk)
BEGIN
  IF (mcu_res = '1') THEN
    sys_cnt_a    <= (OTHERS => '0');
  ELSIF (mcu_clk'EVENT AND mcu_clk = '1') THEN
    sys_cnt_a    <= sys_cnt;
    ASSERT ((unsigned(sys_cnt) = unsigned(sys_cnt_a))   OR
            (unsigned(sys_cnt) = unsigned(sys_cnt_a)+1) OR
            (sys_cnt=X"00") OR
            (sys_cnt=X"FF"))
    REPORT "Event counter is not uniformly advanced"
    SEVERITY ERROR;
  END IF;
END PROCESS sample_cnts_reg;
--pragma translate_on
```

Zde poznamenejme, že ani při použití specializovaného jazyka se někdy nelze vyhnout podpůrným konstrukcím v HDL jazyce. Například v příkladu v rámečku 2 ověřujeme zda mají řídící signály na rozhraní OTP paměti správné časování, konkrétně že předstih[1] zapnutí vysokého napětí pro programování paměti – náběžná hrana signálu *otp_vppen_d* – před hranou hodin *otp_clk* zapisující data je mezi 5 a 10 µs, viz také obrázek 1. S výhodou je zde využita kombinace s HDL jazykem pro generování zpoždění, všimněte si opět použití direktiv *translate_on/off*.

[1] Zkušenější čtenáři zde mohou namítnout, že v tomto případě lze s výhodou použít Vital procedury pro kontrolu časování; to je pravda. Kód byl vybrán proto, že je to jednoduchý ilustrační příklad příslušného konceptu.



Obrázek 1: Sekvence signálů na rozhraní OTP paměti ověřovaná pomocí assertion. Červená čára označuje okamžik aktivace assertion, zelené body jsou kontrolované hodnoty signálů. Soubor otp_vpp_timing.eps.

2

```
-- pragma translate_off
otp_vppen_d_setup_min <= otp_vppen AFTER  5 us;  -- Req. 104.10
otp_vppen_d_setup_max <= otp_vppen AFTER 10 us;  -- Req. 104.11
-- pragma translate_on
-- psl assert_vppen_setup : assert always ( (otp_we='1' AND otp_ck'EVENT
AND otp_ck='1') -> (otp_vppen_d_setup_min='1' AND otp_vppen_d_setup_max='0'))
-- report "DUT: VPPEN setup time violation. Req. 104.10. and 104.11";
```

Poněkud složitější je v HDL jazyce implementace assertions pro měření funkčního pokrytí (*coverage assertions*). Uživatelé specializovaného jazyka pro psaní assertions zde jsou ve výhodě, protože ten podporuje automatické počítání výskytů definovaných událostí. Při implementaci funkčního pokrytí v HDL jazyce je možné v zásadě postupovat více způsoby. Nejjednodušší je použít konstrukce jako pro kontrolní assertions, jen označit výpisy do logu jako „Note“ - poznámka – a vložit do nich charakteristický řetězec (zde *COV000*), například takto:

```
ASSERT (clock_enabled'EVENT and clock_enabled='1')
REPORT "COV000: clock enable active"
SEVERITY NOTE;
```

Po ukončení simulace lze pak zjistit počet výskytů události jednoduchou sadou příkazů zadaných na příkazové řádce operačního systému (zde uvedený příklad je pro Linux):

```
grep msim_simulation_log_file.log -i -e "COV000" | wc -l
```

Jinou, mírně elegantnější možností je postup pro implementaci funkčního pokrytí popsaný v prezentaci [7]. Konečně, poslední možností je využít podporu pro funkční pokrytí integrovanou v knihovně OVL [8].

Závěrem krátké diskuze o technikách implementace assertions si autor dovolí malou odbočku od tématu; během práce na návrhu je častou praxí, že návrhář umístí do HDL komentář typu

```
-- TODO: zde je potřeba dokončit řízení děličky
```

Autorovi se v praxi osvědčilo používání spustitelných „TODO“ komentářů implementovaných pomocí příkazu *ASSERT*, viz rámeček **3**.

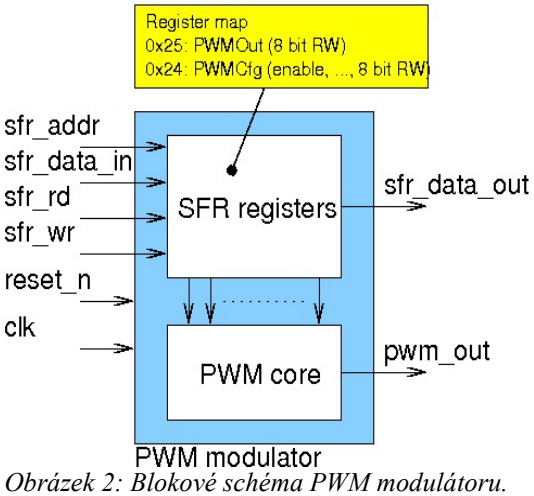
3

```
ASSERT (false)
REPORT "TODO: zde je potřeba dokončit řízení děličky"
SEVERITY WARNING;
```

Výhoda takového postupu je zřejmá; komentář se objevuje na začátku každé simulace a je tak na něj obtížné zapomenout. Je také dobrou praxí revidovat logy simulací před uzavřením návrhových prací; pak je daleko větší šance, že „TODO“ zůstane nepřehlédnuto, než když je zapsáno jako obyčejný komentář v kódu. Snižuje se tak riziko toho, že něco důležitého bude přehlédnuto a zůstane nedokončeno.

4 Verifikace pomocí náhodných stimulů

Představme si, že je naším cílem ověřit správnost implementace bloku pulzně šířkového modulátoru; jeho blokové schéma a základní specifikace je na obrázku 2. Modulátor je implementován jako periferie mikropočítačového systému. K řídícímu mikropočítači se připojuje pomocí SFR (*Special Function Registers*) sběrnice a je ovládán pomocí dvou registrů mapovaných do adresního prostoru na adresy 0x24 (konfigurační registr) a 0x25 (registr střídy výstupu PWM modulátoru).

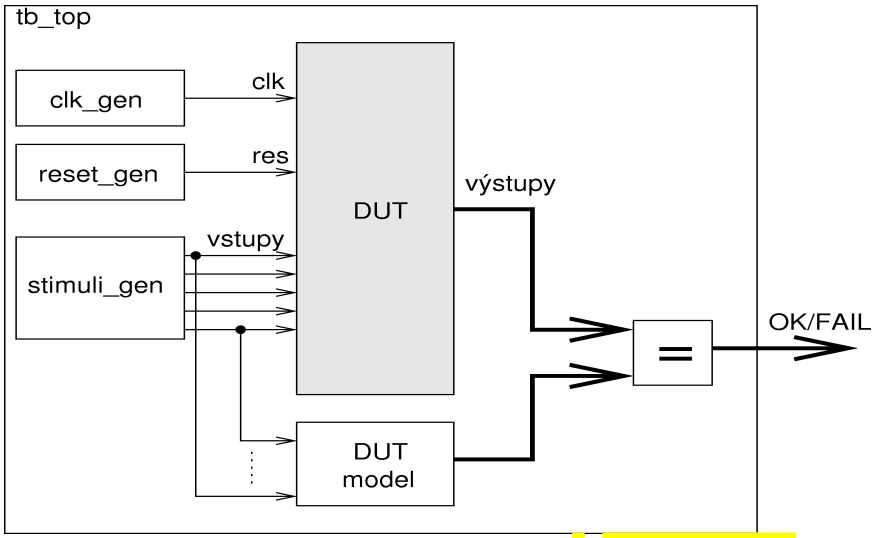


Obrázek 2: Blokové schéma PWM modulátoru.
Soubor pwm_modulator.eps.

Na obrázku 3 je rozkresleno typické blokové schéma verifikačního prostředí pro číslicový obvod. Šedivý blok *DUT* (*Design Under Test*) představuje verifikovaný obvod (zde PWM modulátor); blok *DUT model* je modelem verifikovaného obvodu a na svém výstupu se chová tak, jak by se měl blok *DUT* chovat v reakci na předkládané vstupy, pokud by byl zcela bez chyby. Bloky *clk_gen* a *reset_gen* slouží pro generování hodinového a resetovacího signálu, blok *stimuli_gen* generuje stimuly (průběhy vstupních signálů) pro DUT i model. Komparátor (=) na výstupu srovnává odezvy modelu a skutečného návrhu a pokud jsou odlišné, ohlásí chybu. Naznačená dekompozice do bloků je ideová, *DUT model* a komparátor často může být realizován jako distribuovaný – část může být obsažena například v assertions (ty představují model chování, proti kterému je skutečné chování návrhu porovnáváno), část může být realizována i přímo v generátoru stimulů. Pak budou výstupy bloku *DUT* zapojeny do bloku *stimuli_gen*, a v něm pak můžeme najít kód jako například:

```
send_transaction(write_to_register(16#24#), written_data(16#56#));
check_if_register(16#AA#,equals_to(16#56#));
```

Často používaná technika verifikace je verifikace pomocí deterministických stimulů (*directed testing*). Běžný postup zde spočívá v tom, že verifikační inženýr nadefinuje scénář chování (př. PWM modulátor je resetován, potom je povolen, do registru modulátoru se zapíše slovo 0x80, pak je zkontrolována střída na výstupu modulátoru...), pro scénář napíše kód, který generuje stimuly a ten je potom spouštěn jako test. Psaní stimulů realizujících konkrétní scénáře je časově náročná práce a není složité při ní udělat obtížně odhalitelnou chybu, která se projeví aplikací jiného scénáře, než je plánováno a tedy nedostatečným pokrytím funkcí obvodu testem. Kombinujeme-li funkční pokrytí s deterministickými stimuly, budou assertions užitečné zejména jako validace napsaných testů; lze pomocí nich zjistit, zda se opravdu na nějakou důležitou funkci systému nezapomnělo a zda jsou všechny scénáře implementovány jak mají.



Obrázek 3: Blokové schéma verifikačního prsotředí [9]. Soubor tb_top.tif.

Výhody použití assertions pro měření funkčního pokrytí vyniknou až při jejich kombinaci s verifikací založenou na náhodných stimulech. Verifikace pomocí náhodných stimulů (*Constrained Random Verification*, CRV) je vhodná pro složitější systémy se značným množstvím funkcí, které je možné vzájemně kombinovat téměř bez omezení. Čas nutný pro návrh a implementaci tradičních deterministických testů potom roste nade všechny meze. Při aplikaci CRV bude verifikační inženýr postupovat jinak.

1. Nejprve implementuje *DUT model* jako samostatný blok, zde model pulsně šířkového modulátoru. Model bude monitorovat vstupy do PWM bloku, podle nich nastaví svůj vnitřní stav. Dále bude generovat výstup *PWM* a *INT* s rozlišením jedné periody hodin; bude tzv. *cycle accurate*, k jeho implementaci nicméně bude použit HDL jazyk na behaviorální úrovni.
2. Za model je pak zařazen jako další blok komparátor výstupů návrhu PWM modulátoru a modelu PWM modulátoru, ke srovnání výstupů z *DUT modelu* i *DUT* bude docházet s náběžnou hranou hodin.
3. Do modelu budou vloženy assertions monitorující funkční pokrytí – zda se do registru *PWMOut* PWM modulátoru někdy nahraje hodnota 0x00 a 0xFF (extrémní případ z hlediska modulátoru, v ošetření mezních podmínek lze očekávat chybu) a zda se do registrů v PWM něco zapisuje a něco z nich čte.
4. Potom verifikační inženýr napíše jednoduchý test, který například ve smyčce 128 krát počká náhodný počet hodinových cyklů (min. 512, max 1024), s pravděpodobností 0.5 zapíše na adresu mimo adresní rozsah registrové mapy náhodnou hodnotu, jinak zapíše do náhodně vybraného konfiguračního registru náhodnou hodnotu. Protože na generované náhodné transakce klademe další omezující podmínky (například min. 512 cyklů, max. 1024 cyklů), aby vedly na „rozumné“ chování verifikovaného obvodu, mluvíme o *constrained random verification*.

Vlastní verifikace probíhá tak, že je test spuštěn a po jeho proběhnutí jsou zkontrolovány assertions monitorující pokrytí. Pokud nejsou pokryty všechny stavy, test prodloužíme – implementací více smyček, případně upravíme generování náhodných čísel. Pro složitější nedosažené mezní stavy můžeme na závěr ručně připsat stimuly pro jejich pokrytí.

Výhody uvedeného postupu jsou zřejmé: není třeba vymýšlet stimuly, generují se samy pomocí náhodných čísel. V zásadě obětujeme strojový čas (dobu po kterou běží simulace) a nahradíme složitou a časově náročnou lidskou práci delší simulací. Použití náhodně generovaných testů částečně odstraní manuální práci nutnou pro psaní deterministických testů, sníží se tak i množství chyb v testech a funkční pokrytí zajistí lepší monitorování procesu verifikace. Obyčejně je testů třeba napsat i méně, než pokud je obvod verifikován čistě deterministickým přístupem. Konečně, dobře implementovanou randomizací lze také dosáhnout lepšího pokrytí funkcí DUT (pokryjí se často kombinace funkcí, které by verifikační inženýr při psaní testů nevymyslel) a výsledné verifikační prostředí a testy bude lépe přenositelné mezi návrhy.

Nevýhody zde představuje potřeba mírně složitějšího simulačního prostředí a větší počáteční investice do jeho budování. Ta se ale v dalších fázích návrhu rychle vrátí; pro větší návrhy je verifikace pomocí náhodných stimulů jednoznačně lepší postup, než verifikace pomocí deterministických stimulů. Zdánlivou nevýhodou aplikace náhodných stimulů je to, že test běží pokaždé jinak. To lze ovšem ovlivnit nastavením semínka pro generátor náhodných čísel na začátku testu. Co se tím nicméně nevyřeší je, že každá modifikace testu změní sekvenci pseudonáhodných čísel a tím získáme diametrálně odlišné chování testu. Většinou to ale nevadí, protože pokrytí funkcí je zajištěno monitorováním funkčního pokrytí. Navíc variace v chování testu mohou najít nečekané problémy; nakonec je potenciální „nestabilita“ testů ve spojení s regresním testováním spíš výhoda, a číslicové simulátory proto mají možnost automatického nastavení semínka pro generování náhodných čísel tak, aby každý běh simulace byl unikátní. Nepříjemnou nevýhodou

CRV je, že v obvyklých HDL jazycích (VHDL, Verilog) je generování náhodných čísel splňujících řadu omezení složitější na naprogramování. Zde může částečnou úsporu práce přinést znovupoužití funkcí pro generování náhodných čísel ve formě knihovny. Konečnou úlevu přinese použití jazyka SystemVerilog pro implementaci testů a verifikačního prostředí; jazykové konstrukce v SV umožňují snadno generovat náhodná čísla a posloupnosti se zadanými vlastnostmi. Zajímavý příklad aplikace CRV na verifikaci složitějšího mikroprocesoru lze nalézt v článku [10].

5 Závěr

Článek shrnul základní rady, které by měly čtenáři ulehčit zavedení ABV do každodenní návrhářské praxe a ukazuje, že ABV lze nasadit i bez specializovaných nástrojů a studia dalšího jazyka pro návrh číslicových obvodů. Použití běžného HDL jazyka pro návrh assertions umožní návrhářům alespoň částečně využít přínosy ABV a při systematické aplikaci assertions časem zjistí, že díky nim opravdu nachází chyby rychleji a jsou schopni nalézt problémy, které by jinak bylo obtížné objevit. V tomto okamžiku pak bude patrně správný čas na přechod na sofistikovanější nástroje (PSL/SVA s odpovídajícím simulátorem – např. QuestaSim/ModelSim DE).

Celý seriál o ABV je tak uzavřen textem o tom, jak začít. Paradox je to nicméně jen zdánlivý; zatímco autor teď může chvíli odpočívat s pocitem dobře vykonané práce, čtenář je teprve na počátku nové cesty. Autor textu tiše doufá, že publikované informace čtenáři umožní realizovat stále větší a složitější číslicové systémy s alespoň ne příliš zvýšeným úsilím.

6 Použitá a doporučená literatura

[1] Jakub Šťastný. Verifikace pomocí assertions: seznámení. DPS Elektronika od A do Z, číslo 6, pp. 4 – 8, 2012.

[2] Jakub Šťastný. Verifikace pomocí assertions: jazyk PSL. DPS Elektronika od A do Z, číslo 2, pp. 30 – 34, 2013.

[3] Jakub Šťastný. Verifikace pomocí assertions: případové studie. DPS Elektronika od A do Z, číslo 3, pp. 10 – 13, 2013.

[4] Ping Yeung. Assertion based verification of ARM-core based designs. Information Quaterly, Vol. 3, No. 5, 2004. http://iqmagazineonline.com/magazine/pdf/v_3_5_pdf/pg46-49_0-in_sertion-Artic.pdf [kontrolováno 30.4.2013]

[5] Harry Foster, Adam Kolnik, David Lacey. Assertion Based Design, 2nd edition. 2004 Kluwer Academic Publisher.

[6] Don Mills, Stuart Sutherland. System Verilog Assertions are for design engineers, too! In SNUG San Jose 2006.

[7] Jim Lewis. Constrained Random Verification with VHDL, SynthWorks VHDL trainings. http://www.synthworks.com/downloads/ConstrainedRandom_SynthWorks_2012.pdf [kontrolováno 30.4.2013]

[8] Open Verification Library. http://www.eda.org/ovl/ [kontrolováno 30.4.2013]

[9] Jakub Šťastný. FPGA prakticky, BEN Praha 2010.

[10] Jason C. Chen. Applying CRV to microprocessors. EE Times-India. December 2007. http://eetimes.com/design/eda-design/4018518/Applying-Constrained-Random-Verification-to-Microprocessors [kontrolováno 30.4.2013]