Tento článek je původním rukopisem textu publikovaného v časopise DPS Elektronika A-Z: J. Šťastný. Verifikace pomocí assertions: jazyk PSL, DPS Elektronika od A do Z, no 2, pp. 30-34, 2013.

Verifikace pomocí assertions: jazyk PSL

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

1 Úvod

Z předchozího příspěvku [1] víme, že pro implementaci assertions jsou používány speciální jazyky. Důvodem k jejich použití je snadnější a srozumitelnější zachycení chování verifikovaného číslicového obvodu; v neposlední řadě i jednodušší a rychlejší zápis jednotlivých vlastností, které jsou pomocí assertions zachyceny. Zatímco text [1] byl zaměřen na teoretický popis koncepce, výhod a nevýhod assertions, v tomto článku se budeme věnovat praktičtějším stránkám věci, konkrétně základům jazyka PSL (*Property Specification Language*). PSL byl zvolen proto, že jeho použití je velmi snadné a nepředstavuje tak velký myšlenkový krok pro uživatele VHDL jako je přechod na System Verilog a System Verilog Assertion.

Jazyk PSL bude pro čtenáře znalého běžných programovacích jazyků nezvyklý, protože jeho návrh i filozofie je podřízena snadnému zachycení sekvence stavů navrhovaného systému abstraktním způsobem. PSL navíc nikdy není používán samostatně, ale jeho konstrukce jsou vždy jen doplňkem konstrukcí standardně používaných HDL jazyků. Jelikož není zamýšlen k „samostatnému“ použití, neobsahuje například konstrukce pro definice struktury návrhu, modulů nebo procedur ve smyslu, jaký známe z HDL jazyků.

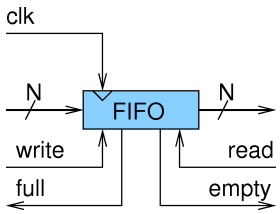
Jazykové konstrukce v PSL jsou do kódu hostitelského HDL jazyka integrovány ve formě komentářů. Na návrh číslicových systémů jsou ovšem běžně používány jazyky VHDL a Verilog na RTL úrovni; jejich syntaxe se značně liší. Zatímco jazyk VHDL vychází z jazyka ADA a je vzdáleně podobný Pascalu, Verilog má spíše „céčkový“ vzhled a vlastnosti. Proto má i jazyk PSL různé varianty podle hostitelského jazyka; my se budeme – vzhledem k preferenci VHDL v Evropě – zabývat VHDL alternativou PSL. Samotné PSL se také snaží v nejvyšší možné míře používat konstrukty hostitelského jazyka a dostupné standardní funkce.

PSL je standardizován jako IEEE standard 1850, *Standard for PSL: Property Specification Language*. Do verze 1.1 byl vyvíjen *Functional Verification Technical Committee* organizace Accellera [2,3], která ho posléze předala do správy IEEE. Zcela původní koncepce PSL pochází z firmy IBM, originální jazyk se jmenoval *Sugar* a pod tímto jménem ho lze stále ještě potkat ve starších publikacích (i jako *Sugar/PSL*).

2 Jednoduché logické výrazy

Nejdůležitější jednotkou při psaní assertion je tzv. vlastnost (*property*). Vlastnost zachycuje nějaké požadované chování systému. Jednoduchým příkladem může být například zápis, který říká, že do plného FIFA se nesmí nikdy zapisovat (tedy, že nikdy nesmí nastat situace, kdy je signál *full*='1' – FIFO paměť je plná – a současně *write*='1' – tedy je vyžadován zápis); pro větší názornost jsou jednotlivé části assertion odlišeny barvami:

```
-- psl asrt_full_write : assert never (write='1' AND full='1') report "Write to a full FIFO!" @rising_edge(clk);
```



Obrázek 1: Blok FIFO paměti - vstupy a výstupy. Soubor fifo.tif.

Assertion se skládá z těchto členů:

- Text *-- psl* indikuje, že bude následovat PSL assertion. Všimněte si, že celý assertion je zapsán jako komentář v

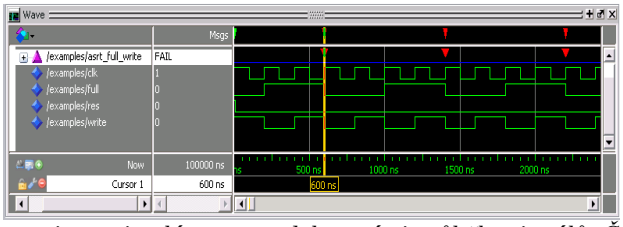
kódu v jazyce VHDL.

- *asrt_full_write* je návěští, kterým je assertion označen. Návěští umožňuje snadnou identifikaci assertion v logu simulátoru i v okně s vlnkami, viz obrázek 2.
- Verifikační direktiva *assert* vždy uvozuje assertion, který je tvrzením – tedy něčím, co je potřeba kontrolovat při simulaci. Druhá nejčastěji používaná direktiva je *cover*; ta instruuje simulační nástroj aby místo kontroly splnění následující podmínky počítal počet hodinových cyklů po které je vlastnost pravdivá.
- Další klíčové slovo *never* definuje, kdy má být kontrolovaná vlastnost splněna; v tomto případě je její neplatnost kontrolována za všech okolností a ve všech hodinových cyklech; pokud bude následující vlastnost pravdivá, bude vypsáno chybové hlášení. Kromě never lze také použít *always* – následující vlastnost musí být splněna v každém hodinovém cyklu a další, méně často užívaná klíčová slova, více viz [4].
- následuje vlastnost, která bude monitorována. Zde je to jen jednoduchý logický výraz *(write='1' AND full='1')* bez dalších časových vazeb. Díky použití klíčového slova *never* bude simulátor kontrolovat, zda je tento výraz nepravdivý ve všech hodinových cyklech, tedy zda současně nedochází k tomu, aby *write*='1' a *full*='1'.
- příkaz *report* definuje, co se vypíše do logu simulátoru v případě toho, že assertion zareguje na chybové chování.
- konečně, část *@rising_edge(clk)* definuje, že se vlastnost kontroluje vždy s náběžnou hranou hodin. Tak je zajištěno, že jsou kontrolovány vždy jen ustálené hodnoty monitorovaných signálů. Uvádět definici hrany hodin pro kontrolu u každé assertion je ovšem poněkud nepraktické; následující jazyková konstrukce umožní definovat implicitní hodiny pro všechny následující assertions v příslušném souboru se zdrojovým kódem:
  -- *psl default clock is rising_edge(clk)*

V obrázku 2 je uveden příklad vizualizace assertion v simulátoru *QuestaSim* [5]. VHDL kód příkladu spolu s vloženým PSL assertion je pro zvídavější čtenáře dostupný na WWW stránce [6]; simulátor porušení podmínky v assertion také oznámí v logu simulace:

*# ** Error: Write to a full FIFO!*

#Time: 600 ns Started: 600 ns Scope: /examples/asrt_full_write File: examples.vhd Line: 52



Obrázek 2: Příklad chování assertion v simulátoru s modelovanými průběhy signálů. Červený trojúhelníček označuje okamžik, kdy byla porušena platnost monitorované vlastnosti. Soubor fifo_waves.png

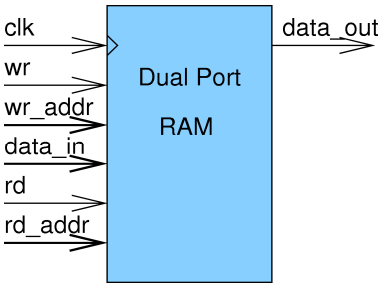
Pro lepší pochopení zápisu assertions bude uveden ještě jeden jednoduchý příklad. Představme si, že pracujeme s dvoubránovou pamětí typu RAM (viz obrázek 3), která nebude poskytovat správný výsledek, pokud je současně zapisováno do a čteno ze stejné adresy paměti. Pak je třeba kontrolovat, že popsaná situace v navrhovaném systému nikdy nenastane. Jeden z možných zápisů assertion je tento:

```
-- psl default clock is rising_edge(clk);
-- psl asrt_mem_rw : assert never ((mem_rd='1') AND (mem_wr='1') AND (wr_addr=rd_addr)) report "Reading and writing from/to the same address activated";
```

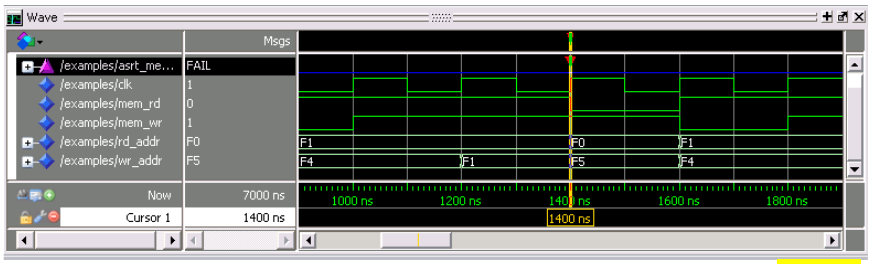
Na obrázku 4 je opět příklad chování assertion v simulátoru; v případě porušení vlastnosti simulátor vypíše chybové hlášení:

*# ** Error: Reading and writing from/to the same address activated*

#Time: 1400 ns Started: 1400 ns Scope: /examples/asrt_mem_rw File: examples.vhd Line: 7



Obrázek 3: Blok dvoubránové paměti, vstupy a výstupy.
Soubor dp_ram.tif.



Obrázek 4: Příklad chování assertion s modelovanými průběhy signálů. Soubor fifo_waves.png.

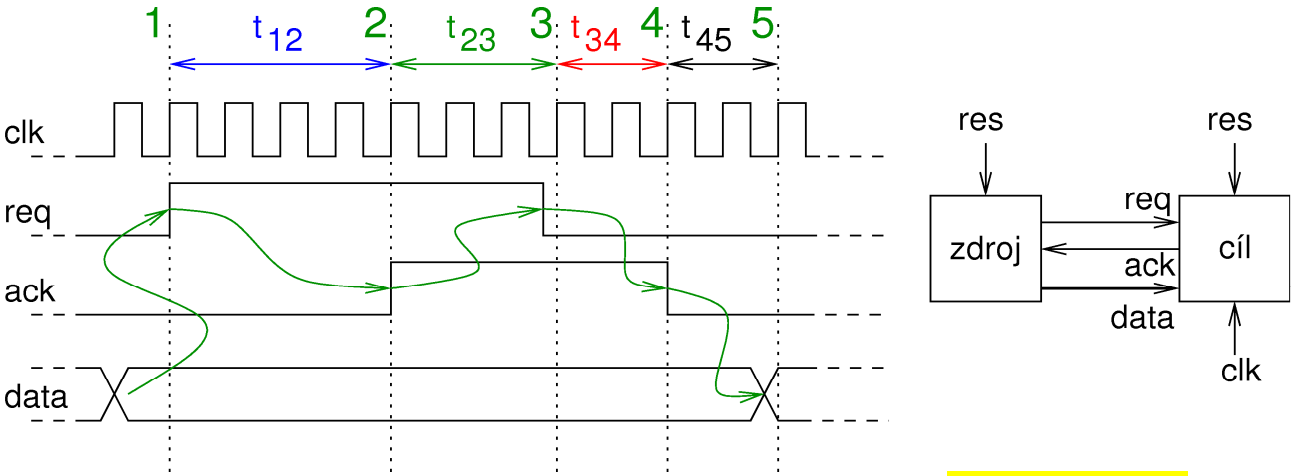
3 Posloupnosti událostí

Skutečná síla jazyka PSL vynikne až při potřebě sledovat výskyt posloupností událostí. Posloupnost (*sequence*) je v PSL textovým vyjádřením časového diagramu průběhů signálů [2, 4]. Syntaxe zápisu posloupností byla navržena tak, aby se daly snadno zapisovat a pochopit a zápis vizuálně respektuje i běžící čas (v textu zleva doprava). Jak uvidíme, zápis posloupnosti událostí je velmi kompaktní a intuitivně čitelný; tomu pomáhá i fakt, že některé jeho prvky byly převzaty z regulárních výrazů běžně známých uživatelům UNIXu a Linuxu. V terminologii PSL jsou posloupnosti často označovány jako *SEREs* (*Sequential Extended Regular Expressions*).

3.1 Příklad posloupnosti událostí

Vhodným příkladem posloupnosti událostí je chování signálů na rozhraní, které používá korespondenční režim (více o jeho významu a vlastnostech viz [7], kapitola 11), viz obrázek 5. Pořadí jednotlivých událostí a jejich návaznost je v obrázku vyznačena šipkami a očíslovanými časovými okamžiky:

1. Zdrojová strana vystaví data a aktivuje signál *req* (*request*) indikující požadavek na zpracování dat.
2. Cílová strana detekuje aktivní signál *req*, zpracuje data (to trvá vyznačenou dobu t_{12}) a aktivuje v odpověď signál *ack* (*acknowledge*) potvrzující příjem dat.
3. Zdrojová strana detekuje aktivní signál *ack* a deaktivuje signál *req*; indikuje tak, že zaznamenala úplné dokončení zpracování dat.
4. Cílová strana detekuje deaktivaci signálu *req* a deaktivuje *ack*; signalizuje tak, že je připravena přijmout nová data.
5. Zdrojová strana detekuje deaktivaci signálu *ack*, vystaví nová data a pokračuje bodem 1.



Obrázek 5: Typické posloupnosti událostí na rozhraní užívajícím korespondenční režim. Soubor handshake.tif.

Při práci s korespondenčním režimem je běžné, že je implementována sada assertions, které monitorují jeho správnou funkci, například mohou provádět kontrolu splnění následujících tvrzení:

- Je dodržena správná posloupnost hran na signálech *req* a *ack*; tedy při náběžné hraně *req* je *ack*='0', a po chvíli přijde vždy náběžná hrana *ack* a naopak, pokud se objeví náběžná hrana na *ack*, je *req*='1'. Tato tvrzení vycházejí z definice protokolu.
- Náběžná hrana na signálu *ack* přijde nejpozději po šesti hodinových cyklech hodin *clk*, $t_{12}<=6T_{clk}$. Toto tvrzení vychází ze znalosti implementace bloku a požadavků na jeho funkci.

Budou-li kontroly uvedených vlastností implementovány přímo v HDL jazyce, bude třeba použít jazykové konstrukce, které jsou používány pro implementaci stavových automatů. Implementace bude značně časově náročná a bude také složité ji odladit. Další udržování a úpravy kontrol v HDL jazyce bude nákladné a návrháři takovou dodatečnou práci nepřivítají. Zatímco HDL jazyky dobře zachycují strukturu systému na nízké úrovni, jsou nevhodné pro abstraktnější popis systému; proto vznikly specializované jazyky pro zápis assertions. Ty naopak snadno zachycují a popisují chování obvodu rozprostřené přes mnoho hodinových cyklů.

3.2 Definice posloupnosti

Nejjednodušší definice posloupnosti v jazyce PSL může vypadat například takto; sledujte současně obrázek 5, barvy jednotlivých částí posloupnosti odpovídají barvám časových kót v obrázku:

```
-- psl sequence seq_handshake is {req='1' AND ack='0';req='1' AND ack='0'; req='1' AND ack='0';req='1' AND ack='0'; req='1' AND ack='1';req='1' AND ack='1';req='1' AND ack='1'; req='0' AND ack='1';req='0' AND ack='1';req='0' AND ack='0'; req='0' AND ack='0'};
```

Posloupnost popisuje chování signálů *req* a *ack* přesně, jak jsou rozkresleny na obrázku 5. Všimněte si, že posloupnost je uzavřena ve složených závorkách a hodnoty signálů v jednotlivých hodinových cyklech jsou odděleny středníky. Výraz který popisuje každý z hodinových cyklů je pravdivý právě tehdy, pokud mají signály *req* i *ack* hodnoty podle obrázku.

Na první pohled vidíme, že zvolený způsob popisu není příliš praktický, jednotlivé stavy signálu se opakují a pokud by tento zápis byl jediný možný a současně například $t_{12}=100$, asi by čtenář neuvěřil, že jsou assertions k něčemu dobré. Proto PSL nabízí široké spektrum operátorů pro práci s posloupnostmi. Nejjednodušší a nejčastěji používaný operátor je operátor spojitého opakování prvku posloupnosti. Pomocí něj je možné ukázkovou posloupnost zapsat takto:

```
-- psl sequence seq_handshake is {(req='1' AND ack='0')[*4];(req='1' AND ack='1')[*3];(req='0' AND ack='1')[*2];req='0' AND ack='0'[*2]};
```

Jazyk PSL podporuje i další a rafinovanější zápisy, některé základní alternativy jsou shrnuty v následující tabulce.

Zápis	Vysvětlení
`(req='1' AND ack='0')[*3 to 5]`	Zápis posloupnosti, kde $t_{12}=3..5$. Limity 3 a 5 v zápisu musí být konstantami známými během elaborace návrhu v simulátoru.
`(req='1' AND ack='0')[*3 to inf]`	Pokud by platilo, že t_{12} >3, použijeme klíčové slovo inf - *infinity*, nekonečno.
`(req='1' AND ack='0')[+]`	Zkratka „+“ představuje jeden, nebo více výskytů, tedy $t_{12}>1$.
`(req='1' AND ack='0')[*]`	Uvedení jen znaku „*“ je zástupce pro 0 a více opakování, $t_{12}>0$.

PSL dále rozeznává mnohé další operátory a zápisy, jejich popis je už ale nad rámec tohoto krátkého seznámení s jazykem, více viz [2,4,8].

3.3 Operátor implikace

Samotná definice posloupnosti jen popisuje posloupnost událostí na signálech, její uvedení v HDL kódu nevede automaticky k žádnému kontrolování/monitorování událostí. Nejjednodušší způsob, jak zajistit kontrolování zadané posloupnosti, je použití operátoru implikace. Jeho funkce je obdobná příkazu *if* z vyšších programovacích jazyků:

if (předpoklad) then následek.

Simulátor zajistí monitorování spouštěcího *předpokladu* a pokud je tento splněn, bude hlídat, zda se kontrolované signály chovají podle *následku*. Pokud *následek* nebude splněn, simulátor ohlásí chybu příslušným hlášením. Jak *předpoklad*, tak *následek* mohou být komplexní posloupnosti, stejně dobře jako jednoduché logické výrazy; v textu i obrázcích budeme předpoklad označovat červeně, následek zelenou barvou. V PSL existují dva základní implikační operátory:

- překrývající se implikace (*overlapping implication*): |->, kdy se *následek* začne kontrolovat ve stejném cyklu, ve kterém je *předpoklad* splněn.
- nepřekrývající se implikace (*nonoverlapping implication*): |=>, kdy se *následek* začne kontrolovat až v příštím hodinovém cyklu, po cyklu ve kterém je *předpoklad* splněn.

Uvedeme si nyní příklady používání obou operátorů; nejprve příklad pro nepřekrývající se implikaci. Budeme například chtít kontrolovat, že $t_{12}=3..5$, tedy že cílový blok přijímající data včas zareaguje. První krok jedné možné implementace assertion definuje spouštěcí podmínku implikace, sledujte souběžně obrázek 6:

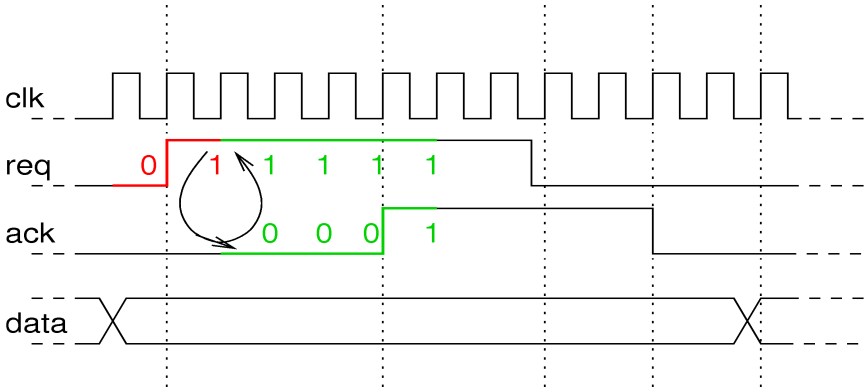
```
-- psl sequence seq_req_start is {req='0';req='1'};
```

Pak definujeme posloupnost, která bude kontrolována:

```
-- psl sequence seq_ack_response is {(req='1' AND ack='0')[*2 to 4];(req='1' AND ack='1')};
```

A nakonec definujeme vlastní assertion s implikačním operátorem:

```
  -- psl asrt_req_ack : assert always (seq_req_start |=> seq_ack_response) abort (res='1') @rising_edge(clk) report "Ack - req handshake failed.";
```



Obrázek 6: Nepřekrývající se implikace - grafické znázornění zpracování assertion. Kontrola následku *začíná až v hodinovém cyklu po naplnění předpokladu. Soubor handshake_nonoverlapping.tif .*

Oproti předchozím příkladům assertion přibyla nová část – operátor *abort* a podmínka. Operátor *abort* umožňuje přerušit kotrolu assertion za předem definovaných podmínek. Jeho použití je užitečné například tehdy, pokud detekovaná sekvence může být korektně přerušena resetem celého systému; v takovém případě by simulátor jinak ohlásil porušení podmínky i v případě, kdy je chování systému v pořádku.

Při porušení assertion simulátor opět vypíše příslušné hlášení do logu simulace:

*# ** Error: Ack - req handshake failed.*

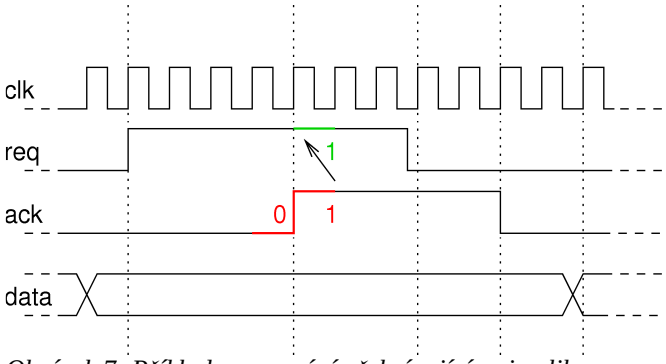
 # Time: 5600 ns ***Started: 4400 ns*** *Scope: /examples/asrt_req_ack File: examples.vhd Line: 128*

Všimněte si, že simulátor ohlásí nejen časový okamžik, kdy došlo k detekci chybového chování, ale i okamžik, kdy detekce posloupnosti začala, protože je rozprostřená přes mnoho hodinových cyklů:

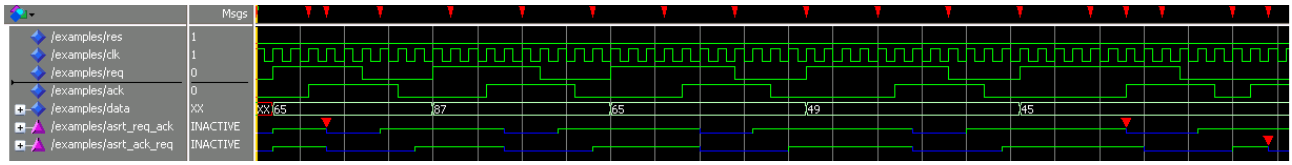
Druhý příklad, viz obrázek 7, bude demonstrovat použití překrývající implikace. Budeme chtít zkontrolovat opačnou podmínku; z definice protokolu vyplývá, že kdykoliv, kdy se vyskytne náběžná hrana na signálu *ack*, musí být signál *req*='1'.

```
-- psl sequence seq_ack_start is {ack='0';ack='1'};
-- psl asrt_ack_req : assert always (seq_ack_start |-> {req = '1'}) abort (res='1') @rising_edge(clk) report "Ack: req='0' failed.";
```

Na obrázku 8 je uveden příklad zpracování assertions v simulátoru QuestaSim [5]. Konečně, oba příklady assertions jsou opět uvedeny v ukázkovém kódu [6] volně dostupném zájemcům o problematiku assertions na internetových stránkách.



Obrázek 7: Příklad zpracování překrývající se implikace. Kontrola následku *začíná v hodinovém cyklu, ve kterém dochází k naplnění předpokladu. Soubor handshake_overlapping.tif .*



Obrázek 8: Ukázka funkce assertions v simulátoru QuestaSim [5]. Poslední dva řádky (fialové trojúhelníčky) představují průběh monitorování obou příkladů assertions, červené trojúhelníčky vrcholem dolů v časových průbězích indikují okamžiky, kdy assertion detekoval chybné chování. Obrázek handhsake_waves_detail.png.

4 Závěr

Jazyk PSL je – i přes absenci mnoha konstrukcí – košatým jazykem; v příspěvku byly čtenáři přiblíženy jen jeho elementární základy a i ty byly prezentovány vzhledem k omezenému rozsahu textu jen intuitivním způsobem. I přes to by čtenář po přečtení příspěvku měl být schopen jazyk PSL v omezeném rozsahu používat a podle příkladů sestavit i složitější assertions podle potřeby svého návrhu. K dalšímu seznamování s PSL lze čtenáři jen doporučit studium pramenů uvedených v použité literatuře, například [4]. Všechny prezentované assertions jsou dostupné ke stažení spolu s ukázkovým VHDL kódem z internetové stránky [6].

K získání úplného obrazu o používání assertions nicméně čtenáři stále mohou chybět případové studie – ukázky používání assertions v reálných návrzích. Ty jsou užitečné pro získání lepší představy o tom, kde všude mohou assertions pomoci a navíc mohou sloužit i jako inspirace pro vlastní práci. Případovým studiím proto bude věnováno další pokračování seriálu.

5 Použitá literatura

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

[2] Doulos. PSL tutorial. http://www.doulos.com/knowhow/psl/development_pslsugar/ [kontrolováno 31.12.2012]

[3] Accelera. Property Specification Language Reference Manual, Version 1.1. http://www.eda.org/vfv/docs/PSL-v1.1.pdf [kontrolováno 31.12.2012]

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

[5] Mentor Graphics. QuestaSim. http://www.mentor.com/products/fv/questa/ [kontrolováno 31.12.2012]

[6] minimizedlogic.sweb.cz [kontrolováno 9.12.2017]

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

[8] Ajeetha Kumari. Property Specification Language Tutorial. http://www.project-veripage.com/psl_tutorial_1.php [kontrolováno 4.10.2012]