Tento článek je upraveným původním rukopisem textu publikovaného v časopise DPS Elektronika A-Z: J. Šťastný. Verifikace pomocí assertions: případové studie, DPS Elektronika od A do Z, no 3, pp. 10-13, 2013.

Verifikace pomocí assertions: případové studie

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

1 Úvod

Předchozí články [1,2] po řadě představily jak základní koncepci, výhody a nevýhody assertions, tak základy jazyka PSL. K získání úplného obrazu čtenáři ale 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 je proto věnován tento příspěvek.

V následujícím textu jsou prezentovány různé modelové situace, se kterými se lze setkat v běžném životě. Předkládané příklady jsou vybrány z praxe autora článku, jsou to situace ve kterých se použití assertions osvědčilo a současně je možné jejich použití vysvětlit v omezeném prostoru tohoto příspěvku. U jednotlivých případů je kód assertion uveden přesně tak, jak je zapsán v příslušném návrhu číslicového systému; jsou také uvedeny další potřebné detaily k pochopení příkladu. Některé z demonstračních příkladů také úmyslně užívají vlastnosti PSL jazyka, které v článku [2] nebyly vysvětleny, aby byl čtenář upozorněn na další možnosti PSL a získal materiál pro další studium.

2 Hardwarový zásobník návratové adresy

V návrhu procesoru je použit blok hardwarového zásobníku pro návratovou adresu. Jedná se o jednoduchý procesor, zásobník je omezen na čtyři úrovně vnoření a návratová adresa se ukládá do registrů implementovaných přímo v zásobníku (pro implementaci zásobníku tedy není použita operační paměť). Rozhraní bloku a základní struktura je rozkreslena na obrázku 1. Blok má asynchronní resetovací vstup *mcu_res* aktivní v log. 1 (ten je vhodné použít pro blokování kontroly assertions), pracuje s náběžnou hranou hodin *mcu_clk* a současná hodnota čítače instrukcí je do něj přivedena po sběrnici *mcu_pc*. Zásobník – hlavní struktura v bloku – je implentován pomocí čtyř šestnáctibitových registrů *reg0 – reg3*. Operace prováděná nad zásobníkem je definována hodnotou na vstupní sběrnici *mcu_pc_cmd* realizované výčtovým typem. Na sběrnici se mohou objevit hodnoty *PC_CALL* (operace *CALL*, na zásobník je uložen aktuální stav čítače programu), *PC_RET* (operace *RETURN*, ze zásobníku je vyčtena poslední uložená návratová adresa) a *PC_RES* (operace *RESET*, obsah zásobníku je zinicializován).

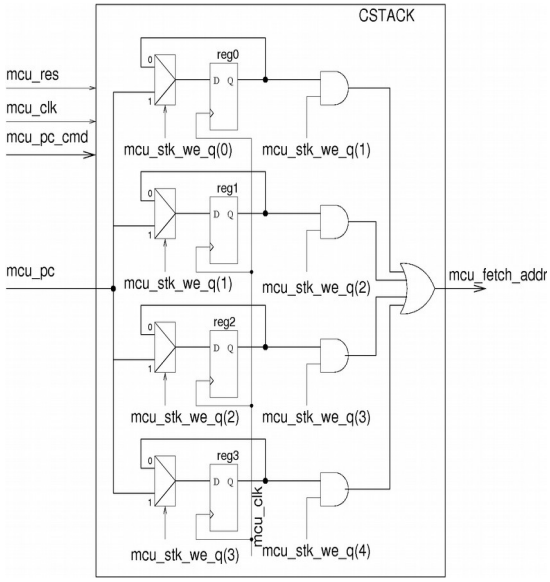
Všimněte si, že (sledujte také obrázek 1)

- zápis a čtení hodnot ze zásobníku vyvolané jako součást provádění operace *CALL* jsou řízeny pomocí sběrnice *mcu_stk_we_q*. Hodnota na této sběrnici je vždy kódována v kódu 1 z N, bit v log. 1 definuje do kterého registru se bude zapisovat a z kterého číst. Když je například *mcu_stk_we_q* = “00100”, je připraven k zápisu třetí registr zásobníku a je čtena hodnota z druhého registru.
- operace *CALL* vede na posuv hodnoty na sběrnici *mcu_stk_we_q* o bit vlevo (vložení hodnoty do zásobníku), operace *RETURN* o bit vpravo (vyčtení hodnoty ze zásobníku).

Kontrolovat zde během simulace můžeme pomocí assertions následující:

- Kódování sběrnice *mcu_stk_we_q* s hodnotou selektoru musí být v kódu 1 z N, jinak se na výstupu bloku objeví špatný výsledek, protože by se prostřednictvím OR hradla zkombinovalo více výstupů z registrů (kontrola vnitřní integrity bloku; assertion se v kódu v rámečku A jmenuje *asrt_stk_1ofn*).
- Nikdy nesmí být vykonána instrukce CALL (*mcu_pc_cmd = PC_CALL*), pokud je zásobník zcela zaplněn (kontrola správného použití bloku a vnitřní konzistence aplikace běžící na procesoru; *asrt_stk_full*). To, že je zásobník zcela zaplněn poznáme podle toho, že signál *mcu_stk_we_q(4)* je v log. 1.
- Niky nesmí být vykonána instrukce RET (*mcu_pc_cmd = PC_RET*), pokud je zásobník prázdný (kontrola správného použití bloku a vnitřní konzistence aplikace běžící na procesoru; *asrt_stk_empty*). To, že je zásobník zcela prázdný poznáme podle toho, že signál *mcu_stk_we_q(0)* je v log. 1.

Jak budou vypadat assertions v jazyce PSL je uvedeno v rámečku A.



Obrázek 1: Ideové schéma návrhu zásobníku návratové adresy. Soubor stack.png

A

```
-- psl default clock is rising_edge(mcu_clk);
-- psl property prop_stk_full is (mcu_stk_we_q(4) ->  NOT(mcu_pc_cmd = PC_CALL));
-- psl asrt_stk_full : assert always (prop_stk_full abort (mcu_res = '1'))
--                     report "CALL executed when the call stack is full!";
-- psl property prop_stk_empty is (mcu_stk_we_q(0) -> NOT(mcu_pc_cmd = PC_RET));
-- psl asrt_stk_empty : assert always (prop_stk_empty abort (mcu_res = '1'))report
--                      "RET executed when the call stack is empty!";
-- psl asrt_stk_1ofn : assert always((onehot(mcu_stk_we_q))  abort (mcu_res = '1'))
--                     report "Stack pointer is not one-hot coded!";
```

3 Čítač programu

V návrhu RISC procesoru pro embedded aplikace je dedikovaný blok realizující čítač programu. Celý blok je opět synchronní s náběžnou hranou hodin *mem_clk* a asynchronně resetovaný signálem *mcu_res* v log. 1. Hodnota čítače programu je přenášená po sběrnici *mcu_pc_q*, celý blok čítače programu je synchronní s náběžnou hranou hodin *mcu_clk*. Čítač programu je generický blok a vlastní šířka čítače i sběrnice, která nese adresu zpracovávané instrukce, je konfigurovatelná pomocí generického parameteru *pc_wdt*.

Z vlastností navrhovaného systému je zřejmé, že čítač programu nesmí nikdy přetéci, t.j. přejít z hodnoty FFFFh na hodnotu 0000h. Taková situace by znamenala, že bude opět spuštěna část firmware určená pro inicializaci systému po studeném startu uložená od adresy 0000h v době, kdy procesor není resetován. Jednalo by se tedy patrně o důsledek chyby aplikace. Assertion je v ukázce kódu pojmenován *asrt_pc_over*. Všimněte si také použití konstrukce *abort (mcu_res='1')*, která blokuje výpis chybového hlášení v případě, že dojde k resetu bloku právě když má čítač programu hodnotu FFFFh; pak je totiž přechod do stavu 0000h důsledkem správné funkce obvodu.

Jak bude vypadat assertion v jazyce PSL je uvedeno v rámečku B.

B

Ve VHDL kódu jsou definovány následující pomocné konstanty:

```
CONSTANT c_all_ones_i     : std_logic_vector (pc_wdt-1 DOWNTO 0) := (OTHERS => '1');
CONSTANT c_all_zeros_i    : std_logic_vector (pc_wdt-1 DOWNTO 0) := (OTHERS => '0');
```

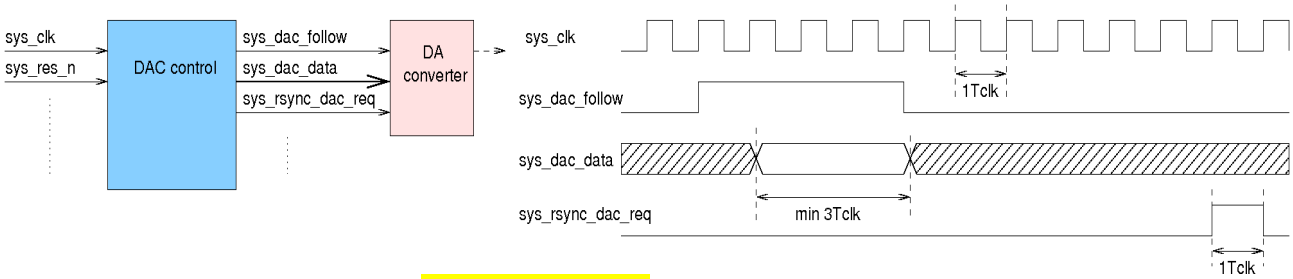
Ty jsou pak užity pro zjednodušení zápisu assertions:

```
-- psl default clock is rising_edge(mcu_clk);
-- psl sequence seq_pc_all_ones     is {mcu_pc_q  = c_all_ones_i};
```

```
-- psl sequence seq_pc_never_zeros  is {mcu_pc_q /= c_all_zeros_i};
-- psl property prop_pc_over        is ({seq_pc_all_ones} |=> {seq_pc_never_zeros});
-- psl asrt_pc_over : assert always ((prop_pc_over) abort (mcu_res = '1'))
--                    report "Program counter overflow!";
```

4 Řízení DA převodníku

Blok DA převodníku v analogově-číslicovém systému má specifické požadavky na řídící signály, k jeho řízení je v návrhu určený blok řadiče DA převodníku *DACcontrol*, viz obrázek 2. Celý blok řadiče je synchronní k náběžné hraně systémových hodin *sys_clk* a asynchronně resetovaný signálem *sys_res_n* aktivní v log. 0. Sestupná hrana signálu *sys_dac_follow* zapisuje do vstupního registru DA převodníku slovo ke konverzi na analogovou napěťovou úroveň. Implementace DA převodníku vyžaduje, aby vstupní slovo bylo stabilní alespoň tři hodinové cykly před sestupnou hranou signálu *sys_dac_follow* vzorkovanou náběžnou hranou systémových hodin. Konečně, blok generuje synchronizační pulz pro navazující systém, který zpracovává analogový výstup DA převodníku. Synchronizační pulz má být vždy široký jeden hodinový cyklus.



Obrázek 2: Blok řízení DA převodníku. Soubor dac_ctrl.png

Kontrolovat pomocí assertions můžeme

- Stabilitu datového signálu před sestupnou hranou *sys_dac_follow* (kontrola správné implementace řadiče DA převodníku, assertion *asrt_sys_dac_data*).
- Šířku synchronizačního pulzů na výstupu bloku – pulz smí být široký právě jeden hodinový cyklus (kontrola správné implementace řadiče DA převodníku, assertion *asrt_sys_rsync_dac*).

Příklady implementace assertions jsou v rámečku C.

C

```
-- psl property prop_sys_dac_data is always (fell(sys_dac_follow)) ->
-- (a_dac_data=prev(sys_dac_data, 1) AND a_dac_data=prev(sys_dac_data, 2) AND
--  a_dac_data=prev(sys_dac_data, 3)) abort (sys_res_n='0') ;
-- psl asrt_sys_dac_data : assert prop_sys_dac_data @rising_edge(sys_clk)
-- report "DAC ctrl: sys_dac_data bus is not stable three clock cycles before the
dig_dac_follow falling edge.";
-- psl property prop_sys_rsync_dac is always (rose(sys_rsync_dac_req) ->
--        {sys_rsync_dac_req; not(sys_rsync_dac_req)}) abort (sys_res_n='0') ;
-- psl asrt_sys_rsync_dac : assert prop_sys_rsync_dac @rising_edge(sys_clk)
--        report "Output sys_rsync_dac_req is not one clock cycle wide";
```

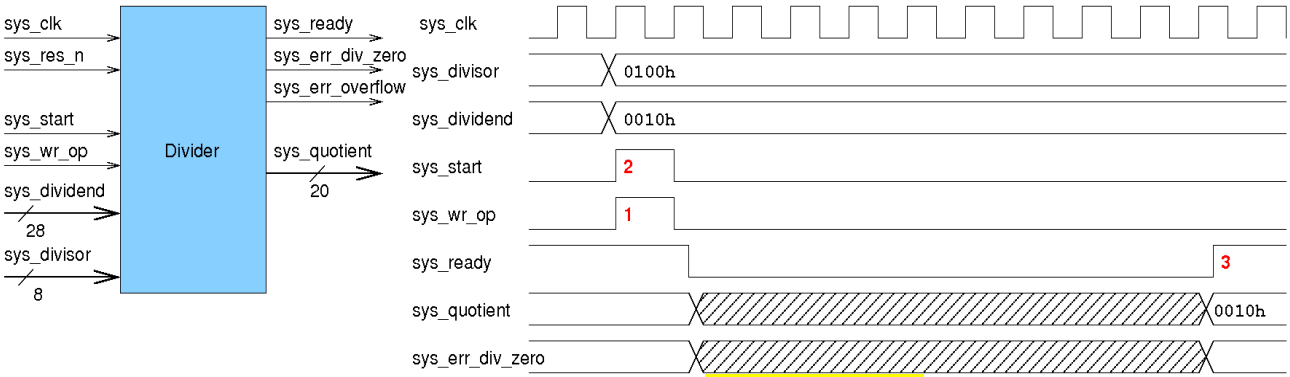
5 Řízení děličky

Mikropočítačový systém používá pro urychlení operace dělení blok hardwarové děličky, rozhraní bloku a základní komunikační protokol na rozhraní děličky viz obrázek 3. Blok děličky je synchronní s náběžnou hranou na hodinách *sys_clk* a je také asynchronně resetován signálem *sys_res* v log. 1. Dělenec a dělitel je do registrů dělence a dělitele děličky nahrán tehdy, když je vstupní signál *sys_wr_op* v log. 1 (časový okamžik 1). Během vlastního dělení jsou tyto registry použity pro realizaci funkce dělení a jejich obsah se v průběhu dělení mění, nelze je tedy přepsat, pokud dělička běží, aby nedošlo k porušení její správné funkce. Rozhraní děličky pracuje v korespondenčním režimu; dělení je spouštěno signálem *sys_start* (okamžik 2) a dokončení operace dělení indikuje náběžná hrana signálu *sys_ready* (okamžik 3). Pro snažší implementaci bloku se počítá s tím, že během výpočtu dělení se může jak výstupní sběrnice *sys_quotient,* tak příznaky přetečení a dělení nulou libovolně měnit (šrafovaná oblast v obrázku 2); nicméně po

ukončení výpočtu (po náběžné hraně signálu *sys_ready*) už řadič mikropočítače předpokládá, že se výstupy bloku nemění a jsou stabilní. Dělení je spuštěno mikroprogramem nadřazeného řadiče mikropočítače.

Do bloku děličky tak můžeme přidat například tyto assertions:

- Indikace dělení nulou – signál *sys_err_div_zero* – nesmí měnit svoji hodnotu, pokud neprobíhá dělení (kontrola správné implementace děličky – pokud neprobíhá dělení, nesmí se nic dít; assertion *asrt_stable_sys_err_div_zero*).
- Řadič mikropočítače nesmí číst výstupní registr děličky, dokud není dokončeno dělení (kontrola správné implementace mikroprogramu nadřazeného mikropočítače – tedy integrace děličky, assertion *asrt_early_div_lo*). K děličce mikroprogram přistupuje instrukcemi pro čtení a zápis do/z registrů; například čtení výstupu dělení lze identifikovat tak, že je na adresní sběrnici *mcu_rd_addr* přítomna adresa uložená v konstantě *divquotient*.
- Řadič mikropočítače se nesmí pokusit odstartovat dělení, pokud dělení právě probíhá (kontrola správné implementace mikroprogramu nadřazeného mikropočítače – tedy integrace děličky – a současně kontrola toho, že dělení „jednou skončí“, protože nekonečná operace dělení vyvolá při příštím spuštění bloku chybu; assertion *asrt_div_start*).
- Řadič mikropočítače nesmí přepsat registry dělence a dělitele pokud probíhá dělení (opět kontrola správné implementace mikroprogramu – integrace děličky; assertion *asrt_div_wr*).
- V rámci verifikace celého systému je třeba provést i ověření správné funkce děličky v mezních stavech. Verifikátor bloku tedy bude chtít vědět, zda během simulací došlo k tomu, že bylo vyvoláno dělení nulou (coverage assertion je pojmenován *cov_err_zero*) a že došlo k přetečení (*cov_err_over*). Konečně, je vhodné vědět i kolikrát bylo dělení v rámci verifikace obvodu vůbec spuštěno (assertion *cov_start*). Všimněte si zde použití vlastnosti *seq_detect_nonzero* s parametrem, obdoby volání funkce v běžných programovacích jazycích.



Obrázek 3: Blok děličky a posloupnosti signálů na jejím rozhraní. Soubor divider.png.

Jak budou assertions vypadat v jazyce PSL je uvedeno v rámečku D.

D

```
-- psl default clock is rising_edge(sys_clk);
-- psl property prop_stable_sys_err_div_zero is always ((sys_ready and
-- prev(sys_ready)) -> (stable(sys_err_div_zero))) abort (sys_res='1') ;
-- psl asrt_stable_sys_err_div_zero : assert prop_stable_sys_err_div_zero
-- report "Output sys_err_div_zero is not stable if sys_ready='1'.";
-- psl property prop_early_div_lo is ( (unsigned(mcu_rd_addr) = divquotient) ->
(sys_div_ready = '1') );
-- psl asrt_early_div_lo : assert always(prop_early_div_lo) abort (mcu_res = '1')
-- report "Divider output is read too early.";
-- psl property prop_div_start is ( rose(sys_start) -> (sys_ready='1'));
-- psl asrt_div_start : assert always (prop_div_start) abort (sys_res='1')
--                      report "Divider is being started while it is busy.";
-- psl property prop_div_wr is ( rose(sys_wr_op) -> (sys_ready='1'));
-- psl asrt_div_wr : assert always (prop_div_wr) abort (sys_res='1')
-- report "Divider operands are being written while the block is busy.";
-- psl sequence seq_detect_nonzero(boolean sig) is {sig='1'};
-- coverage assertions
-- psl cov_err_zero : cover(seq_detect_nonzero(sys_err_div_zero));
-- psl cov_err_over : cover(seq_detect_nonzero(sys_err_overflow));
-- psl cov_start    : cover(seq_detect_nonzero(sys_start));
```

6 Závěr

V textu článku byly prezentovány některé příklady použití assertions v návrhářské praxi. Ty mohou sloužit jednak jako inspirace, jednak po úpravě mohou být použity pro řešení podobných situací ve vlastních návrzích. Seriál o praktické aplikaci a použití assertions bude zakončen příštím příspěvkem, který bude věnován detailům metodologie použití assertions a několika radám, jak s verifikací pomocí assertions začít i bez potřeby simulátoru s podporou jazyka PSL.

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