Tento článek je původním rukopisem textu publikovaného v časopise DPS Elektronika A-Z:
J. Šťastný. Simulace číslicových obvodů: triky i úskalí simulace, DPS Elektronika od A do Z, pp. 20-23, březen/duben 2015

Simulace číslicových obvodů: triky i úskalí simulace

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

1 Úvod

Předchozí příspěvek [1] shrnul základní principy práce číslicového simulátoru, popsal úrovně abstrakce, na kterých se návrhář pohybuje a jejich základní implikace pro návrhářskou práci. Nyní se zaměříme na elementární úskalí RTL simulace a popíšeme několik jednoduchých triků, které mohou práci návrháře ulehčit. Článek proto spíše než ucelený text shrnuje „vybrané kapitoly z práce návrháře“.

2 Delta cykly a simulace

S delta cykly byl čtenář již seznámen v [1]. Koncepce delta cyklů v událostmi řízené simulaci je velice užitečná abstrakce, nicméně její použití s sebou přináší i drobné komplikace. Nevhodně implementovaný model číslicového systému s kombinační zpětnou vazbou může vést na nekonečnou smyčku v simulátoru. Nejjednodušší nekonečná smyčka vznikne například implementací následujícího procesu:

```
delta_loop : PROCESS (data)
BEGIN
  data <= NOT(data);
END PROCESS delta_loop;
```

Zde se jedná o uměle vykonstruovaný případ; kód, se kterým se setkáte ve vašem projektu, a způsobí problémy, bude jistě zákeřnější. Nicméně například simulátor ISim problém rychle detekuje:

ERROR: at 0 ps: Iteration limit 10000 is reached. Possible zero delay oscillation detected...

Pokud by bylo třeba odladit složitější problém, ISim umožňuje detekovat, kde problém vzniká. Čtenář může (po spuštění ukázkového příkladu v simulátoru) například zkusit příkazy:

```
isim ptrace on
run
```

K objasnění jejich funkce lze použít dokument [2]; limit na počet delta cyklů lze nastavit kliknutím pravým tlačítkem na *Simulate Behavioral Model* v okně *Processes*, volbou *Properties* a vyplněním:

```
-maxdeltaid <iterační limit>
```

do políčka *Other Simulator Commands*.

Simulátor ModelSim bude reagovat stejným způsobem:

*# ** Error: (vsim-3601) Iteration limit reached at time 0 ns.*

V pravém dolním rohu okna simulátoru je hlášení o simulačním čase: *Now: 0 ns Delta: 4999.* Simulace tedy byla zastavena po 5000 delta cyklech v čase 0 ns. Limit na počet delta cyklů, po kterých je simulace ukončena, je možné změnit v souboru *modelsim.ini*:

; Maximum iterations that can be run without advancing simulation time

IterationLimit = 5000

Ačkoliv delta cyklus reprezentuje nekonečně malý časový okamžik, má z funkčního hlediska dopad jako každé jiné reálné zpoždění. Někdy delta cyklový posuv dat a hodin v obvodu může způsobit dokonce nesprávnou funkci modelu obvodu v simulátoru, více viz např. [4], nebo příklad v [3], kapitola 7, strana 419. Odladit takový problém by bez podpory simulátoru nebylo snadné, simulátory proto často nabízí možnost delta cykly vizualizovat v okně *Wave*. Například v simulátoru ModelSim se k této možnosti dostanete kliknutím pravým tlačítkem myši do okna *Wave*, podmenu *Expanded Time*, více viz [5].

3 Časové rozlišení simulátoru

VHDL pro podporu práce s časem definuje fyzický typ *time* ve VHDL Language Reference Manual [6], sekce 3.1.3.1. Čas je zde definován jako celočíselný datový typ s rozsahem alespoň -2^{31} do $+2^{31}$, přitom konkrétní implementace v simulátoru může podporovat větší rozsah hodnot (podle [7] se v simulátorech používá spíše 64 bitová reprezentace). Protože je typ *time* definován jako celé číslo, ale čas je ze své podstaty neceločíselná veličina, je přepočet na skutečný

čas realizován s pomocí časové jednotky. Tu lze definovat různě; pro simulátor ModelSim a jazyk VHDL je nejsnáze dosažitelná v souboru *modelsim.ini*:

```
; Simulator resolution
; Set to fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or 100.
Resolution = ns
```

Časovou jednotku lze také nastavit v grafickém rozhraní ModelSim, pod položkou menu *Simulate → Start simulation*, v menu *Resolution*. V simulátoru ISim lze pro VHDL návrh použít stejný postup jako pro nastavení počtu iterací – kliknutím pravým tlačítkem na *Simulate Behavioral Model* v okně *Processes*, volbou *Properties* a vyplněním například:

```
-timeprecision_vhdl 1ns
```

do políčka *Other Compiler Options*. Další užitečné přepínače jsou *-override_timeprecision*, *-override_timeunit* a *-timescale* <*time_unit/time_precision*>, více viz například [8].

Konečně, pracujete-li v jazyce Verilog, je možné použít direktivu `'timescale` vloženou do zdrojového kódu:

```
'timescale 1 ns/1 ps
```

Potom bude 1 ns použita jako časová jednotka ve výpisech ze simulátoru a 1 ps jako časová jednotka použitá pro přepočet z celého čísla na čas. Bez ohledu na způsob nastavení časové jednotky je tato nejmenším časovým kvantem, které simulátor rozliší. Když tedy nastavíte jednotku 1 ns, nebude možné vyjádřit kratší časový interval.

Skutečnost, že je čas vyjádřený prostřednictvím celočíselného typu občas komplikuje život. Například při výpočtu podílu dvou časů zápis `time1/time2` vede na celočíselné dělení s (možná) překvapivými výsledky; podíl dvou časů lze lépe spočítat například takto (pro rozlišení času 1 ps):

```
  VARIABLE r_time1 : real;
  VARIABLE r_time2 : real;
BEGIN
  r_time1 := 1.0 * (time1 / 1 ps);
  r_time2 := 1.0 * (time2 / 1 ps);
  time_div <= r_time1 / r_time2;
```

4 Černá skříňka – bílá skříňka

Při verifikaci složitějších návrhů je z mnoha důvodů přístup technikou „černé skříňky“ neefektivní a nevýhodný. Často totiž není postačující monitorovat signály na rozhraní číslicového systému, ale je třeba sledovat i stavy konkrétních signálů uvnitř návrhu, aby bylo možné posoudit správnou funkci celého systému. Dříve podporu pro přímý přístup k signálům napříč hierarchií návrhu ve VHDL nabízely simulátory jen ve formě specifických utilit. V simulátoru ModelSim je tato konstrukce nazývána *Signal Spy* a její použití je snadné; do entity, ve které se chceme „dívat“ na vnitřní signály v návrhu, je třeba vložit tento kód:

```
LIBRARY modelsim_lib;
USE modelsim_lib.util.ALL;
init_spies:PROCESS
BEGIN
  init_signal_spy ("/i_top_level/i_pwm_core/overflow", "/overflow_i");
  WAIT;
END PROCESS init_spies;
```

Popsaná konstrukce propojí zdrojový signál `i_top_level/i_pwm_core/overflow` s cílovým *overflow_i*, jehož stav pak lze sledovat. Čtenáři zde doporučíme nastudovat další detaily z dokumentu [3]. O problematice černé/bílé či šedé skříňky se lze více dozvědět například v knize [4].

Implementace této funkce je bohužel specifická pro konkrétní simulátor. Kód, který užije *Signal Spy,* bude třeba pro jiný simulátor přepsat (také je taková konstrukce nepoužitelná v syntetizovatelném kódu, ovšem zdrojový signál, na který se díváme, v syntetizovatelném kódu samozřejmě být může). Naštěstí dnes všechny moderní jazyky pro návrh obsahují konstrukce, které tento problém snadno překlenou; speciálně ve VHDL je prostředek pro přístup k objektům návrhu napříč hierarchií definován normou VHDL 2008, více viz např. přehledová stránka [9], odstavec *Hierarchical names*. Pro úplnost poznamenejme, že v jazycích Verilog i SystemVerilog je hierarchický přístup také podporovaný přímo jazykovými konstrukcemi.

5 Práce s *wave file*

Každý simulátor dokáže ukládat do *wave file* (databáze s časovými průběhy logických hodnot na signálech v návrhu) na pevný disk události probíhající v návrhu; ty je pak možné zpětně zkoumat. To je velice praktické, na druhou stranu

nevhodné použití *wave file* může významně zpomalit simulaci a rychle zaplnit pevný disk. U malých návrhů problémy nenastávají, ale pro větší návrhy a delší simulace *wave file* může růst velice rychle (až do řádu gigabajtů a více) a simulaci může citelně zpomalovat ukládání událostí do souboru. Na druhou stranu se ale ukládání událostí do *wave file* nelze jednoduše zbavit, protože pak nelze zkoumat chování obvodu v okně prohlížeče časových průběhů (okno *Wave*). Problémy mohou snadno nastat při potřebě zkoumat chování velkého návrhu až po delším simulačním čase; pak může pomoci jednoduchý postup:

1. Nechat simulaci doběhnout „kousek“ před místo, které chceme zkoumat, příkazem `run <čas>`.
2. Zapnout ukládání do *wave file* příkazem `log`.
3. Nechat simulaci příkazem `run` doběhnout kam potřebujeme.

Alternativou k tomuto postupu může také být zapnout zápis událostí do *wave file* už na začátku simulace, ale zapisovat události jen z části návrhu (bloku, jehož funkci je třeba odladit). I tímto způsobem může dojít k významnému zmenšení velikosti *wave file* a zrychlení běhu simulace. Čtenáři zde doporučujeme detailněji prostudovat dokumentaci k příkazu, který v jeho simulátoru řídí ukládání časových průběhů do souboru.

Při práci se simulátorem se hodí vědět ještě o následujících maličkostech:

1. Číslicové simulátory typicky umějí jednou uložený *wave file* později opět načíst a zobrazit v okně s časovými průběhy signálů. To může být užitečné například pro zpětné zkoumání běhu simulace, která byla spuštěna v dávkovém režimu. Hodí se to i tehdy, pokud je k dispozici méně licencí na grafické rozhraní, než na simulační jádro simulátoru. Návrháři nemusí blokovat licenci na grafickou nadstavbu simulátoru po dobu běhu simulace, ale využívají ji opravdu jen po dobu, po kterou zkoumají chování obvodu.
2. V některých simulátorech lze také navzájem porovnávat dříve uložené *wave files*. To může být užitečné při porovnávání chování dvou verzí jednoho návrhu, nebo například hradlové a RTL implementace návrhu. Simulátor po načtení obou srovnávaných *wave files* přehledně zobrazí, kde se chování návrhů liší a umožní tak mnohem rychleji objevit problém. Vzhledem k omezenému rozsahu článku zde nicméně čtenáře odkazujeme na podrobnější informace dostupné v dokumentaci k jeho simulačnímu nástroji.

6 RTL modelování a syntéza obvodu

6.1 Inicializace signálů

Je standardem ve VHDL inicializovat neinicializované objekty typu *T* na hodnotu *T'LEFT*. Například při použití RTL VHDL popisu a knihovny *std_logic_1164* mají všechny neinicializované signály typu *std_logic* na počátku simulace hodnotu *U*. Tak lze snadno odhalit signály, které mají nedefinované hodnoty, neinicializované registry, či celé neresetované bloky v systému. Pokud ale autor RTL kódu použije implicitní přiřazení v deklaraci signálu, např.:

```
SIGNAL a: std_logic := '0';
```

této výhody se zříká a navíc může ještě v systému zamaskovat nepříjemnou chybu. Proto je tento způsob inicializace hodnoty signálu v RTL kódu vhodné nepoužívat. Dobrou praxí je také občas spustit simulaci a prohlédnout v okně *Wave* všechny signály; cílem je najít ty, které jsou dlouhodobě v *U* či *X*. Simulátor tento úkol sám usnadňuje tím, že oba stavy bývají znázorněny jinou barvou, než běžné logické stavy 0 a 1. Je to tedy jednoduchý způsob kontroly správné implementace resetů v návrhu, nezapojených (nebuzených) signálů (*U*) a případných zkratů (*X*).

6.2 Neúplné citlivostní seznamy

Častou chybou v RTL návrhu je nesestavení úplného citlivostního seznamu u procesu. Citlivostní seznam definuje podmínky za kterých je proces přepočítán – obsahuje seznam signálů, na kterých změna vyvolá přepočítání procesu. Je-li neúplný, není proces přepočítán vždy, kdy by měl, a jeho výstup nemusí odpovídat modelované fyzikální realitě. Co všechno v seznamu může chybět, ukazuje příklad:

```
log: PROCESS (a,b)
BEGIN
  d <= a AND b;
  y <= d OR c;
END PROCESS log;
```

Zkušenější čtenář jistě snadno odhalí, že v hlavičce procesu má být správně:

```
log: PROCESS (a,b,c,d)
```

Signál *c* musí být v citlivostním seznamu proto, že je primárním vstupem modelované kombinační funkce. A signál *d* je sice vnitřním výstupem, ale ve skutečnosti se chová také jako primární vstup. Vše je znázorněno i s vysvětlením na obrázku 1; implementace v simulátoru se chová přibližně jako v následujícím výpisu, kde *n* je číslo současného delta cyklu:

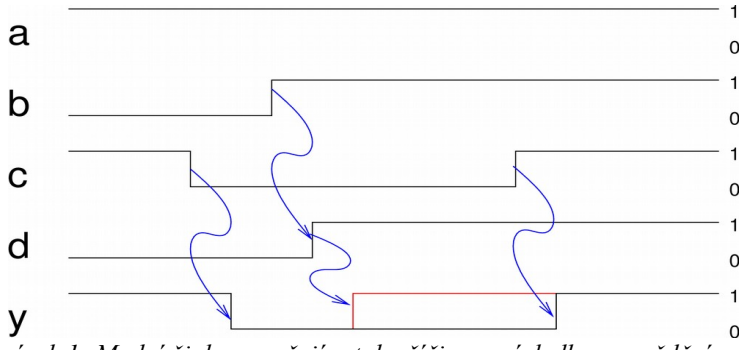
```
log: PROCESS (an,bn)
BEGIN
  dn+1 <= an AND bn;
```

```
  y_{n+1} <= d_n OR c_n;
END PROCESS log;
```

Poznamenejme, že celý proces můžeme řešit také takto:

```
log: PROCESS (a,b,c)
  VARIABLE d:std_logic;
BEGIN
  d := a AND b;
  y <= d OR c;
END PROCESS log;
```

Zde není třeba (a ani možné) mít proměnnou *d* v citlivostním seznamu. Syntéza se s tímto řešením bez problémů vypořádá, *d* ovšem už není signál a jeho hodnota tedy není dostupná mimo proces.



Obrázek 1: Modré šipky označují vztah příčiny a následku a zpoždění o jeden delta cyklus. Červenou je u výstupu y *vyznačeno správné chování; v tomto místě se RTL model s realitou rozchází. Soubor sensitivity.eps*

Druhým extrémem je přespecifikovaný citlivostní seznam obsahující signály navíc. Ten nezpůsobuje funkčně chybnou simulaci, ale simulace bude zpomalena zbytečným spouštěním příslušného procesu. Citlivostní seznam by tedy měl obsahovat jen přesně to, na co má proces reagovat. ModelSim umožňuje automaticky kontrolovat citlivostní seznamy pomocí přepínače kompilátoru *check_synthesis* (`vcom -check_synthesis`) ; kompilátor během kompilace výše uvedeného procesu vypíše do okna *Transcript* následující varování:

```
# ** Warning:...: (vcom-1400) Synthesis Warning: Signal "d" is read in the process but is not in the sensitivity list.
# ** Warning:...: (vcom-1400) Synthesis Warning: Signal "c" is read in the process but is not in the sensitivity list.
```

Podobně na problémy s citlivostními seznamy upozorňují syntézní nástroje, například Xst od firmy Xilinx při syntéze kódu s neúplnými citlivostními seznamy vypíše podobné varování označené jako `HDLCompiler:92`. Je na návrháři, aby systematicky a pravidelně kontroloval logy z nástrojů a opravoval podobné problémy. Je také zřejmé, že ignorovat varování generovaná nástroji je hrubá chyba.

Závěrem poznamenejme, že problém neúplného citlivostního seznamu elegantně řeší VHDL 2008 pomocí klíčového slova *all*, více viz např. přehled v [9].

6.3 Synthesis on/off

Za určitých podmínek může být výhodné mít v jednom souboru současně jak syntetizovatelný, tak nesyntetizovatelný kód. Pomocí běžných jazykových konstrukcí ale nelze syntezátoru snadno říci, co má syntetizovat a co ne. Proto byly zavedeny direktivy (*pragmas*) vkládané do kódů. Standardní podoba direktivy je:

```
-- pragma direktiva
```

`direktiva` v našem případě bude `synthesis_on` či `synthesis_off`. Protože je zapsaná jako komentář, nástroje, které ji neznají ji ignorují. Všechny autorovy známé syntézní nástroje nicméně podporují direktivu `synthesis_on/off`. Po nalezení `synthesis_off` syntezátor přestane následující RTL kód syntetizovat, direktiva `synthesis_on` syntézu opět zapíná. Direktiva vypínající syntézu může být užitečná v řadě případů, například při implementaci různých pomocných verifikačních konstrukcí napsaných ve VHDL a začleněných přímo do RTL kódu. Typické použití pak vypadá takto:

```
-- pragma synthesis_off
... VHDL kód, který se nemá syntetizovat ...
-- pragma synthesis_on
```

Direktivy je nicméně možné používat i „kreativnějším“ způsobem:

```
  sync <= (shift_reg(1)
```

```
  --pragma synthesis_off
  AND '0') OR (shift_sim(2)
  --pragma synthesis_on
  );
```

Simulátor potom ve standardním nastavení „uvidí“

```
  sync <= (shift_reg(1) AND '0') OR (shift_sim(2));
```

tedy vlastně

```
  sync <= shift_sim(2);
```

zatímco syntézní nástroj vysyntetizuje

```
  sync <= shift_reg(1);
```

Autor ve své praxi narazil na řadu případů, kdy taková aplikace direktivy byla užitečná. Na druhou stranu ale není zcela standardní, mohly by s ní být v některých nástrojích problémy a je namístě podobné konstrukce používat se zdravou dávkou nedůvěry a opatrnosti.

Při používání direktiv řídících syntézu je třeba nezapomenout na „uzavírající“ direktivu, která opět syntézu povolí (tj. nevynechat druhé `--pragma synthesis on`). Jinak se část návrhu nevysyntetizuje a může chvilku trvat, než se najde a odstraní příčina (zdánlivě nelogicky jsou pak ignorovány části návrhu).

Direktiv je nástroji pro návrh typicky podporováno víc (například `translate_on/off`, která se od `synthesis_on/off` liší jen v detailech). Zde čtenáři opět doporučujeme prostudovat dokumentaci k jeho oblíbenému syntéznímu nástroji. Některé direktivy jsou také podporovány simulátory.

7 Závěr

Článek vysvětlil několik užitečných triků a shrnul nejtypičtější úskalí, se kterými se návrhář během RTL návrhu může setkat. Nabízí se nyní otázka, zda se jim dá bezpečně vyhnout, nebo je alespoň rychle odhalit? Zde lze čtenáři jen doporučit pravidelnou kontrolu vytvářeného číslicového návrhu odpovídajícími nástroji. Částečně si lze pomoci kompilací s přepínačem *check_synthesis*, pravidelným spouštěním nástrojů (například v týdenním cyklu každý pátek) pro syntézu/rozmístění a propojení a kontrolou jejich logů; na trhu jsou také k dispozici nástroje pro statickou kontrolu kódu (tzv. *linting*). Vzhledem k omezenému rozsahu příspěvku nebylo možné probrat všechny potřebné detaily a problematiku bylo třeba dosti zjednodušit, čtenáře zde vybízíme k dalšímu studiu (začít lze například prameny uvedenými v použité literatuře).

8 Použitá literatura

[1] Jakub Šťastný, Simulace číslicových obvodů: úvod. DPS Elektronika od A do Z, leden/únor 2015, str. 23-27.

[2] Xilinx, ISim User Guide. [online: http://www.xilinx.com/cgi-bin/docs/rdoc?l=en;v=14.7;d=plugin_ism.pdf] (kontrolováno 5.11.2014)

[3] MentorGraphics, ModelSim User's Manual (dostupné v instalaci ModelSimu: *docs\pdfdocs\modelsim_se_user.pdf*)

[4] Janick Bergeron, Writing Testbenches: Functional Verification of HDL Models. Springer, 2nd edition, 2003.

[5] Mentor Graphics, ModelSim Reference Manual. Dostupné po instalaci nástroje v adresáři: modelsim_installation_path\docs\pdfdocs\xxx_sim_ref.pdf, kde *xxx* je jméno simulátoru.

[6] IEEE, IEEE Standard 1076-2008. VHDL Language Reference Manual, Section 12, Elaboration and Execution. IEEE 2009. [online: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4772738] (kontrolováno 29.10.2014)

[7] ALDEC, Time Calculations in VHDL. [online:https://www.aldec.com/en/support/resources/documentation/articles/1165] (kontrolováno 29.10.2014)

[8] Xilinx, FUSE Options. [online: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ism_cl_fuse_options.htm] (kontrolováno 25.11.2014)

[9] Doulos, VHDL 2008 Ease of Use. [Online: http://www.doulos.com/knowhow/vhdl_designers_guide/vhdl_2008/vhdl_200x_ease/] (kontrolováno 25.11.2014)

[10] Don Mills a Clifford Cummings, RTL coding styles that yields simulation and synthesis mismatches. SNUG 99. [online: *www.vhdl.org/vlog-synth/Mills_Final.pdf*] (kontrolováno 29.10.2014)