Tento článek je původním rukopisem textu publikovaného v časopise DPS Elektronika A-Z:
J. Šťastný. Simulace číslicových obvodů: úvod, DPS Elektronika od A do Z, pp. 23-27, leden/únor 2015.
Bez souhlasu autora tohoto materiálu a redakce časopisu DPS a uvedení zdroje není povolena jakákoli další publikace, přetištění nebo distribuce tohoto materiálu nebo jeho části. Další podmínky použití jsou uvedeny na internetové stránce http://minimizedlogic.sweb.cz/.

Simulace číslicových obvodů: úvod

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

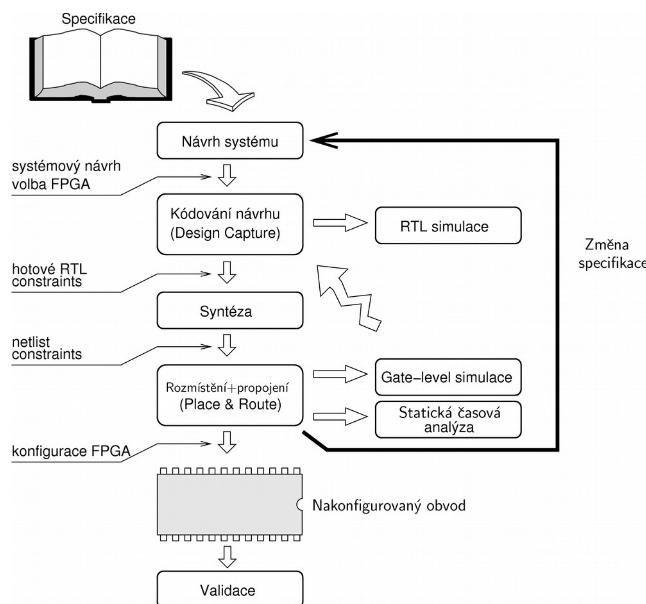
1 Úvod

Proces návrhu číslicového systému zahrnuje řadu aktivit od specifikace až po validaci v aplikaci, viz obrázek 1 (a kapitola 2 v [1]). Už při letmém pohledu na obrázek si nelze nevšimnout, jak často se v něm vyskytuje slovo *simulace*. Při simulaci aplikujeme na vstupy navrhovaného obvodu stimuly realizující zvolené scénáře a ověřujeme, že reakce návrhu jsou správné. Tomuto procesu říkáme verifikace. Právě verifikací (a tedy simulacemi) tráví návrhář většinu svého času; moderní číslicové simulátory ovšem obsahují řadu funkcí, které mají návrháři jeho práci zjednodušit a ulehčit. Pro efektivní práci se simulátorem se také hodí vědět o potenciálních úskalích, které práce s číslicovou abstrakcí reálného světa přináší. Popis nejlepších postupů (*best practices)* a varování před nejtypičtějšími úskalími číslicové simulace jsou proto tématem tohoto seriálu.

Jelikož v centru pozornosti bude stát číslicový simulátor, předpokládá se, že čtenář má elementární zkušenosti s psaním syntetizovatelného kódu a s jazykem VHDL nejlépe v rozsahu knih [1,2]. Simulátor bude použit jednak ModelSim od firmy Mentor Graphics a jednak ISim od firmy Xilinx. Drtivá většina poznatků je nicméně aplikovatelných i na jiné jazyky a simulační nástroje jak při návrhu zákaznických číslicových obvodů, tak při návrhu číslicových obvodů s použitím FPGA. Informace obsažené v seriálu pocházejí jak ze zdrojů citovaných na konci každého článku, tak i z návrhářských a pedagogických zkušeností autora.

2 Úrovně abstrakce

Prací návrháře je převést abstraktní specifikaci návrhu na skutečný fyzický návrh prostřednictvím řady po sobě jdoucích kroků, viz obrázek 1.



Obrázek 1: Postup návrhu číslicového systému (design flow) s použitím FPGA obvodu. Obrázek design_flow.eps

Prvním krokem tohoto procesu je tvorba specifikace následovaná systémovým návrhem. Ten dále v kroku kódování návrhu (*design capture*) modelujeme na RTL úrovni (*Register Transfer Level*, viz [1]). V souvislosti s RTL návrhem hovoříme o *modelování* navrhovaného systému; RTL je model, který může (ale nemusí) odpovídat systémovému návrhu a specifikaci a návrhář se to může (ale nemusí) včas dozvědět. Správnost návrhu na RTL úrovni je třeba ověřit mimo jiné jeho simulací.

Při modelování převádíme popis (model) návrhu mezi úrovněmi abstrakce [3]. Čím je model na vyšší úrovni abstrakce, tím potřebujeme v modelu větší rezervy a model je abstraktnější s méně detaily. Každý přechod na nižší úroveň

abstrakce detaily naopak přidává a zvyšuje složitost celého modelu. Proto se modely na vyšší úrovni abstrakce snáze implementují, simulace na vyšší úrovni abstrakce běží rychleji a umožňuje rychlejší odladění chyb. Abstraktnější model ale nezachycuje všechny detaily, a proto některé typy chyb nelze na vyšších úrovních abstrakce ani detekovat. Obvykle pracujeme na těchto úrovních:

- na behaviorální úrovni (*behavioral level*): navrhovaný systém je popsán chováním; jeden řádek kódu může být ekvivalentem i tisíců hradel. Model na behaviorální úrovni obvykle nepracuje s hodinami, neužívá pro popis chování stavové automaty a volně nakládá s potřebnou pamětí. Například dělička je implementována pouhým užitím operátoru dělení (/). Na behaviorální úrovni lze odladit správnou funkci algoritmů, které číslicový obvod realizuje. Obvykle pracujeme v jazyce C++, SystemVerilog, nebo VHDL.
- na RTL úrovni (*RTL simulation*, funkční simulace, *functional simulation*): návrh je popsán jako sada registrů propojených pomocí kombinačních obvodů. Jeden řádek kódu může být ekvivalentem desítek až stovek hradel, pomocí RTL kódu je modelována mikroarchitektura navrhovaného systému. Prováděná simulace nerespektuje reálná zpoždění na jednotlivých prvcích logiky, ale je stále velmi rychlá. Na RTL úrovni lze ověřit správnost implementace mikroarchitektury navrhovaného systému – zda je kód funkčně správně a obvod správně reaguje. Není ovšem možné ověřit korektnost interního časování (zpoždění na kombinačních prvcích, apod.). Pro modelování jsou k dispozici jednoduché aritmetické operátory (+, -, *), ale složitější operace (př. dělení) je třeba vymodelovat jako obvod realizující danou operaci (t.j. implementovat děličku). Typické jazyky používané pro RTL modelování jsou RTL (syntetizovatelné) podmnožiny jazyků VHDL a Verilog či SystemVerilog.
- na hradlové úrovni (*back-annotated gate level simulation*, *timing simulation*): simulujeme obvod po rozmístění a propojení (*place and route*), tedy finální podobu obvodu implementovaného ve zvolené technologii. Návrh je popsán ve formě schématu obvodu vyjádřeného ve zdrojovém kódu (*netlist*) a před simulací anotován informacemi o zpoždění na obvodových prvcích a spojích (*back annotation*). Několik řádků kódu běžně odpovídá i jen jednomu hradlu ve finálním obvodu. Typické jazyky používané na hradlové úrovni jsou VHDL a Verilog, více se o tomto popisu zmíníme v jednom z dalších článků.
- na tranzistorové úrovni (*transistor level*): simuluje se schéma obvodu sestavené z elementárních obvodových prvků, tranzistorů (realizovaných příslušnými modely) a zdrojů. Zatímco na vyšších úrovních abstrakce se pracuje s logickou jedničkou a logickou nulou, na tranzistorové úrovni simulátor pracuje přímo s obvodovými veličinami (proud a napětí) a řeší soustavy diferenciálních rovnic popisujících simulovaný obvod. To s sebou nese vyšší přesnost simulace, ale také významné zpomalení jejího běhu. Zatímco na behaviorální úrovni není problém simulovat celý navrhovaný systém, na tranzistorové úrovni lze za stejný čas odsimulovat jen malé bloky, případně jen kritické fáze běhu celého číslicového systému (například chování po zapnutí napájecího napětí). Jsou také potřeba jiné simulační programy (typicky *SPICE* a odvozené). Simulace na tranzistorové úrovni může například pomoci odhalit chyby v rozhraních mezi analogovými a číslicovými bloky v systému. I proto jsou simulace na tranzistorové úrovni prováděny při návrhu číslicových systémů na zákaznických integrovaných obvodech, ne při návrhu s použitím FPGA.

Moderní simulátory podporují současně více jazyků a více úrovní abstrakce v jednom návrhu. Zcela běžná je například kombinace hradlového popisu návrhu (*netlist*) ve Verilogu, RTL návrhu ve VHDL a verifikačního prostředí na behaviorální úrovni v SystemVerilogu, VHDL či Verilogu. Běžná je i možnost integrovat do běžící simulace kód napsaný v C++ (resp. SystemC) a dalších jazycích. Stejně dobře lze provádět i kosimulaci analogového a číslicového návrhu. Ta je potřebná pro simulaci celých systémů obsahujících jak analogové (například předzesilovače, AD převodníky, oscilátory, generátor resetu; simulovány na tranzistorové úrovni), tak číslicové bloky (simulovány na RTL úrovni). Simulace pak běží rychleji než čistě tranzistorová simulace celého systému.

3 Jak pracuje simulátor

Simulátory běžně používané pro simulaci číslicových obvodů jsou tzv. simulátory řízené událostmi (*event-driven*). Simulace řízená událostmi je používaná, protože je rychlá; simulátor se věnuje jen těm částem obvodu, které mění svůj stav. Událostí se označuje trojice *(uzel v obvodu, jeho nová hodnota, čas ve kterém uzel změní svoji současnou hodnotu na novou)*. Simulátor udržuje informaci o aktuálním čase, dále pro každý uzel simulovaného obvodu jeho současný logický stav a také frontu událostí, která obsahuje budoucí (naplánované) události. Události ve frontě jsou seřazeny podle jejich času. Z fronty jsou události vybírány a prováděny v chronologickém pořadí; v důsledku toho dochází v daném čase ke změně logického stavu v příslušném uzlu obvodu. Změna hodnoty v uzlu dále ovlivní všechny logické prvky, které mají tento uzel jako vstup (například procesy, které mají signál v citlivostním seznamu), je přepočítán jejich výstup a určen nový stav spolu s časem, kdy se do něj má výstup překlopit a nakonec je to vše ve formě nové události vloženo do fronty událostí.

Čas je v simulátoru řízeném událostmi složen ze dvou komponent. První složkou je současný čas v simulaci, který označíme t_c. Druhou složkou je číslo tzv. delta cyklu v rámci aktuálního časového okamžiku; každý časový okamžik může být rozdělen na libovolné množství nekonečně krátkých časových okamžiků, kterým říkáme delta cykly. Každá událost, která se vykonává ve stejném čase jako předchozí událost, se vykonává v jiném delta cyklu. Až poté, co se vykonají všechny události plánované na stejný reálný čas, se reálný čas t_c posune na další časový okamžik t_n. Ten je

daný nejbližší událostí ve frontě událostí k aktuálnímu času.

Práci simulátoru řízeného událostmi lze popsat takto [4]:

- *Před startem* vlastní simulace je nejprve načten popis návrhu. Této fázi se říká elaborace (*elaboration*). Během elaborace je zkompletována hierarchie návrhu, rozbaleny konstrukce *FOR GENERATE* a *IF GENERATE* a je zkontrolováno, zda žádné bloky nechybí. Pokud ano, elaborace je ukončena chybovým hlášením. Pak jsou všem uzlům v obvodu přiřazeny/dopočteny počáteční stavy. Pokud v důsledku této inicializace vznikne událost, bude zpracována během simulačního cyklu. Nakonec je zahájen simulační cyklus, počáteční čas $t_c=0$, příští časový okamžik t_n se nastaví podle kroku 3 simulačního cyklu (viz níže).
- *Během simulačního cyklu* simulátor postupně vykonává tyto kroky:
    1. Současný stav t_c je nastaven na čas t_n. Simulace je ukončena, pokud t_n=*TIME'HIGH* a současně nejsou naplánované žádné události na čas t_n.
    2. Každý aktivní signál v modelu je aktualizován a každý proces ve VHDL, který má ve svém citlivostním seznamu signál na kterém se vyskytla událost, je vykonán. V důsledku toho mohou vzniknout další události a být vloženy do fronty událostí.
    3. Čas příštího simulačního cyklu t_n se nastaví na nejbližší z časů:
        1. *TIME'HIGH*
        2. kdy je další budič ve frontě událostí aktivní,
        3. nebo další čas ve kterém se probudí některý z procesů v návrhu (uspaný např. pomocí příkazu WAIT).

        Pokud $t_n = t_c$, pak je další simulační cyklus delta cyklus.
    4. Simulátor pokračuje krokem 1.

Detailněji je celý proces simulace a elaborace popsán v [4] či [5].

Simulace řízená událostmi má některé zajímavé důsledky pro praktické použití:

- **Doba běhu simulátoru** se neodvíjí od toho, jak dlouhý časový úsek je třeba odsimulovat, ale roste jednak s velikostí návrhu (množstvím použitých jazykových konstrukcí a modelovaných obvodových prvků), a dále jak s počtem hodinových cyklů, které synchronní číslicový obvod zpracovává, tak s počtem událostí, které jsou modelovány na jednotlivých prvcích návrhu. Simulace číslicového návrhu s hodinami o frekvenci 1 GHz bude vyžadovat zpracovat řádově miliardy událostí na jednu sekundu simulovaného času. Simulace stejného návrhu s hodinami o frekvenci 100 MHz ale zpracuje jen řádově stovky miliónů událostí na jednu sekundu, za stejný reálný („lidský“) čas tedy proběhne desetkrát delší úsek simulace. Odsimulováno nicméně bude zhruba stejné množství hodinových cyklů za stejný reálný čas. Dále se simulace zpomaluje směrem k nižším úrovním abstrakce, kde je třeba modelovat více prvků detailněji. Konečně, modelování reálných zpoždění na hradlové úrovni vede na modelování statických a dynamických hazardů – více zpracovávaných událostí pak prodlužuje reálný simulační čas oproti simulaci na RTL úrovni.
- **Simulace je ukončena, když je fronta událostí prázdná** (*event starvation*, tzv. vyhladovění*)***.** Simulaci lze ukončit například pomocí příkazu jazyka VHDL
  ```
  ASSERT (false) REPORT „konec simulace‟ SEVERITY FAILURE;
  ```
  Lepší je ale přestat generovat hodiny pro simulovaný obvod a přestat stimulovat jeho primární vstupy. Od hodin a primárních vstupů jsou generovány další události v návrhu a pokud přestaneme budit hodiny a primární vstupy, přestanou se tyto dále generovat. Fronta událostí se postupně vyčerpá a simulace bude ukončena. Příklad tohoto přístupu lze najít v příkladu k tomuto článku (viz odstavec 6).
- **Události lze ukládat a zkoumat zpětně.** K tomu účelu simulátor vytváří tzv. *wave file*, do něj implicitně ukládá jen události na zobrazovaných signálech. Proto se po ukončení simulace můžeme dívat jen na signály, které byly do prohlížeče vlnek (*Wave window*) přidány na začátku/během simulace. Pokud chceme zpětně detailně zkoumat chování obvodu, lze zapnout ukládání událostí na dalších signálech v návrhu.

4 Typické simulační prostředí

Pro verifikaci číslicového návrhu používáme verifikační prostředí (*testbench*). Verifikační prostředí umožňuje provádět jednotlivé simulace (testy). Z pohledu verifikovaného návrhu simuluje vnější svět, ve kterém navrhovaný obvod pracuje. Dále prostředí provádí kontrolu správné funkce obvodu během běhu simulace srovnáním jeho odezev na stimuly s chováním modelu. Implementace automatické kontroly odezev obvodu umožňuje za všech podmínek objektivní kontrolu správné funkce systému nezávisle například na únavě návrháře. Verifikujeme-li i jen jednodušší obvod, automatická kontrola se stává zcela nezbytnou i z časových důvodů. Poslední důležitou funkcí verifikačního prostředí je monitorování funkčního pokrytí obvodu.

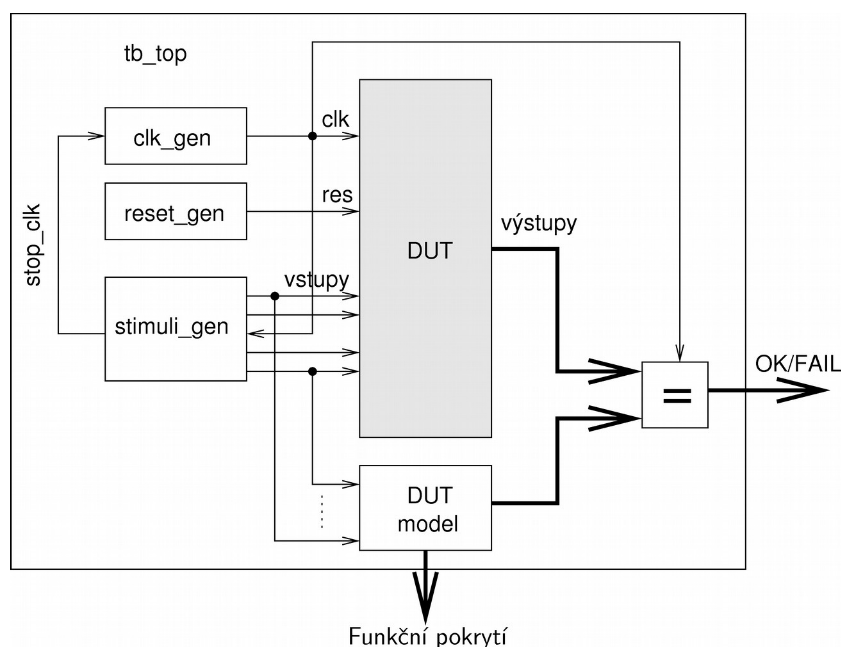
Verifikační prostředí může být implementované v jazyce VHDL pomocí behaviorálních konstrukcí jazyka (není určeno k syntéze, ale jen pro provádění simulací). Stejně dobře ho lze ale implementovat i v jiných jazycích – například v SystemVerilogu, který je svou podstatou určený pro implementaci komplexních verifikačních prostředí a pro verifikaci

složitých číslicových návrhů. K implementaci verifikačního prostředí lze také použít specializované knihovny, tzv. *frameworky* – například UVM (*Universal Verification Methodology*) [6].

Na obrázku 2 je zjednodušené schéma verifikačního prostředí. Blok nejvyšší úrovně (*top level*) *tb_top* obsahuje tyto bloky:

- **verifikovaný návrh** (*DUT, Design Under Test*): DUT je instance simulovaného číslicového návrhu. Obvykle to bývá buď RTL model navrhovaného obvodu, nebo jeho hradlová implementace.
- **generátor stimulů** (*stimuli_gen*)*:* řídí vstupní signály návrhu tak, aby byly postupně aplikovány všechny sekvence potřebné pro uspokojivou verifikaci číslicového návrhu. Generování stimulů může být deterministické, plně randomizované, nebo kombinované, viz např. [3], nebo [7].
- **model verifikovaného návrhu** (*DUT model*): model dostává z generátoru stimulů stejné vstupy jako verifikovaný obvod a na jejich základě vypočítává očekávané odezvy testovaného obvodu. Součástí modelu může být monitor funkčního pokrytí zaznamenávající, které funkce návrhu byly aktivovány v simulaci. Model typicky implementujeme na behaviorální úrovni.
- **komparátor** (=): komparátor vzájemně srovnává odezvy návrhu a jeho modelu. Pokud jsou stejné, je vše v pořádku, pokud jsou odlišné, je indikována chyba při simulaci. Srovnání odezev se obvykle provádí synchronně s vybranými hodinami v systému.
- **generátor hodin** (*clk_gen*): blok slouží ke generování periodického průběhu - hodinového signálu - pro testovaný obvod. Generátor hodin umožňuje také ukončení simulace. Když blok *stimuli_gen* indikuje konec simulace, ukončí *clk_gen* další generování hodin a tím zastaví generování dalších událostí. To ukončí simulaci.
- **generátor resetovacího signálu** (*reset_gen*): blok řídí resetovací signál pro verifikovaný návrh a návrh tak inicializuje po startu simulace.

Názvy jednotlivých bloků verifikačního prostředí se mohou v závislosti na použitém *frameworku* a metodice verifikace měnit, například dvojice DUT model a komparátor se při použití knihovny UVM a odpovídající metodiky nazývá *scoreboard*, generátor stimulů se pak rozpadá na test a sekvencéry, atd. Princip ovšem zůstává stále stejný. Podrobnější informace o verifikaci a implementaci verifikačních prostředí může čtenář nalézt na stránkách [8].



Obrázek 2: Zjednodušené verifikační prostředí. Soubor tb_top.tif.

5 Simulační nástroje

Pro provádění simulací je k dispozici široká škála nástrojů v různých výkonnostních i cenových kategoriích. V prvním přiblížení lze simulátory rozdělit na ty, které jsou volně dostupné bez licenčních poplatků a ty, za které je třeba zaplatit. Mezi volně dostupné simulační nástroje patří například ISim firmy Xilinx dostupný v rámci nástroje ISE WebPack či ModelSim PE ve verzi určené pouze studentům. Mezi placené nástroje patří například ModelSim firmy MentorGraphics (více viz např. [9]), NCSim firmy Cadence či VCS firmy Synopsys. Placená je i „plná“ vyšší verze ISim dodávaná v rámci prostředí ISE.

Verze simulátorů dodávané zdarma mají řadu zabudovaných omezení, díky kterým se nehodí pro profesionální práci (nepodporují např. výpočet verifikačních metrik, jsou proti placené verzi o řád až dva zpomalené, mohou obsahovat

omezení na velikost simulovaného návrhu, nemusí podporovat kombinaci více HDL jazyků v jednom návrhu, nepodporují např. jazyk PSL pro zápis assertions, atd.). Jsou ovšem stále dobře použitelné pro domácí kutily a studenty.

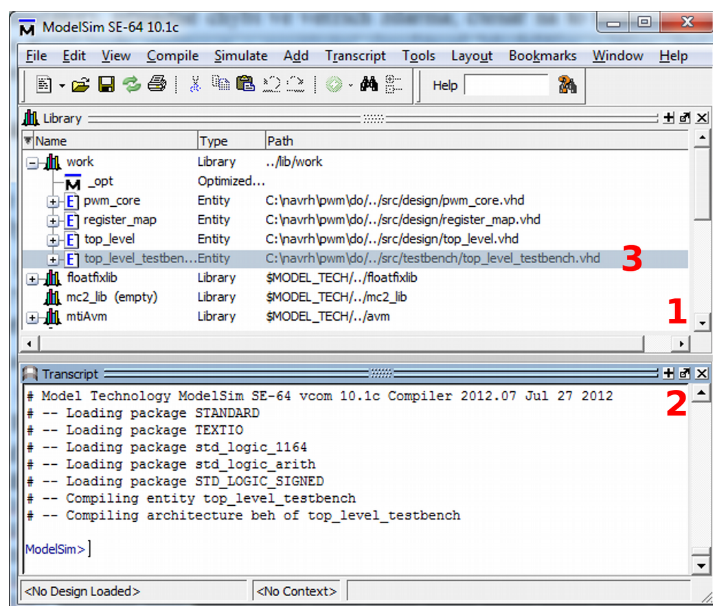
6 Základy práce se simulátorem ModelSim

Závěrem textu uveďme příklad základního postupu práce se simulátorem ModelSim při rozběhnutí RTL simulace. Podobně obšírný příklad pro ISim neuvádíme, lze ho najít v knize [1], kapitola 2.

Jak tedy postupovat:

1. z WWW adresy [10] si stáhněte demonstrační návrh – jednoduchý pulzně šířkový modulátor. Projekt rozbalte, například do adresáře `c:\navrh\pwm`. Po rozbalení prostudujte soubor `navrh.pdf` s popisem návrhu.
2. Spusťte ModelSim. V okně *Transcript* (označené 2 na obrázku 3) zadejte příkaz
   ```
   cd c:/navrh/pwm/do
   ```
   pro přepnutí do příslušného adresáře.
3. Před simulací je potřeba vytvořit knihovnu pro simulaci. Do knihovny se zkompiluje kód návrhu a knihovnu uvidíte i v okně *Library* (1 na obrázku 3). Vytvoření knihovny lze provést příkazy
   ```
   vlib ../lib/work
   vmap work ../lib/work
   ```
   Po zadání druhého z příkazů simulátor odpoví
   ```
   # Copying C:\mentor\modeltech64_10.1c\win64/../modelsim.ini to modelsim.ini
   # Modifying modelsim.ini
   # ** Warning: Copied C:\mentor\modeltech64_10.1c\win64/../modelsim.ini to modelsim.ini.
   #          Updated modelsim.ini.
   ```
   V aktuálním adresáři nyní najděte konfigurační soubor simulátoru `modelsim.ini`. Jeho prostudování je čtenáři doporučeno, je v něm řada komentářů a mnoho zajímavých nastavení nástroje. K některým z nich se vrátíme v dalších dílech.
4. Zkompilujte návrh. To lze mnoha různými způsoby, ve staženém balíčku je už připraven kompilační skript. Spusťte jej příkazem
   ```
   do compile_rtl.do
   ```
   Zde je čtenáři doporučeno obsah skriptu prozkoumat. Kompilaci lze provést také z menu simulátoru, *Compile* → *Compile*; najděte soubor, který chcete zkompilovat a klikněte na *Compile*. Všimněte si, že i v tomto případě je do okna *Transcript* vypsán příkaz, který kompilaci provede.
5. Spusťte vlastní simulátor. Lze tak učinit například poklepáním na entitu *top_level_testbench* (3 v obrázku 3) v okně *Library* myší. Všimněte si, že simulátor do okna *Transcript* vypsal a vzápětí provedl příkaz
   ```
   vsim work.top_level_testbench
   ```
   Ano, zadáním tohoto příkazu v okně *Transcript* lze také spustit simulaci.
6. Zobrazte si okno s časovými průběhy; to lze například pomocí menu *View* → *Wave*, nebo příkazem `view wave` zapsaným do okna *Transcript*. Do okna *Wave* si z okna *Objects* přetáhněte vybrané signály (okno *Objects* lze vyvolat také z menu *View*, nebo pomocí příkazu `view objects`).
7. Příkazem `run -all` spusťte kompletní simulaci až do konce.
8. Podívejte se, že pulzně šířkový modulátor funguje, jak má. Pokud si budete chtít přidat do okna *Wave* nový signál a podívat se na jeho průběh, zjistíte, že simulátor místo toho napíše do okna *-No Data-*. Je to proto, že do *wave file* se ukádají implicitně jen zobrazované vlnky. Problém může vyřešit například sekvence příkazů
   ```
   restart -f
   log -r /*
   run -all
   ```
   Objasnění jejich funkce je přenecháno čtenáři k zamyšlení (pomoci mu k tomu může dokument [11] přítomný v každé instalaci ModelSim). Pro simulátor ISim může sekvence příkazů vypadat například takto (více viz [12]):
   ```
   restart
   wave log -r /
   run all
   ```
   Jak vidíte, rozdíly mezi základní prací s různými simulátory opravdu nebývají veliké.

Popsaný postup je jen jeden z mnoha; jeho výhodou je názornost jednotlivých kroků. Jinou možností je použít pro nastavení RTL simulace projektu, více viz např. [13].



Obrázek 3: ModelSim - hlavní okno s knihovnami a konzolí simulátoru.Soubor modelsim_okno.png.

7 Závěr

Článek shrnul základní vlastnosti simulace řízené událostmi a stručně popsal jejich dopad na návrhářovu práci. RTL simulace je nicméně jen prvním krokem při návrhu. K úspěšné práci je třeba ovládnout i postupy práce se simulátorem související s hradlovou simulaci, měřením návrhu, atd. Těm budou věnovány následující díly seriálu. Závěrem bych rád poděkoval Ing. Janu Kubákovi za svolení použít jeho semestrální práci, jež se po úpravě stala příkladem použitým v tomto článku.

8 Použitá literatura

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

[2] J. Pinker, M. Poupa. Číslicové systémy a jazyk VHDL. BEN Praha 2006

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

[4] 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)

[5] Michael John Sebastian Smith. Application Specific Integrated Circuits. Addison-Wesley, 1997. Kapitola 13 [online: http://www10.edacafe.com/book/ASIC/ASICs.php] (kontrolováno 29.10.2014)

[6] Accelera. Universal Verification Methodology. http://www.accellera.org/downloads/standards/uvm

[7] J. Šťastný. Verifikace pomocí assertions: seznámení, DPS Elektronika od A do Z, listopad/prosinec 2012, pp. 4-8, 2012.

[8] Mentor Graphics. Verification Academy web site [online: https://verificationacademy.com/] (kontrolováno 29.10.2014)

[9] Petr Matějka, ModelSim – klasika, která nestárne. DPS elektronika od A do Z, březen/duben 2014, str. 22 – 23.

[10] Ukázkový příklad [online: http://amber.feld.cvut.cz/fpga] (kontrolováno 20.11.2014)

[11] 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.

[12] 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)

[13] ModelSim tutorial [online: http://amber.feld.cvut.cz/fpga/fpga_lab_resources/msim_certification/msim_demos.html] (kontrolováno 4.11.2014)