8. VYUŽITIE MODELSIMU V ĎALŠÍCH ETAPÁCH NÁVRHU

V predchádzajúcom cvičení boli demonštrované základné vlastnosti (funkčná simulácia) a výhody (využitie dávkového spracovania) ModelSimu. V ModelSime však je možné efektívne realizovať aj ďalšie etapy overovania návrhu. Je to predovšetkým využitie tzv. **testbenchov** na zautomatizovanie overovania (testovania) návrhu a využitie ModelSimu aj v **časovej simulácii**. Uvedené činnosti patria do kategórie pokročilejších testovacích techník a vytvárajú z Modelsimu testovacie prostredie pomocou ktorého je možné veľmi efektívne realizovať všetky etapy[1] testovania návrhu.

V rámci cvičenia bude demonštrovaná resp. precvičovaná nasledujúca problematika:

- využitie príkazu generate vo VHDL,
- zvýšenie modularity návrhu VHDL s využitím **balíkov** (packages),
- automatické testovanie s využitím testbenchov,
- **časová simulácia** v ModelSime,
- ďalšie príklady vo VHDL.

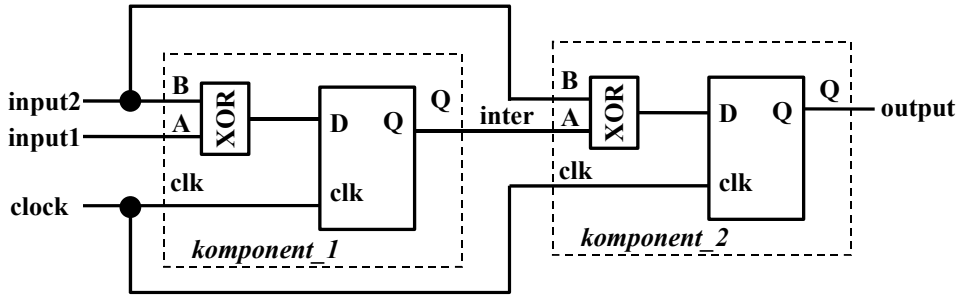
8.1 UŽITOČNÉ KONŠTRUKCIE JAZYKA VHDL

V tejto podkapitole je opísaný príkaz **Generate** a využitie **balíkov** (packages), ktoré umožňujú vo väčších návrhoch zvýšiť prehľadnosť VHDL kódu. Aj keď pre jednoduché projekty ktoré boli doteraz preberané na cvičeniach sa môže zdať ich využitie zbytočne komplikované, je potrebné ich vedieť aspoň „čítať“, t.j. vedieť pochopiť ich význam získať tak možnosť pochopiť napr. význam VHDL kódov od iných autorov.

[1] V prípade časovej simulácie musí ModelSim využívať informácie o cieľovej ASIC alebo FPGA technológii. Musí teda spolupracovať minimálne s nástrojmi na P&R od jednotlivých výrobcov (v našom prípade Quartus II) a umožňovať využitie informácii o cieľovej technológii, ktorú výrobcovia poskytujú vo forme špeciálnych VHDL knižníc. Všetci významní výrobcovia ASIC a FPGA obvodov vzhľadom na rozšírenie ModelSimu poskytujú aj automatickú podporu ModelSimu priamo z ich návrhových prostredí. Niektoré činnosti (napr. časovú simuláciu) je však možné realizovať len pomocou plnej verzie ModelSimu, prípadne verzie ModelSimu pre konkrétneho výrobcu (napr. ModelSim-Altera, ModelSim-Xilinx, ModelSim-Actel, ...). Časovú simuláciu v obvodoch Altera tak nebude možné realizovať pomocou verzie Modelsim-Xilinx (a platí to samozrejme aj opačne).

8.1.1 PRÍKAZ GENERATE

Na minulom cvičení bola demonštrovaná možnosť využitia VHDL komponentov pri modulárnom návrhu. Zložitejší návrh vytvorený kaskádnym zapojením dvoch identických častí, ktoré boli mapované do celého návrhu je zobrazený na obr.1.



Obr.1: Využite 2 komponentov v modulárnom VHDL návrhu

Dva identické komponenty boli do výsledného návrhu mapované následovnými VHDL príkazmi:

```
komponent_1 : komponent port map (A=>input1, B=>input2, clk=>clock, Q=>inter);
komponent_2 : komponent port map (A=>inter, B=>input2, clk=>clock, Q=>output);
```

V praktických návrhoch vzniká často potreba využívať aj väčšie množstvo identických komponentov, ktorých počet je dokonca potrebné **parametrizovať**. Predchádzajúca stratégia využitia komponentov by tak bola pomerne nepraktická, pretože pri každej zmene počtu komponentov by bolo nevyhnutné meniť zdrojový VHDL kód pridávaním resp. vynechávaním príkazov na mapovanie. Naviac, v prípade využitia veľkého počtu komponentov (v praktických návrhoch to často môžu byť desiatky identických komponentov) by zdrojový kód začal byť neprehľadný. Uvedený problém je možné elegantne vyriešiť použitím VHDL príkazu **generate**.

Príklad 1
Zapojte parametrizovateľný počet komponentov z obr.1 kaskádne za sebou.

Nasledujúci VHDL kód demonštruje riešenie s využitím príkazu **generate**.

```
library ieee;
use ieee.std_logic_1164.all;

entity use_generate is
generic
        (
        constant SIZE   : natural          := 10
        );
port
        (
        input1  : in      std_logic;
        input2  : in      std_logic;
        clock   : in      std_logic;
        output  : out     std_logic
        );
end entity use_generate;

architecture use_generate_arch of use_generate is
        component xor_plus_dff is
        port
                (
                A       : in      std_logic;
                B       : in      std_logic;
                clk     : in      std_logic;
                Q       : out     std_logic
                );
        end component;

        signal inter      : std_logic_vector(1 to SIZE);
       begin
        schema: for i in 1 to SIZE generate
                prvy_clen       : if i=1 generate
                        prvy    : xor_plus_dff port map (A=>input1, B=>input2, clk=>clock, Q=>inter(1));
                end generate prvy_clen;

                zvysne_cleny    : if i/=1 generate
                        dalsie  : xor_plus_dff port map (A=>inter(i-1), B=>input2, clk=>clock, Q=>inter(i));
                end generate zvysne_cleny;
        end generate schema;
        output <= inter(SIZE);
end architecture use_generate_arch;
```

Hodnota `constant SIZE : natural := 10` určuje počet komponentov, ktoré sa majú kaskádne[2] zapojiť. Jednoduchou zmenou tejto konštanty je možné meniť počet komponentov, ktoré budú mapované do výsledného návrhu. V predchádzajúcom cvičení, kde boli prepojené navzájom len dva komponenty, bolo potrebné vytvoriť len jeden vnútorný signál *inter*. Pretože konštanta SIZE určuje počet mapovaných komponentov, je v súčasnom riešení od nej odvodená aj veľkosť vektora

```
signal inter      : std_logic_vector(1 to SIZE);
```

K jednotlivým prvkom (bitom) tohto vektora je možné vo VHDL konštrukciách pristúpiť pomocou indexu *i*.
Prvým **generate** príkazom je **"for - generate"** ktorý vykoná príkazy obsahujúce v tele tohto príkazu *SIZE* – krát.

[2] Konštrukciu generate je samozrejme možné použiť aj pre iné „zapojenia" komponentov. Konkrétne „zapojenie" je definované použitým mapovaním a VHDL príkazmi.

```
schema: for i in 1 to SIZE generate
        -- telo príkazu for-generate
end generate schema;
```

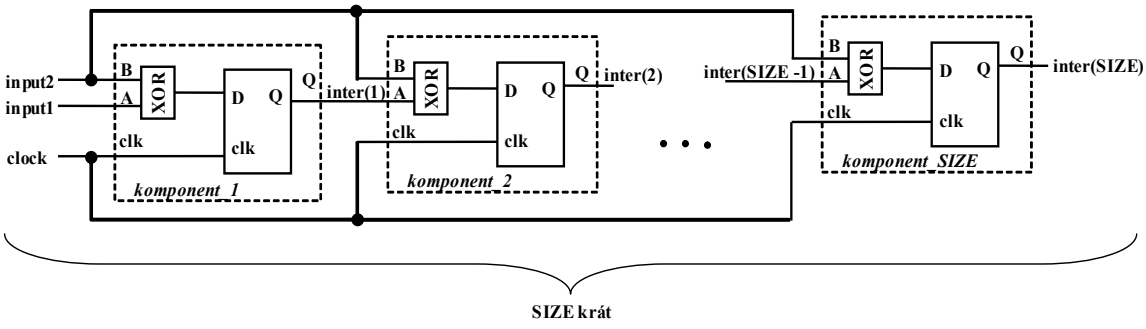
Z obr.2 na ktorom je znázornená grafická reprezentácia kaskádneho zapojenia identických komponentov je vidno, že prvý komponent je v porovnaní so zvyšnými komponentmi zapojený odlišne. Na signál *A* prvého komponentu sa mapuje signál *input1*, a nie niektorý z vnútorných bitov signálu *inter*. Na to, aby sa v tele príkazu ***for*-generate** určilo, ktorý z komponentov sa práve mapuje je využitý príkaz **if-generate**, ktorého vhodným použitím dosiahneme vytvorenie požadovanej parametrizovateľnej topológie zapojenia.

```
prvy_clen       : if i=1 generate
        prvy    : xor_plus_dff port map (A=>input1, B=>input2, clk=>clock, Q=>inter(1));
end generate prvy_clen;

zvysne_cleny    : if i/=1 generate
        dalsie  : xor_plus_dff port map (A=>inter(i-1), B=>input2, clk=>clock, Q=>inter(i));
end generate zvysne_cleny;
```

Nakoniec je posledný vnútorný signál *inter* pripojený na výstupný port návrhu:

```
output <= inter(SIZE);
```



Obr.2 Grafická reprezentácia kaskádneho zapojenia parametrizovateľného počtu identických komponentov

8.1.2 VYUŽITIE VHDL BALÍKOV (PACKAGES)

Okrem rozdelenia celého návrhu do menších častí (modulov - komponentov), je možné (pri modulárnom projekte dokonca žiadúce) logicky súvisiace veličiny, resp. funkcie a pod. uložiť do samostatných balíkov (*názov_balika.vhd*), ku ktorým je možné v návrhu pristupovať po ich inicializácii (deklarácii) v tvare:

```
library work;
use work.package_name.all;
```

Ak sú napr. v návrhu používané veľké počty konštánt a návrh je rozčlenený do menších modulov, bolo by pomerne komplikované meniť konštanty v každom z modulov. Naviac, medzi jednotlivými konštantami môžu existovať súvislosti a v najhoršom prípade by bolo potrebné vedieť, presne v ktorom module sa daná konštanta, ktorú chceme zmeniť nachádza resp. s akou inou konštantou súvisí. Preto je vhodné všetky konštanty a prípadné súvislosti uložiť do jedného balíka. Príklad VHDL kódu balíka, ktorý zlučuje viacero konštánt je uvedený nižšie:

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.functions.all;

package constants is

        constant SIZE           : integer       := 100;
        constant LENGTH         : positive      := 7;

        type state is (state1, state2, state3, state4);

        constant stav    : state   := state1;

        constant offset  : positive        := get_offset(SIZE, LENGTH);
end package constants;
```

Všimnite si predposledný riadok výpisu, v ktorom je deklarovaná a nakoniec aj definovaná konštanta *offset*. Táto konštanta využíva funkciu *get_offset*, ktorá sa nachádza v ďalšom z balíčkov označeného ako *functions*, sprístupneného kľúčovým slovom *use*.
Aby bolo jasné, akú funkciu *get_offset* funkcia vykonáva, je uvedený aj VHDL kód balíka *functions:*

```
library ieee;
use ieee.std_logic_1164.all;

package functions is
        function get_offset(SIZE : integer; LENGTH : positive) return positive;
end functions;

package body functions is
        function get_offset(SIZE : integer; LENGTH : positive) return positive is
        begin
                return (SIZE - LENGTH);
end functions;
```

Medzi kľúčovými slovami "package functions is" a "end functions" je deklarovaná funkcia, ktorá je využitá v tele balíka predchádzajúceho VHDL kódu.

8.2 VERIFIKÁCIA NÁVRHU S VYUŽITÍM TESTBECHOV

Cieľom tejto podkapitoly je vysvetliť podstatu využitia **testbenchov** na verifikáciu FPGA návrhov opísaných vo VHDL. Testbenche je možné podobne ako samotný jazyk

VHDL použiť na opis a následnú verifikáciu opisu omnoho širšej množiny návrhov z rôznych (aj netechnických) oblastí. Doteraz preberané VHDL konštrukcie využívali len **menšiu** časť tzv. **syntetizovateľných**[3] príkazov VHDL. V prípade testbenchov však budú využité aj **nesyntetizovateľné**[4] konštrukcie ako je napríklad časové oneskorenie o presne definovaný časový úsek.

Taktiež budú opísané základné príkazy využiteľné pri vytváraní testbenchov. Tieto príkazy spolu s príkladmi umožnia realizáciu praktických testbenchov pre návrhy preberané v rámci cvičení.

8.2.1 PRINCÍP VYUŽITIA TESTBENCHOV

Vzhľadom na zväčšujúcu sa veľkosť návrhov a nárast ich zložitosti sa verifikácia, teda overenie funkčnosti návrhu stáva pomerne zložitou úlohou. Na to, aby bolo možné overiť aj takéto zložité návrhy, je často potrebné využiť niektorý z verifikačných nástrojov, resp. metód. Pre veľké návrhy, ktoré pozostávajú z niekoľko miliónov ekvivalentných hradiel, je snaha využívať formálne verifikačné nástroje (formal verification tools). Avšak, pre menšie a jednoduchšie návrhy je často výhodnejšie využitie HDL simulátorov (napr. ModelSimu) v kombinácii s tzv. testbenchmi.

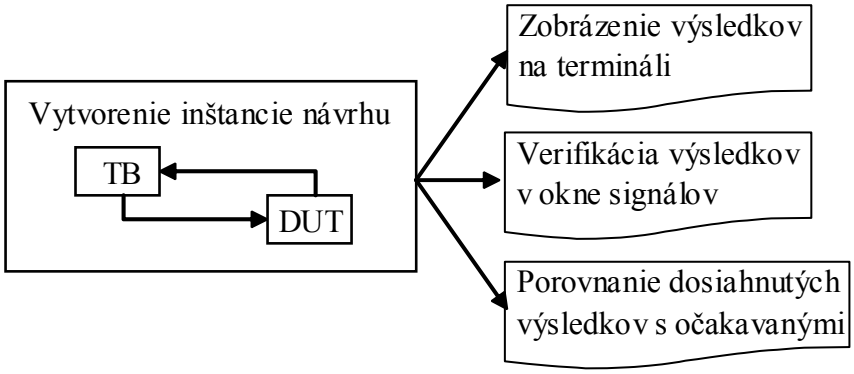
Testbenche sa stáli jednou zo štandardných metód na overenie HLL (**H**igh-**L**evel **L**anguage, do ktorých patrí aj VHDL) návrhov. Ich principiálna funkcia je znázornená na obr.3. Vo všeobecnosti testbenche vykonávajú nasledujúce úlohy:

- vytvárajú inštanciu testovaného návrhu DUT (**D**esign **U**nder **T**est),
- simulujú DUT, a to tým spôsobom, že aplikujú testovacie vektory na testovaný model,
- vypisujú výsledky na terminál, resp. v okne signálov (waveform window), ktoré je možné vizuálne overiť,
- taktiež môžu porovnávať aktuálne (dosiahnuté) výsledky s očakávanými, ktoré môžu byť uložené napr. v súboroch s referenčnými hodnotami.

Testbenche je možné napísať buď v jazyku VHDL, resp. Verilog. Samozrejme záleží na návrhárovi, pre ktorý z jazykov sa rozhodne. Najčastejšie sa predpokladá, že pokiaľ je návrh realizovaný v jednom jazyku, je aj testbench napísaný v tom istom jazyku. Niektoré simulátory nedokážu kombinovať návrhy napísané v rozdielnych jazykoch. Zložitejšie testbenche obsahujú ďalšie dodatočné operácie, ako napr. logiku určujúcu konkrétne stimuly pre overenie návrhu, resp. porovnávanie aktuálnych výsledkov s očakávanými.

[3] Týmto príkazom je možné priradiť ekvivalentnú hardvérovú realizáciu napr. v obvodoch FPGA.

[4] Tieto konštrukcie sú z pohľadu jazyka VHDL úplne korektné, akýkoľvek pokus o ich priamu syntézu pomocou prostriedkov pre návrh v obvodoch ASIC prípadne VHDL však bude viesť k neúspechu. Napr. oneskorenie signálu o presne definovaný časový interval je v reálnom hardvéri nemožné, pretože v skutočnosti závisí od mnohých technologických faktorov ako je teplota, napájanie a pod.



Obr.3 Princíp využitia testbenchov

8.2.2 ŠTRUKTÚRA VHDL TESTBENCHOV

Vo všeobecnosti môže byť testbench napísaný v jazyku VHDL alebo Verilog. V ďalšej časti sa obmedzíme na testbenche vo VHDL. Všetky testbenche sa skladajú so základných sekcií, a to:

- deklarácia entity a architektúry,
- deklarácia signálov,
- vytvorenie inštancie „Top-level“ návrhu,
- vytvorenie stimulov.

Typický testbench ktorý verifikuje synchrónny DUT návrh, resp. návrh, ktorý využíva systémové hodiny k riadeniu vnútornej logiky, obsahuje generátor hodinového signálu. Nasledujúci príklad ukazuje možný spôsob (s využitím **nesyntetizovateľných** VHDL príkazov!) generovania hodinového signálu.

```
constant ClockPeriod : time := 10ns;      -- deklaracia periody hodinoveho signalu
clk_gen : process                         -- generovanie hodinoveho signalu
        begin
                clk <= '0';
                loop
                        wait for (ClockPeriod / 2);
                        clk <= not clk;
                end loop;
        end process clk_gen;
```

Na to, aby bolo možné overiť, či daný DUT funguje správne, je potrebné na vstupy DUT priviesť vstupné stimuly. V testbenchoch sa používajú paralelne vykonávané bloky stimulov tak, aby boli vytvorené požadované vstupné signály. Existujú dve metódy určenia času počiatku generovaného stimulu:

- absolute-time stimuly,

- realative-time stimuly.

V **absolútnej metóde** sa hodnoty simulácie vzťahujú k času simulácie t=0. **Časovo relatívne stimuly** inicializujú počiatočné hodnoty, následne čakajú na udalosť, ktorá určí stimulom iné hodnoty. Jeden testbench môže s výhodou kombinovať obe metódy (záleží len na požiadavkách návrhára, resp. na spôsobe ako sa rozhodol vytvoriť testbench).

V nasledujúcom príklade sú využité časovo absolútne definované (**time-absolute**) stimuly:

```
-- time-absolute stimuly
time_abs_process: process
begin
        reset <= '1';
        load <= '0';
        wait for 100 ns;
        reset <= '0';
        wait for 20 ns;
        load <= '1';
end process
```

Uvedený proces definuje blok stimulov. Najskôr sa nastaví signál reset na hodnotu log1 a súčasne sa nastaví signál load na hodnotu log0. Toto priradenie sa vykoná v čase t=0, teda úplne na začiatku simulácie. Následne sa čaká 100 ns, počas ktorých si signály reset a load zachovávajú svoje hodnoty. Po uplynutí 100 ns zmení signál reset svoju hodnotu z log1 do stavu log0 a čaká sa ďalších 20 ns, po uplynutí ktorých sa zmení hodnota signálu load z log0 do log1. Týmto spôsobom sa dajú definovať vstupné vektory pre DUT, t.j. tvary vstupných signálov (stimulov).

V nasledujúcom príklade sú využité časovo relatívne (**time-relative**) stimuly:

```
-- time-relative stimuly
first_stimulus: process (clock)
        begin
                if clock'event and clock='1' then
                tb_count <= tb_count + 1;
        end if;
end process;
```

```
second_stimulus: process
begin
        if (tb_count <= 5) then
                reset <= '1';
                load <= '0';
                else
                reset <= '0';
                load <= '1';
        end if;
end process;

final_stimulus: process
        begin
                if (count = "1100") then
                        report "Citac dosiahol hodnotu 1100"
                end if;
end process;
```

Jednotlivé *process* bloky sú vykonávané súčasne, spolu s inými inicializačnými blokmi v súbore. Avšak vnútri každého bloku (inicializačný alebo process), sa jednotlivé udalosti vykonávajú sekvenčne, v poradí v akom sú zapísané. To znamená, že sekvencia stimulov v každom bloku začína v čase t=0, a záleží potom na podmienkach, pri splnení ktorých sa stimuly opäť vyhodnotia. V uvedenom príklade máme celkovo 3 procesy, a každý z nich reaguje na inú udalosť. Prvý z procesov reaguje na signál clock (hodinový signál), resp. presnejšie, reaguje na nábežnú hranu hodinového signálu clock, a pri každej takejto zmene sa zvýši hodnota tb_count o 1. Tento signál následne využíva druhý process, ktorý testuje jeho hodnotu a v prípade ak je stav čítač=5, vykoná zodpovedajúce priradenia hodnôt signálom reset a load, inak týmto signálom priradí iné konkrétne hodnoty. V poslednom procese sa na terminál príkazom

```
report "Citac dosiahol hodnotu 1100"
```

vypíše určitá správa v prípade ak čítač count nadobudol hodnotu "1100".

8.2.3 ZÁKLADNÉ PRÍKAZY PRE ZOBRAZOVANIE VÝSLEDKOV

Na rozdiel od jazyka Verilog, ktorý disponuje kľúčovými slovami **$display** a **$monitor**, ktoré umožňujú zobrazenie výsledkov, jazyk VHDL neobsahuje ekvivalentné príkazy. Avšak v balíku *std_textio* ktorý umožňuje presmerovanie vstupov/výstupov na terminál.

Príkaz **report "sprava"** zobrazí text *sprava* na terminály.

Ďalším z možných príkazov, ktorý je možné výhodne využiť v simuláciách je príkaz **assert**. Tento príkaz kontroluje, či daná podmienka je pravdivá, a ak nie je, vykoná nejakú akciu. Obsahuje dve voľby: **report** a **severity**, pričom je možné použiť ľubovoľné z nich samostatne, alebo súčasne.

report – zobrazí užívateľom definovanú správu, v prípade ak podmienka nie je pravdivá

severity – dovoľuje užívateľovi stanoviť úroveň dôležitosti, v prípade ak podmienka nie je pravdivá

Existujú štyri možné úrovne závažnosti: **note**, **warning**, **error** a **failure**, ktoré sa vo všeobecnosti využívajú v procese simulácie, v prípade ak podmienka nie je pravdivá. Napr. hodnota *failure* sa dá využiť k zastaveniu vykonávania simulácie.

```
assert <podmienka>
  report "ak nie je podmienka pravdiva, vypise sa tento text"
  severity error;
```

8.2.3.1 OTVORENIE SÚBORU

Často je užitočnejšie výsledky zapisovať/čítať do/zo súboru. Takto je možné realizovať rozsiahlejšie simulácie pomocou dávkového spracovania, prípadne realizovať prepojenie simulácie v Modelsime s inými nástrojmi (Matlab, Excel, …). Premennú typu súbor je možné deklarovať nasledovne:
- ak sa jedná o súbor, z ktorého chceme vyčítavať určité hodnoty, jedná sa o vstupný súbor a použijeme nasledujúci zápis

```
FILE subor: TEXT open READ_MODE is "vstupny_subor.txt"
```

- ak sa jedná o súbor, do ktorého chceme zapisovať výsledky simulácie, resp. iné relevantné hodnoty, potom v zápise nahradíme kľúčové slovo **READ_MODE** za slovo **WRITE_MODE**, v tvare

```
FILE vysledky: TEXT open WRITE_MODE is "vysledky.txt"
```

8.2.3.2 ČÍTANIE ZO SÚBORU[5]

Ak už máme definovaný „handle" súboru, teda deklaráciu premennej, ktorá ukazuje na príslušný súbor, ktorý bol definovaný ako vstupný, je možné zo súboru vyčítavať jednotlivé riadky. Na vyčítanie jedného riadku zo súboru slúži príkaz **readline**.

```
readline(subor, riadok);
```

kde *subor* je handle vstupného súboru, v ktorom sa nachádzajú údaje, ktoré je potrebné vyčítať a *riadok* je premenná, do ktorej sa uloží vyčítaný riadok. Táto premenná sa deklaruje ako:

```
variable riadok: line;
```

[5] Zo súboru je napr. možné čítať vstupné stimuly, ktoré napr. môže automaticky vygenerovať iný nástroj (napr. Matlab, C-program, ...).

V nasledujúcom príklade predpokladajme, že každý riadok vo vstupnom súbore pozostáva z dvoch hodnôt, z dekadického čísla a z binárneho čísla, napr.

0 10100010
10 11100110
20 00110100
....

Na vyčítanie jednotlivých hodnôt je použitá funkciu **read** ktorá má tri vstupné parametre, a to názov premennej ktorá bude obsahuje načítaný riadok (v našom prípade je to **line**), druhým parametrom je názov premennej, do ktorej sa vyčítané desiatkové číslo zapíše, napr. **cislo**. Premenná **cislo** je deklarovaná ako: variable cislo: real;
Posledným parametrom (voliteľným) je premenná, do ktorej sa uloží výsledok operácie načítania hodnoty. To znamená, že ak sa z riadku vyčítala hodnota správne, teda že nedošlo k žiadnej chybe, táto premenná obsahuje hodnotu TRUE.

read(riadok, cislo, dobre_cislo);

Premenná **dobre_cislo** je deklarovaná ako: variable dobre_cislo: boolean;

Následne je potrebné vyčítať do nejakej premennej typu *character* znak medzery, ktorý nasleduje za dekadickým číslom. Potom je možné vyčítať binárny vektor, ktorý sa uloží do premennej typu *std_logic_vector*. Veľkosť vektora je 8 bitov.
Test, či ešte nebol dosiahnutý koniec súboru sa realizuje pomocou funkcie

```
endfile(handle_suboru);
```

Napríklad tento test je možné použiť v cykle *while*

```
while not endfile(subor) loop
<prikazy vycitania>
end loop;
```

8.2.3.3 ZÁPIS DO SÚBORU

V prípade zápisu do súboru sa v podstate jedná o formálne ten istý zápis, ako to bolo v prípade čítania zo súboru. Slovo **read** je však nahradené slovom **write**. Detaily sú vysvetlené v nasledujúcich praktických príkladoch testbenchov.

8.2.4 PRAKTICKÉ PRÍKLADY TESTBENCHOV

Príklad 2
Vytvorte testbench pre VHDL kód čítača vpred, ktorý bol vytvorený na jednom z predchádzajúcich cvičení.

Najskôr definujeme knižnice potrebné v analyzovanom návrhu

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

use ieee.std_logic_textio.all;
use std.textio.all;
```

Význam prvých 3 knižnic bol vysvetlený v rámci predchádzajúcich cvičení. Posledné dve knižnice doteraz neboli využívané. Tieto knižnice sú potrebné, ak sa využíva zápis, resp. vyčítavanie údajov do/zo súboru.
Nasleduje definícia entity[6] testbencha, kde je definovaná hodnotu *N* ktorá určuje veľkosť čítača.

```
entity counter_up_tb is
generic
        (
                N : integer:= 9
        );
end entity counter_up_tb;
```

Podobne ako v doterajších VHDL konštrukciách, po definícii entity nasleduje definícia architektúry testbencha, v ktorom je ako prvý deklarovaný komponent čítača, pre ktorý je daný testbench vytváraný.

```
component counter_up is
generic
        (
                N : integer:= 9
         );
 port
       (
        clk       : in     std_logic;
        reset     : in     std_logic;
        vystup    : out    std_logic_vector (N downto 1)
        );
end component;
```

Následne je definovaná konštanta, ktorá je využívaná pri generovaní hodinového signálu:

```
constant clk_period   : time := 50 ns;
```

V tele architektúry je najskôr vytvorená inštancia testovaného čítača:

[6] Je dôležité upozorniť na to, že testbench nemá definované žiadne vstupné a výstupné signály. V návrhoch pre syntézu sa s podobným javom nestretávame.

```
DUT: counter_up port map (
  clk => clk,
  reset => reset,
  vystup => vystup
  );
```

a následne vytvorený hodinový signál požadovanej periódy, ktorá je určená konštantou clk_period:

```
clk_gen : process
        begin
                clk <= '0';
                loop
                        wait for (clk_period / 2);
                        clk <= not clk;
                end loop;
end process clk_gen;
```

Nakoniec sú zapísané výsledky simulácie do súboru:

```
stimulus : process
variable riadok   : LINE;
file results      : TEXT open WRITE_MODE is "results.txt";
        begin
                write(riadok, now);                              - - aktualny cas
                write(riadok, string'(" vystup = "));
                write(riadok, vystup);
                writeline(results, riadok);
                wait for clk_period;
end process stimulus;
```

Uvedený proces deklaruje premennú typu *LINE*, do ktorej budú zapisované jednotlivé hodnoty, a v konečnom dôsledku, v ktorej bude zapísaný jeden riadok, ktorý sa následne zapíše do súboru. Kľúčové slovo **now** obsahuje aktuálny čas simulácie. Po zapísaní jedného riadku do súboru sa čaká jednu periódu, kým sa nezopakuje celý proces odznova. Tvar výstupného súboru je následovný:

0 ns vystup = UUUUUUUUU
50 ns vystup = 000000001
100 ns vystup = 000000010
150 ns vystup = 000000011
.....

Pre ucelenosť nasleduje výpis celého VHDL kódu:

```
library ieee;                                   --zadefinovanie kniznic
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
```

```
--definovanie vstupov a vystupov
entity counter_up_tb is
generic
        (
                N : integer:= 9
        );
end entity counter_up_tb;

architecture counter_up_tb_arch of counter_up_tb is

component counter_up is
generic
         (
                N : integer:= 9
         );
  port
         (
         clk       : in    std_logic;                                    --hodiny
         reset     : in    std_logic;                                    --inicializcia citaca
         vystup    : out   std_logic_vector (N downto 1)
         );
        end component;

        signal clk          : std_logic;
        signal reset        : std_logic;
        signal vystup       : std_logic_vector (N downto 1);

        constant clk_period   : time := 50 ns;

begin

DUT: counter_up port map (
                                clk => clk,
                                reset => reset,
                                vystup => vystup
                        );


--Clock signal generation
clk_gen : PROCESS
BEGIN
        clk <= '0';
        LOOP
                WAIT FOR (clk_period / 2);
                clk <= NOT clk;
        END LOOP;
END PROCESS clk_gen;

stimulus : process
VARIABLE riadok   : LINE;
FILE results      : TEXT OPEN WRITE_MODE IS "results.txt";
        begin
                write(riadok, now);                             - - aktualny cas
                WRITE(riadok, string'(" vystup = "));
                WRITE(riadok, vystup);
                WRITELINE(results, riadok);
                wait for clk_period;
end process stimulus;

end architecture counter_up_tb_arch;
```

Dávkový súbor, ktorý realizuje kompiláciu potrebných súborov[7], ako aj konečnú simuláciu by mohol vyzerať napr. následovne[8]:

quit -sim
vcom -work work -2002 -explicit -novitalcheck -no1164 -novital counter_up.vhd counter_up_tb.vhd
vsim work.counter_up_tb
view wave
do wave.do
restart -f
run 2000

Pritom uvažujeme, že vektorový súbor *wave.do* sme si už predtým vytvorili, jedným zo spôsobov popísaných na predchádzajúcom cvičení.
Upozornenie: výstupný súbor *result.txt* sa vytvorí až po uzavretí simulácie napr. príkazom *quit -sim*

Príklad 3
Vytvorte testbench na automatické overenie správnej funkčnosti VHDL kód prevodníka grayovho kódu plus 3, ktorý bol vytvorený na jednom z predchádzajúcich cvičení.

Jednou z najdôležitejších vlastností použitia testbencha na kontrolu návrhu je to, že ho je možné použiť na porovnanie dosiahnutých výsledkov s referenčnými hodnotami, teda s hodnotami, ktoré považujeme za správne. V konkrétnom súbore môžu byť v určitej forme zapísané hodnoty, ktoré by mala testovaná realizácia dosiahnuť, za predpokladu že funguje správne. Tieto hodnoty je možné vyčítať zo súboru pomocou funkcie **readline**, ktorá vyčíta jeden riadok zo súboru. Na tento riadok je následne možné aplikovať funkciu **read**, ktorá vyčíta jednotlivé elementy riadku.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;
use std.textio.all;

entity gray_to_plus3_tb is
end entity gray_to_plus3_tb;
```

[7] Najskôr je potrebné štandardným spôsobom vytvoriť projekt obsahujúci príslušné VHDL súbory.
[8] V starších verziách môže byť problém so zvolenou VHDL verziou –2002. V prípade potreby je samozrejme potrebné modifikovať príslušné súbory.

```
architecture testbench of gray_to_plus3_tb is

component gray_to_plus3 is
port
     (
              gray_code             :  in     std_logic_vector (4 downto 1);
              plus3_code            :  out    std_logic_vector (4 downto 1);
              ostatne_segmenty      :  out    std_logic_vector (11 downto 0)
              );
end component;

          signal gray_code             : std_logic_vector (4 downto 1);
          signal plus3_code            : std_logic_vector (4 downto 1);
          signal ostatne_segmenty      : std_logic_vector (11 downto 0);
begin

DUT: gray_to_plus3
port map(
               gray_code             => gray_code,
               plus3_code            => plus3_code,
               ostatne_segmenty      => ostatne_segmenty
       );

          ostatne_segmenty  <= "111111111111";

          stimulus: process
                       variable nezhoda   : boolean;
                       variable riadok    : line;
                       variable medzera   : character;
                       FILE vector_file   : TEXT open READ_MODE is "good_values.txt";
                       variable plus3_code_good   : std_logic_vector(4 downto 1);
                       variable gray_code_good    : std_logic_vector(4 downto 1);
          begin
            gray_code <= "0000";
            nezhoda := false;
            wait for 10 ns;
           while not endfile(vector_file) loop
           readline(vector_file, riadok);
          read(riadok, gray_code_good);
          read(riadok, medzera);
          read(riadok, plus3_code_good);
          gray_code <= gray_code_good;
          wait for 10 ns;
          if (plus3_code /= plus3_code_good) then
                    nezhoda := true;
          end if;
     end loop;


     assert (not nezhoda)
       report "Doslo k chybe"
       severity error;

     if (not nezhoda) then
        report "TestBench prebehol v poriadku";
     end if;

          end process;
end architecture testbench;
```

Súbor, ktorý obsahuje referenčné (správne) hodnoty má tvar:

0000 1100
0001 1011

0010 1001
0011 1010
0100 0101
0101 0110
0110 1000
0111 0111
1000 1111
1001 1111
1010 1111
1011 1111
1100 0100
1101 0011
1110 1111
1111 1111

Prvý vektor v riadku určuje vstup do testovaného návrhu, teda signál *gray_code*, pričom túto hodnotu uložíme do premennej

gray_code_good: **read(riadok, gray_code_good);**

Druhý vektor v riadku určuje správnu hodnotu, ktorá by sa mala objaviť na výstupe testovaného návrhu za podmienky že na vstup privedieme hodnotu *gray_code_good*. Túto hodnotu uložíme do premennej

plus3_code_good: **read(riadok, plus3_code_good);**

Medzi týmito dvoma operáciami je potrebné ešte vyčítať medzeru, nakoľko sú jednotlivé vektory od seba oddelené medzerou:

read(riadok, medzera);

Priradíme hodnotu *gray_code_good* na vstup testovanej jednotky (nášho návrhu):

gray_code <= gray_code_good;

a vyčkáme určitý čas:

wait for 10 ns;

Výstup z testovanej jednotky podrobíme testu, či sa rovná správnej hodnote uloženej v premennej *plus3_code_good*, a ak sa nerovná, nastavíme premennú *nezhoda* na hodnotu *true*, čo nám určuje, že dosiahnutá hodnota sa nerovná predpokladanej hodnote. Táto premenná je následne rozhodovacím výrazom pre príkaz *assert*, ktorý ak došlo k chybe vypíše zodpovedajúcu správu a ukončí simuláciu. Ak všetky dosiahnuté vektory sa rovnajú predpokladaným, vypíše sa správa, že simulácia prebehla v poriadku.

Nasledovný výpis dávkového súboru realizuje kompiláciu a simuláciu:

quit -sim

vcom -work work -2002 -explicit -novitalcheck -no1164 -novital gray_to_plus3.vhd gray_to_plus3_tb.vhd
vsim work.gray_to_plus3_tb
view wave
do wave.do
restart -f
run 200

8.3 ČASOVÁ SIMULÁCIA V MODELSIME

Ak simulovaný VHDL kód využíva prvky z optimalizovaných LPM knižníc, resp. ak je potrebné vykonať **časovú** simuláciu **konkrétneho** FPGA obvodu, je potrebné v ModelSime najskôr skompilovať príslušné **technologicky závislé knižnice**. Tieto VHDL knižnice dodávajú výrobcovia FPGA obvodov a typicky ich je možné nájsť ako súčasť vývojového prostredia (napr. Quartus II pre Alteru). V ďalšej časti bude opísaná simulácia FPGA obvodov Altera, naznačený postup je však možné po malých modifikáciách využiť aj pre obvody iných výrobcov (Xilinx, Actel, …).

8.3.1 KOMPILÁCIA TECHNOLOGICKY ZÁVISLÝCH KNIŽNÍC

Predpokladajme, že budeme využívať aj určité LPM funkcie. Je preto potrebné skompilovať VHDL kódy LPM knižnice (*220pack.vhd* a *220model.vhd*).

Postup je následovný:

1. spustiť ModelSim a zatvoriť aktuálne otvorený projekt,
2. nastaviť pracovný adresár na adresár, kde sa nachádzajú požadované VHDL súbory (napr. *C:\altera\quartus\eda\sim_lib*). Zmenu pracovného adresára je možné realizovať cez GUI menu (**File->Change directory**), alebo pomocou príkazu **cd** v príkazovom riadku ModelSimu,
3. vytvoriť LMP knižnicu a nastaviť ju ako pracovnú pomocou príkazov
   vlib lpm
   vmap work lpm
4. skompilovať príslušné knižničné VHDL súbory (*220pack.vhd* a *220model.vhd*). Je nevyhnutné dodržať uvedené poradie, pretože kód v súbore *220model.vhd* využíva súbor *220pack.vhd*. Je to možné vykonať opäť, buď pomocou GUI alebo pomocou príkazov. V prípade využitia GUI je najskôr potrebné nastaviť sa na súbor, ktorý sa bude kompilovať a následne vybrať položku *Compile->Compile*. Ekvivalentnú činnosť je možné realizovať pomocou príkazu **vcom** priamo z príkazového riadku, prípadne v rámci dávkových súborov,
5. ako posledný krok je potrebné upraviť súbor *modelsim.ini*, ktorý sa nachádza v hlavnom adresári ModelSimu. Do súboru modelsim.ini je potrebné pridať

cestu na práve skompilovanú knižnicu. V súbore nájdeme blok označený ako ***[Library]*** a pridáme riadok ***lmp=C:/altera/quartus/eda/sim_lib/lpm.***

Keďže časová simulácia bude realizovaná pre konkrétny cieľový (FPGA) obvod, je potrebné postupovať podobným postupom aj pri kompilácii potrebných knižníc pre jednotlivé cieľové rodiny obvodov (napr. *flex10k, apex20k, apex20ke, .., cyclone, ...* obvodov Altera).

Dávkový súbor ktorý realizuje kompiláciu LPM knižnice môže mať následovný tvar:

```
# kompilacia LPM kniznice
cd C:/altera/quartus/eda/sim_lib
vlib lpm
vmap work lpm
#vmap –c work lpm
vcom -reportprogress 300 -work lpm 220pack.vhd
vcom -reportprogress 300 -work lpm 220model.vhd
```

Pre úplnosť je v nasledujúcom odseku výpis dávkového súboru realizujúceho kompiláciu knižníc obvodov radu *flex10ke*

```
# kompilacia kniznic obodov flex10ke
cd C:/altera/quartus/eda/sim_lib
vlib flex10ke
vmap work flex10ke
#vmap –c work flex10ke
vcom -reportprogress 300 -work flex10ke flex10ke_atoms.vhd
    flex10ke_components.vhd
```

Upozornenie: Ak inicializačný súbor modelsim.ini nie je možné editovať v adresári s originálnou inštaláciou ModelSimu, tak pomocou príkazu '**vmap –c ...**' je možné vytvoriť lokálnu kópiu v aktuálnom adresári. Pridanie knižnice do aktuálneho projektu sa realizuje príkazom 'vmap nazov_kniznice cesta_k_skompilovanej_kniznici'. Tento postup aplikujeme ak je knižnica už skompilovaná, ale nebol zmenený inicializačný súbor, pripadne chceme knižnicu používať len v danom projekte a nie po každom spustení ModelSimu.

8.3.2 ČASOVÁ SIMULÁCIA V MODELSIME

Funkčná simulácia sa vyznačuje tým, že ak niektorý zo signálov spôsobí zmenu stavu iného signálu (prechod z 0 do 1 a pod.), táto zmena sa vykoná okamžite, teda v čase zmeny (prechodu) signálu, ktorý túto zmenu spôsobuje. To znamená, že doba medzi podnetom a reakciou je nulová. Vo všeobecnosti funkčná simulácia slúži na overenie **principiálnej funkčnosti** navrhovaného obvodu, pričom neuvažuje skutočné

doby oneskorení v cieľovej súčiastke. Na počiatočné overenie funkčnosti návrhu to zvyčajne postačuje, avšak správny výsledok funkčnej simulácie ani zďaleka nezaručuje, že návrh bude korektne fungovať aj po implementácii v cieľovej (FPGA) súčiastke. Presnejší obraz o správaní sa návrhu v cieľovej súčiastke poskytuje **časová simulácia**, ktorá zohľadňuje aj časy prechodov signálov z jedného logického stavu do druhého, konečnú rýchlosť sírenia sa signálov, limitovaný počet krátkych a rýchlych spojení v cieľovej súčiastke, a pod. Časová simulácia vykonávaná v prostredí Quartus bola vysvetlená na jednom z predchádzajúcich cvičení. Taktiež bolo naznačené, že prostredie ModelSim je všeobecnejšie simulačné prostredie poskytujúce mnoho ďalších vlastností, ktorými napr. prostredie Quartus II nedisponuje. Preto je výhodné mať možnosť realizovať časovú simuláciu aj v ModelSime.

Na to je však potrebné získať VHDL kód návrhu, ktorý už **obsahuje aj konkrétne oneskorenia** špecifické pre cieľový (FPGA) obvod. Výstupné súbory pre časovú simuláciu zohľadňujú tieto oneskorenia a preto je časová simulácia bližšie k reálnemu správaniu sa súčiastky. **Súbor *VHO*** obsahuje opis projektu v jazyku VHDL, avšak oproti pôvodnému *VHDL* súboru, ktorý je použitý na opis návrhu, *VHO* súbor už obsahuje aj informácie o technologických prvkoch cieľovej súčiastky (z VHO súboru je napr. možné vyčítať, pomocou akých blokov sú realizované požadované operácie).

Ďalší potrebný **súbor *SDO*** obsahuje informácie o časových oneskoreniach medzi vstupnými a výstupnými signálmi jednotlivých blokov v rámci projektu. Súbory VHO a SDO je možné získať pomocou prostredia Quartus II nasledovným postupom:

- skopírovať *VHDL* súbor návrhu do zvoleného pracovného projektu,
- vytvoriť nový projekt s rovnakým názvom v zvolenom pracovnom adresári spolu s dodatočnými, doposiaľ nevyužívanými nastaveniami (viď ďalšie body),
- v okne **EDA tools settings** zvoliť **Simulation** a z ponuky zvoliť **ModelSim (VHDL output from Quartus II)**, čo zabezpečí automatické vytvorenie súborov (SDO a VHO) potrebných pre časovú simuláciu v ModelSime,
- zvolíme si cieľovú súčiastku, pre ktorú je návrh realizovaný,
- skompilovať projekt
- súbory pre časovú simuláciu sa nachádzajú v adresári *„./simulation/modelsim“*

Príklad 4

*Vytvorte *.vho a *.sdo súbory zodpovedajúce implementácii jednoduchého čítača (citac_plus1.vhd,) v obvode Altera EPF10K20RC240-4. Zobrazte výsledky časovej simulácie v prostredí Modelsim.*

V prostredí Quartus II vytvoríme nový projekt (**File->New Project Wizard**), nastavíme sa do vytvoreného adresára, zadáme názov **top-level** entity a názov projektu tak, ako to bolo precvičované v predchádzajúcich cvičeniach. Pri špecifikácii cieľovej rodiny a

obvodu zvolíme rodinu *Flex10k* a konkrétne súčiastku[9] *EPF10K20RC240-4*. V ďalšom okne zvolíme položku **EDA Simulation tool**, kde vyberieme položku *ModelSim VHDL* a ukončíme konfiguráciu projektu. Po vytvorení projektu zrealizujeme kompiláciu, počas ktorej sa automaticky vytvoria požadované súbory **.vho* a **.sdo*. Z menu **Processing** vyberieme položku **Start Compilation**, čím spustíme kompiláciu[10]. Po ukončení kompilácie nájdeme v adresári *„./simulation/modelsim"* požadovaná súbory *citac_plus1.vho* a *citac_plus1_vhd.sdo.*

Nasleduje samotná časová simulácia v ModelSime. Opäť vytvoríme nový adresár pre nový projekt, do ktorého skopírujeme generované súbory *citac_plus1.vho* a *citac_plus1_vhd.sdo*. Pomocou **File->New->Project** v ModelSime vytvoríme nový projekt, ktorý nazveme, npar. *citac_plus1* a nastavíme sa do vytvoreného adresára (*Project Location*). Z ďalšieho okna vyberieme **Add existing file** a vyberieme súbor *citac_plus1.vho*[11] a potvrdíme, čím zabezpečíme všetky potrebné nastavenia projektu. Nasleduje samotná kompilácia (v Modelsime) a jeho simulácia. Nasledujúce operácie využívajú príkazy Modelsimu a nie GUI. Samozrejme všetky operácie by bolo možné realizovať aj prostredníctvom GUI.

Kompilácia sa vykoná príkazom **vcom**, pričom kompilujeme namiesto VHDL súboru zodpovedajúci súbor VHO pomocou príkazu:

vcom -work work -2002 -explicit -novitalcheck -no1164 -novital citac_plus1.vho

Simuláciu vykonáme príkazom **vsim**, pričom ako jeden z parametrov špecifikujeme súbor *citac_plus1_vhd.sdo:*

vsim -sdftyp citac_plus1_vhd.sdo work.citac_plus1

Do okna **wave** pridáme požadované signály a zapíšeme ich pre opätovné spúšťanie:

add wave clk vystup
write format wave wave.do

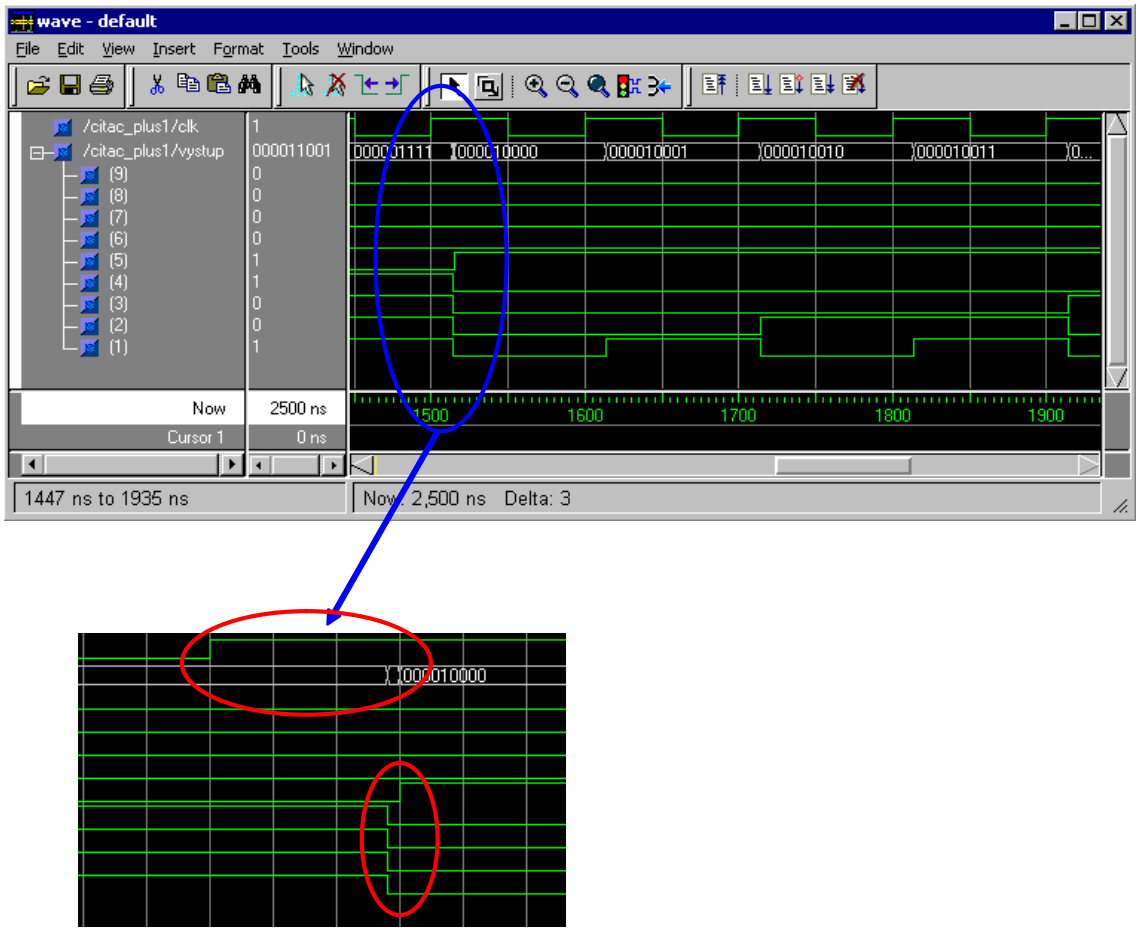
Definujeme vstupné stimuly:

restart -f
force -freeze sim:/citac_plus1/clk 1 0, 0 {50 ns} -r 100
run 2500

Na obr.4 je znázornený výsledok časovej simulácie a tiež detail výsledku simulácie, kde je jasne vidieť že výstupné signály sa nemenia okamžite, keď sa objaví podnet na ich

[9] Obvod je využitý vo vývojovej doske Altera UP1.
[10] Počas kompilácie je v prostredí Quartus II realizovaná syntéza a následne proces P&R. Do vytvorených súborov *.sdo a *.vho je vložená aj informácia o umiestnení logiky a oneskoreniach jednotlivých blokov logiky.
[11] Pôvodný zdrojový VHDL kód už nie je pre simuláciu v ModelSime potrebný.

zmenu, ale až po určitom čase (viď. červeno orámované oblasti). Niekedy je tiež možné pozorovať **zákmity** (tzv. **glitches**) spôsobené rýchlymi zmenami signálov v logike. **Rozhodujúci je však okamih vzorkovania signálu – v prípade synchrónnych návrhov na príslušnú hranu hodinového signálu**.



Obr.4 Výsledky časovej simulácie VHDL kódu čítača+1 v obvode EPF10K20RC240-4

Pre úplnosť je nižšie uvedený VHDL súbor, z ktorého boli generované súbory *citac_plus1.vho* a *citac_plus1_vhd.sdo.*

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity citac_plus1 is
generic (N : integer:= 9);
port (
        clk             : in    std_logic;                          --hodiny
        vystup          : out   std_logic_vector (N downto 1));
end entity citac_plus1;

architecture arch of citac_plus1 is
        signal vystup_sig       : std_logic_vector (N downto 1);    --pomocny signal
begin
        process (clk)
                begin
                        if (clk'event and clk='1') then
                                vystup_sig <= vystup_sig + '1';
                        end if;
        end process;
        vystup <= vystup_sig;
end architecture arch;
```

PRÍLOHA I

Vytvorenie signálu ktorý trvá jednu periódu (1T) hodinového signálu

Príklad 5
Vytvorte VHDL kód obvodu, ktorý predspracuje asynchrónny vstupný signál tak, aby bolo možné spoľahlivo počítať výskyt zmien (z log0 do log1) externého asynchrónneho signálu. Návrh odsimulujte v Modelsime. Aké nevýhody má uvedené riešenie?

Pri práci so signálmi, ktoré sa privádzajú do obvodu z externého prostredia je častým problémom ich asynchrónnosť. Asynchrónne signály je pred vstupom do synchrónnej logiky potrebné synchronizovať, čím sa výrazne zníži pravdepodobnosť chybnej činnosti následnej logiky spôsobenej tzv. metastabilnými stavmi. Synchronizáciu asynchrónneho vstupu je možné realizovať jednoduchým *D* klopným obvodom, ktorý má na vstup *D* privedený signál, ktorý je potrebné synchronizovať a na hodinový vstup *clk* privedený hodinový signál, ktorým ma byť daný signál synchronizovaný. Nasledujúci VHDL kód realizuje uvedenú synchronizáciu.

```
synchronizacia: process (clock)
begin
        if (clock'event and clock='1') then
                vstup_synchronizovany <= vstup;
        end if;
end process;
```

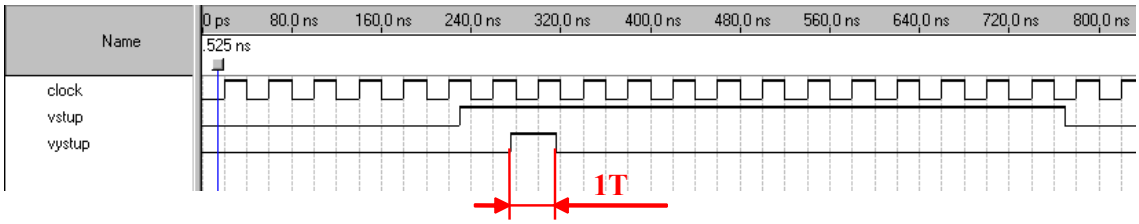
V ďalšej časti je vytvorený impulz široký 1T (jednu periódu hodinového signálu) privedením synchronizovaného signálu do ďalšieho *D* klopného obvodu podobne ako v predchádzajúcej časti.

```
oneskorenie: process (clock)
begin
        if (clock'event and clock='1') then
                vstup_oneskoreny <= vstup_synchronizovany;
        end if;
end process;
```

Následne je pomocou hradla AND a jedným invertorom NOT vytvorený signál široký 1T:

```
vystup <= vstup_synchronizovany and not vstup_oneskoreny;
```

Na obr.5 je zobrazený výsledok simulácie opísanej realizácie obvodu.



Obr.5 Výsledky časovej simulácie

Ak máme vytvorený signál široký 1T zo signálu širokého niekoľko periód, a ak je návrh zosynchronizovaný hodinovým signálom, z ktorého je odvodená perióda 1T, je možné v jednoduchom procese aktívnom na hodinový signál počítať výskyt takýchto impulzov. Ak je signál vytvorený uvedeným spôsobom, tak zaručene práve raz sa počas aktívnej doby tohto signálu (pokiaľ je doba trvania **dostatočne dlhá**!) vyskytne **jediná** nábežná hrana hodinového signálu. Je to jednoduchý princíp, ktorý má viaceré praktické využitia.

Úplný výpis VHDL kódu:

```
library ieee;
use ieee.std_logic_1164.all;

entity signal_1t is
port
	(
	clock	: in	std_logic;
	vstup	: in	std_logic;
	vystup	: out	std_logic
	);
end entity signal_1t;

architecture signal_1t_arch of signal_1t is
	signal vstup_synchronizovany	: std_logic;
	signal vstup_oneskoreny		: std_logic;
begin

	synchronizacia: process (clock)
	begin
		if (clock'event and clock='1') then
			vstup_synchronizovany <= vstup;
		end if;
	end process;

	oneskorenie: process (clock)
	begin
		if (clock'event and clock='1') then
			vstup_oneskoreny <= vstup_synchronizovany;
		end if;
	end process;

vystup <= vstup_synchronizovany and not vstup_oneskoreny;

end architecture;
```