Návrh digitalnych systemov pre FPGA

Martin Šimka
ver.1.0

8. februára 2005

+lščťčéáíéžíýáž

1 Simulacia, synteza a implementacia

Na jednoduchom príklade čítača ukážeme základný postup pri vývoji aplikácií pre FPGA obvody. Cieľom cvičenia je osvojiť si operácie bežne vykonávané počas simulácie a syntézy projektov. V krátkosti je vysvetlený spôsob opisu správania sa čítača pomocou jazyka VHDL. Ďalej je uvedený postup na vykonanie funkčnej a časovej simulácie v programe ModelSim, syntéza návrhu a Place&Route.

Funkčná simulácia sa používa na základné otestovanie funkčnosti navrhnutého projektu. Nezohľadňuje časové oneskorenie, ktoré vzniká v realnych obvodoch. Predstavuje prvy krok pri navrhu lubovolneho digitalneho systemu pre hradlove polia. Na zaklade specifikacie pozadovaneho spravania sa systemu najprv urcime strukturu - rozdelime system na jednotlive funkcne bloky a urcime rozhrania medzi nimi. Definuje sa rozhranie s vonkajsimi systemami. Nasledne je mozne postupit k popisu spravania sa jednotlivych blokov pomocou jazyka VHDL.

Nasim cielom je implementovat jednoduchy parametrizovalny citac vpred a vzad s menitelnou velkostou kroku a poctom krokov, ktory po spusteni vykona dany pocet krokov vpred (inkrementacia vystupneho signalu) a nasledne vzad (dekrementacia). Vstupnymi signalmi teda budu:

- hodinovy signal,
- signal pre inicializaciu obvodu (reset),
- signal pre spustenie citaca.

Ako vystup z citaca pozadujeme signal s in- a dekrementujucim sa obsahom. Jedna sa o jednoduchu operaciu, preto bude realizovana ako jeden blok. Na riadenie jednotlivych stavov pouzijeme konecny automat so 4 stavmi:

- zakladny stav po inicializacii alebo po skonceni operacie,
- stav inkrementacie,
- stav dekrementacie,
- stav skoncenia operacie.

Na sledovanie stavu citaca pouzijeme 3 priznakove signaly:

- priznak inkrementacie,
- priznak dosiahnutia minimalnej hodnoty,
- priznak dosiahnutia maximalnej hodnoty.

Sposob popisu takto definovaneho spravania citaca je mozne sledovat v zdrojovom subore VHDL *example.vhd*.

Časová simulácia návrhu zohľadňuje časové onskorenia vo vnútri digitálneho obvodu. Ako vstup nie sú použité pôvodné VHDL súbory, ale súbory VHO a SDO, ktoré boli vygenerované pri syntéze a P&R, a ktoré obsahujú časové údaje pre náš návrh a zvolenú súčiatku.

1.1 VHDL

Na popis návrhu je použitý jazyk VHDL. Ide o široko používaný programovací jazyk na popis hardvéru. Nie je cieľom naučiť sa syntax a použitie všetkých príkazov jazyka, preto budú podrobnejšie vysvetlené len základné bloky a vlastnosti jazyka.

Prvú časť kódu VHDL tvoria definície knižníc. Medzi základné a najčastejšie používané knižnice patria knižnice:

```
ieee.std_logic_1164.ALL;
ieee.std_logic_arith.ALL;
ieee.std_logic_unsigned.ALL;
```

Nasleduje deklaracia *entity* – pre lepšiu predstavu možno použiť analógiu z klasických "fyzických" súčiastok, keď entitou nazveme takúto súčiastku a v jej deklaracii uvádzame aké má vstupné a výstupné *porty* – vývody, prípadne možno naviac uviesť parametre ako v našom prípade veľkosť kroku čítača pomocou parametra $STEP$, pocet inkrementacii ($INCR$) a sirka vystupneho signalu ((D_WIDTH)).

V deklaracii portov máme napr.:

```
clk        : IN   STD_LOGIC;
```

Kde *clk* je názov portu, *IN* udáva, že ide o vstupný port a *STD_LOGIC* určuje typ portu.

V architektúre entity je popísané správanie popísaného obvodu. V prvej časti sú deklarované signály a typy signálov. V druhej časti pomocou tzv. procesov môžeme popísať činnosť obvodu. Treba si uvedomiť, že jednotlivé procesy sa realizujú paralelne, teda nezáleží na poradí procesov v akom su zapisane v subore.

Uvedený čítač sa po zrealizovaní resetu (*reset* signál aktívny počas jednej periódy hodín), zinicializuje a očakáva aktívny signál *start*. Následne inkrementuje a dekrementuje výstupný signál, po čom sa opäť dostáva do čakacieho stavu, keď môže prísť ďalší signál *start*. Jednotlivé stavy čítača sú riadené podľa konečného automatu. Jednoduchou zmenou hodnot parametrov čítača v deklaracii entity, dosiahneme zmenu správania návrhu.

1.2 Funkčná simulácia

Do pracovného adresára skopírujeme pripravené súbory: *do.do* a *example.vhd*. Spustíme program ModelSim a vytvoríme nový projekt s názvom *example* cez voľbu *File* → *New* → *Project*. V otvorenom okne zapíšeme do *Project name:* example, pre *Project location:* nastavíme pracovný adresár, ktorý sme vytvorili pre tento projekt. V ďalšom okne pridáme existujúci súbor *example.vhd* (*Add existing file* → *Browse*), potvrdíme výber a zavrieme malé okno. Pred samotnou simuláciou je nutné po každej zmene zdrojových súborov celý projekt skompilovať. V našom prípade je použitý len jeden zdrojový súbor, a preto nezáleží na poradí kompilácie.

Na zrýchlenie práce bol vytvorený dávkový súbor *sim.do*, ktorý po zadaní príkazu

```
do sim.do
```

do príkazového riadku vykoná automaticky operácie, ktoré sú opísané v nasledujúcom odseku. V prípade problémov je dobré prečítať si, čo sa v dávkovom súbore vykonáva a na základe toho odhaliť zdroj chybových hlášok, resp. miesto, kde vykonať požadované zmeny.

Pri ručnom postupe označíme súbor *example.vhd* a v lište s ikonami klineme na *Compile* resp. *Compile All*. Ak je kompilácia v poriadku vypíše sa hláška

```
# Compile of example.vhd was successful.
```

a môžeme pokračovať simuláciou klikom na ikonu *Simulate*. V otvorenom okne rozbalíme strom v položke *work* a zvolíme *example*. Na otvorenie všetkých okien simulátora môžem zadať

```
view *
```

Nás budú najviac zaujímať okná *Wave* a *Signals*. Preto stačí ak sa zobrazia len tieto dve okná, čo docielime pomocou príkazu

```
view signals wave
```

V okne *Signals* pridáme signály na simuláciu (*Add* → *Wave* → *Signals in region*). Následne je možné v okne *Wave* upravovať formát zobrazovania jednotlivých signálov. Výsledné nastavenie sa uloží cez *File* → *Save* → *Format*, do súboru *wave.do*, ktorý možno použiť pri opätovnej simulácii pomocou zadania príkazu

```
do wave.do
```

Pred spustením simulácie je potrebné nastaviť hodnoty vstupných signálov v okne *Signals* cez *Edit* (alebo pravý klik) → *Force* resp. *Clock*, v prípade hodinového signálu. Všimnime si, že všetky operácie realizované pomocou menu sú vykonávané v príkazovom riadku pomocou príkazov programu ModelSim. Tieto môžeme uložiť do tzv. dávkovacieho súboru s koncovkou .do a následne spúšťať pomocou príkazu `do`. Vyskúšať si to môžeme pri spustení pripraveného dávkového súboru *do.do*, ktorý vykoná jednoduché spustenie čítača. Zadáme teda:

```
do do.do
```

a vo *Wave* okne skontrolujeme grafický výstup simulácie.

Ak chceme zmeniť parametre návrhu, zeditujeme súbor *example.vhd*, uložíme ho a použijeme opäť príkaz

```
do sim.do
```

V prípade, že v grafickom okne nie je zobrazený koniec simulácie, môžeme v simulácii pokračovať zadávaním príkazu

```
run 1000
```

kde 1000 predstavuje 1000 základných jednotiek, zvyčajne nanosekúnd.

1.2.1 Testbench

Na zautomatizovanie priebehu simulacie je casto pouzivany tzv. testbench. Jedna sa o subor v jazyku VHDL, ktory zahrna navrhovany projekt, produkuje stimuly pre jeho vstupne signaly a kontroluje vystupne signaly. V oblasti klasickej elektroniky mozeme testbench prirovnat k testovaciemu zapojeniu, kde na vstup testovaneho obvodu pripojime generator a vystup sledujeme na osciloskope a pod.

Testbench je zvlast prakticky, ak vstupne data a signaly prichadzaju v urcitej postupnosti, ktora sa navyse urcitych intervaloch opakuje v zavislosti na vystupnych signaloch (napr. pri roznych typoch rozhrani). V takom pripade je riadenie vstupnych signalov pomocou funkcie *Force* nedostatocne. Dalsou vyhodnou vlastnostou testbenchov je moznost citat a zapisovat data z/do suborov, co umoznuje rychlu zmenu vstupnych dat pri testovani.

Pri pouziti testbencha postupujeme podobne ako uz bolo uvedene. Do pracovneho adresara ulozime VHDL subory *example.vhd, example_tb.vhd* a davkove subory *sim_tb.do, wave.do*. Kompilaciu a spustenie simulacie mozeme vykonat rucne alebo pouzitim davkoveho suboru *sim_tb.do*. Testbench realizuje kontrolu vystupneho signalu *data* po vykonani resetu a po kazdom kroku inkrementacie resp. dekrementacie.

1.3 Syntéza

Syntéza obvodu je potrebná pre implementáciu do cieľovej súčiastky. Na zaklade simulovaneho popisu projektu v jazyku VHDL sa vytvori struktura pozostavajuca zo zakladnych blokov cielovej suciatky, teda registrov, multiplexorov, hradiel a pod. Použijeme 3 rôzne programy: vseobecne často používaný program LeonardoSpectrum určený len na syntézu, podobny program Precision RTL Synthesis (oba z balika programov FPGA Advantage od spolocnosti Mentor Graphics) a program Quartus od spoločnosti Altera, ktorý v sebe zahŕňa prostriedky na syntézu, simuláciu aj Place&Route.

1.3.1 LeonardoSpectrum

LeonardoSpectrum ponúka po spustení 3 rôzne úrovne. Tie je možné voliť podľa toho, aký skúsený je užívateľ a aké cieľové obvody chceme použiť. Zvolíme level 1 a z menu vyberieme obvody od firmy Altera. Po otvorení základného okna si v hornej lište ikon zvolíme *Quick Setup*. Ide o najjednoduchší spôsob ako vykonať syntézu a plne postačuje pre naše účely. V časti *Technology* zvolíme typ obvodu, ktorý do ktorého chceme návrh implementovať. V časti *Input* zvolíme pracovný adresár a otvoríme zdrojový súbor *example.vhd*.

Pre spustenie procedúry *Place and Route* je potrebné zaškrtnúť *Run Integrated Place and Route*. Po skončení syntézy a P&R získame informácie o počte vstupných a výstupných vývodov, počte obsadených Logických elementov (LE) a o maximánej možnej taktovacej frekvencii hodinového signálu.

1.3.2 Precision Synthesis

Precision Synthesis je podobne ako LeonardoSpectrum urceny na syntezu navrhov pre FPGA. Pri vytvarani noveho projektu je mozne postupovat intuitivnym sposobom. V lavej casti sa zobrazuju len ikony tych operacii, ktore uz je mozne vykonat. Teda po vytvoreni pracovneho adresara, kde umiestnime zdrojovy subor *example.vhd*, klikneme na ikonu *New Project*, zvolime nazov projektu a nastavime cestu na pracovny adresar. Do projektu pridame pomocou *Add Input Files* subor *example.vhd* a v *Setup Design* zvolime typ cielovej suciastky a pozadovanu frekvenciu navrhu. Dalej mozeme projekt skompilovat a vykonat syntezu pomocou kliknutia na prslusne ikony - *Compile* resp. *Synthesize*. Po kompilacii je mozne si v zalozke *Design Analysis* prezriet schemu navrhu kliknutim na *View RTL Schematic*. Po synteze je mozne si pozriet podobnu schemu cez *View Schematic*, kde vidiet realizaciu navrhu pomocou zdrojov, ktore ponuka zvolena cielova suciastka. Pre spustenie operacie *Place and Route* zvolime *Run Quartus II* v zalozke *Quartus II*. Pod zalozkou *Design Center* najdeme vystupne subory vyprodukovane po place&route programom Quartus. Súbory pre časovú simuláciu sú umiestnené v adresári $\star$*_impl_1/simulation/modelsim*.

1.3.3 Quartus

Quartus predstavuje program, ktorý umožňuje kompletný vývoj projektov pre obvody Altera. Cieľom je získať informácie o veľkosti a rýchlosti navrhovaného projektu a zároveň súbory potrebné na časovú simuláciu návrhu. Do zvoleného pracovného adresára skopírujeme súbor *example.vhd*. Spustíme program Quartus. Vytvoríme nový projekt s rovnakým názvom v zvolenom adresári (*File* $\rightarrow$ *New Project Wizard*). Klikneme dvakrát na *Next*, v okne *EDA tools settings*, zvolíme *Simulation* a z ponuky vyberieme *ModelSim (VHDL output from Quartus II)*, čo zabezpečí vytvorenie súborov potrebných na časovú simuláciu v ModelSime. Opäť klikneme na *Next* a v ďalšom okne si zvolíme rodinu obvodov od Altery, do ktorej chceme návrh umiestniť. Klikneme na *Finish*. Tým je skončená fáza vytvárania projektu a môžeme postúpiť ku kompilácii, ktorá v sebe zahŕňa analýzu a syntézu návrhu, umiestnenie do zvolenej súčiastky a časovú analýzu. Proces spustíme kliknutím na príslušnú ikonu (bordová šípka) alebo z menu (*Processing* $\rightarrow$ *Start*

compilation). Výsledku spracovania si možno prezrieť v reporte. Súbory pre časovú simuláciu sú umiestnené v adresári *simulation/modelsim*.

1.4 Časová simulácia

Súbory *example.vho* a *example_vhd.sdo* skopírujeme do adresára pre nový projekt. Ďalej pokračujeme podobne ako pri funkčnej simulácii. Spustíme ModelSim a vytvoríme nový projekt, pridáme doňho súbor *example.vho* a skompilujeme.

Na zrýchlenie práce bol opäť vytvorený dávkový súbor s názvom *tsim.do*, ktorý po zadaní príkazu

```
do tsim.do
```

vykoná časovú simuláciu projektu. Oproti funkčnej simulácii je odlišné spustenie simulácie, keď v okne *Simulate* navyše určíme aj súbor obsahujúci informácie o oneskoreniach obvodu. V záložke SDF teda zvolíme *Add* a vyberieme súbor *example_vhd.sdo* a potvrdíme. Ďalej postupujeme rovnako ako pri funkčnej simulácii v časti 1.2.

Je potrebné si všimnúť, že návrh už neobsahuje niektoré signály, ktoré boli použité pri opise návrhu v pôvodnom VHDL súbore. Tieto boli po optimalizácii z projektu odstránené alebo premenované, na druhej strane pribudli ďalšie signály potrebné pre fyzickú realizáciu návrhu. Zaujímavé sú aj zmeny hodnôt signálu *data*, ktoré neprebiehajú ideálne ako to bolo vo funkčnej simulácii. Dolezite je sledovat kedy nastanu zmeny daneho signalu v zavislosti od hodinoveho signalu. Signaly menia hodnotu s urcitym oneskorenim po nabeznej hrane hodinoveho signalu a signal *data* dokonca nemeni hodnotu vsetkych bitov sucasne, ale opat s urcitym oneskorenim.

2 Práca s MegaCore funkciou – FIR Compiler

V tejto časti je uvedený postup pri práci s MegaCore funkciou FIR Compiler ver.3.1.0 od spoločnosti Altera, ktorá umožňuje efektívny návrh a implementáciu FIR filtrov. Funkcia poskytuje priamu podporu pre nové obvody Stratix a Cyclone, umožňuje vytvárať širokú škálu architektúr filtrov:

- s pevne zadanými koeficientami – plne paralelné, sériové, viac-bitové sériové, podpora interpolácie a decimácie,
- variabilné filtre - viaccyklové, podpora viacerých sád koeficientov.

Nasleduje podrobný postup pre návrh a skompilovanie FIR filtra. Otvoríme program Quartus II a vytvoríme nový projekt - je potrebné špecifikovať pracovný adresár a názov projektu. V ďalšom okne klikneme na *User Library Pathnames* a následne na *Add* a nastavíme cestu: C:/altera/MegaCore/fir_compiler-v3.1.0/lib. Vytváranie ukončíme klikom na *Finish*.

Po vytvorení projektu môžeme spustiť samotnú MegaCore funkciu. V menu *Tools* zvolíme *MegaWizard Plug-In Manager* a vyberieme prvú položku, teda vytvorenie novej megafunkcie. Otvorí sa okno s ponukou megafunkcií a takisto s nami požadovaným FIR kompilátorom. V ľavej časti okna (v menu *DSP* → *Filters*) klikneme na FIR Filter. Zvolíme VHDL (v pravej časti okna) a zvolíme názov filtra totožný s názvom projektu (toto musí byť bezpodmienečne dodržané!!!). Následne sa spustí FIR kompilátor.

K dispozícii je 6 položiek menu:

- About this core – poskytuje základné informácie o funkcii,
- Documentation – zoznam literatúry k funkcii,
- Display symbol – zobrazí symbol vytváranej funkcie,
- Parameterize – umožňuje samotný návrh filtra,
- Set up simulation – vytvára rôzne typy simulačných modelov napr. pre ModelSim, Quartus, Matlab.
- Generate – vytvorí filter s požadovanými parametrami spolu so simulačným modelom.

V programe Quartus klikneme na Start compilation (bordová šípka v hornom rade ikon). Po úspešnej kompilácii môžeme skontrolovať výsledky implementácie, čiže počet obsadených logických a pamäťových elementov a maximálnu frekvenciu hodinového signálu.

Pre spustenie simulácie v Quartuse stačí zvoliť príslušný simulačný model a spustiť simuláciu. Po skončení simulácie je možné si prezrieť získané výstupy.

Pre získanie porovnania medzi rôznymi implementáciami filtrov môžeme celý postup opakovať a meniť typ architektúry filtra, využiť DSP bloky obvodov a podobne.

Prílohy

Literatúra

[1] Weijun Zhang: VHDL Tutorial: Learn by Example, http://www.cs.ucr.edu/content/esd/labs/tutorial/

[2] Hamburg VHDL archive, http://tech-www.informatik.uni-hamburg.de/vhdl/

[3] VHDL verification course, http://www.stefanvhdl.com/

[4] VHDL language guide, http://www.acc-eda.com/vhdlref/

[5] Mujtaba Hamid: Writing Efficient Testbenches, http://direct.xilinx.com/bvdocs/appnotes/xapp199.pdf

[6] Michael John Sebastian Smith: Application-Specific Integrated Circuits, http://www.edacafe.com/books/ASIC/ASICs.php#anchor749424

Čo urobiť pred prvou simuláciou

V ďalšom sa bude predpokladať, že ModelSim je riadne nainštalovaný s platnou licenciou. Na to, aby bolo možné v projektoch používať niektoré optimalizované prvky z LPM knižnice alebo funkcie viazané na daný typ FPGA obvodov, je potrebné skompilovať príslušné knižnice podľa nasledujúceho postupu (príklad je pre knižnice LPM):

- Spustiť ModelSim a zatvoriť práve otvorené projekty,
- Nastaviť pracovný adresár na adresár so zdrojovými VHDL súbormi, obyčajne *C:/altera/quartus/eda/sim_lib (File → Change directory)*,
- Vytvoriť lpm knižnicu a nastaviť ju ako pracovnú pomocou príkazov

```
vlib lpm
```

a

```
vmap work lpm
```

- Skompilovať príslušné VHDL súbory (*220pack.vhd* a *220pack.vhd*, na poradí záleží), buď kliknutím na príslušnú ikonu (Compile), nájdením položky v hornej lište (*Compile* → *Compile*) alebo použitím príkazu `vcom`,

- Upraviť súbor *modelsim.ini*, ktorý sa nachádza v hlavnom adresári ModelSimu pridaním cesty na skompilovanú knižnicu. Do sekcie *[Library]* pridať *lpm = C:/altera/quartus/eda/sim_lib/lpm*.

Podobne treba postupovať pri skompilovaní všetkých potrebných knižníc (altera_mf, apex20k, apex20ke a iné, podľa používaných súčiastok). Na záver je potrebné namapovať opäť pracovný adresár work pomocou:

```
vmap work work
```

DO súbory na automatické skompilovanie knižníc lpm, apex20k a apex20ke sú priložené k dokumentu. Pre ich použitie je potrebné skontrolovať a vhodne upraviť cesty k príslušným súborom. Ručne je potrené upraviť inicializačný súbor *modelsim.ini*.

Ďalšou úpravou sa zabezpečí bezchybné skompilovanie testbenchov, ktoré obsahujú neštandardné VHDL príkazy, ktoré môžu byť pri nesprávnom nastavení kompilátore prezentované ako chyba. Preto je potrebné v *Compiler Options* (*Compile* → *Compile Options*) zrušiť položku *Check for VITAL Compliance*, ďalej *Optimize for StdLogic1164* a *Optimize for VITAL*. Tieto nastavenia sa uložia do hlavného inicializačného súboru *modelsim.ini* a budú použité pri novovytváraných projektoch.

Pre zamedzenie výpisu upozornení, ktoré nemajú vážnejší vplyv na priebeh simulácie (napr. nedefinovaný vstup a pod.) je dobré v *Simulation Options* (*Simulate* → *Simulation Options*) zaškrtnúť položku *Suppress Warnings: From Synopsys Packages* a *Suppress Warnings: From IEEE Numeric Std Packages*, čo môžeme urobiť pred ako aj počas samotnej simulácie.

lpm.do

```
cd C:/altera/quartus/eda/sim_lib
vlib lpm
vmap work lpm
vcom -reportprogress 300 -work work C:/altera/quartus/eda/sim_lib/220pack.vhd
vcom -reportprogress 300 -work work C:/altera/quartus/eda/sim_lib/220model.vhd
```

apex20ke.do

```
cd C:/altera/quartus/eda/sim_lib
vlib apex20k
vmap work apex20k
vcom -reportprogress 300 -work work C:/altera/quartus/eda/sim_lib/apex20k_atoms.vhd
vcom -reportprogress 300 -work work C:/altera/quartus/eda/sim_lib/apex20k_components.vhd
cd C:/altera/quartus/eda/sim_lib
vlib apex20ke
vmap work apex20ke
vcom -reportprogress 300 -work work C:/altera/quartus/eda/sim_lib/apex20ke_atoms.vhd
vcom -reportprogress 300 -work work C:/altera/quartus/eda/sim_lib/apex20ke_components.vhd
```

example.vhd

```
-- example.vhd
-- VHDL subor pre demostracne cvicenie s programom ModelSim
-- autor: Martin Simka
-- upravy:
-- 30-07-2004:
--    pridane komentare, upravene podmienky pre full a empty,
--    pridany parameter INCR
-- 04-02-2005:
--    pridany parameter D_WIDTH, plus dalsie mensie zmeny a pridane komentare

LIBRARY ieee; USE ieee.std_logic_1164.ALL; USE
ieee.std_logic_arith.ALL; USE ieee.std_logic_unsigned.ALL;

--- [ Deklaracia entity] ---

ENTITY example IS
  GENERIC(
    D_WIDTH : integer :=   8;           -- pocet bitov vystupu
    STEP    : integer :=   10;           -- velkost kroku citaca
    INCR    : integer :=   2            -- pocet inkrementacii citaca
                                        -- minimalna hodnota je 2 !!!
  );
  PORT(
  clk       : IN   STD_LOGIC;           -- hodinovy signal
  start     : IN   STD_LOGIC;           -- startovaci signal
  reset     : IN   STD_LOGIC;           -- reset signal
  data      : OUT  STD_LOGIC_VECTOR(D_WIDTH-1 DOWNTO 0)
                                        -- vystupny datovy signal (velkost je ohranicena na 8 %%@
bitov)
  );
END example;

--- [Deklaracia architektury] ---

ARCHITECTURE cviko OF example IS

--- [Deklaracia signalov] ---

-- Konecny automat so 4 stavmi
  TYPE SM_MAIN_TYPE IS (
    main_wait,
    main_increase,
    main_decrease,
    main_finished
  );
  SIGNAL sm_main   : SM_MAIN_TYPE;      -- signal obsahujuci aktualny stav automatu
  SIGNAL increase  : STD_LOGIC;         -- priznakovy signal pre inkrementaciu
  SIGNAL full      : STD_LOGIC;         -- priznakovy signal pre dosiahnutie hornej hranice citaca
  SIGNAL empty     : STD_LOGIC;         -- priznakovy signal pre dosiahnutie dolnej hranice citaca
  SIGNAL sucet     : STD_LOGIC_VECTOR(D_WIDTH-1 DOWNTO 0);
                                        -- signal s aktualnou hodnotou suctu

BEGIN

--- [Definicia procesov] ---

-- Popis konecneho automatu PROCESS(clk, reset) BEGIN
  IF reset = '1' THEN
    sm_main <= main_wait;       -- ak je aktivny 'reset' signal, automat sa nastavi do %%@
referencneho
                                -- stavu 'main_wait'
  ELSIF clk'event AND clk = '1' THEN
    CASE sm_main IS             -- inak sa na nabeznu hranu hodin kontroluje, v ktorom stave sa
                                -- automat nachadza a pri splneni podmienok sa stav zmeni
      WHEN main_wait =>
        IF start = '1' THEN     -- ak je automat v zakladnom stave 'main_wait' pri aktivnom
                                -- signale 'start' sa stav zmeni na 'main_increase'
          sm_main <= main_increase;
          increase <= '1';      -- a zaroven sa signal 'increase' (priznak inkrementacie)
                                -- nastavi na hodnotu '1'
        END IF;
      WHEN main_increase =>
        IF (full = '1') THEN    -- ak je automat v stave 'main_increase' a signal 'full' ma
                                -- hodnotu '1' (bola dosiahnuta horna hranica citaca),
                                -- automat zmeni stav na 'main_decrease'
         sm_main <= main_decrease;
         increase <= '0';       -- a signal 'increase' sa nastavi na hodnotu '0'
        END IF;
      WHEN main_decrease =>
        IF (empty = '1') THEN   -- ak je automat v stave 'main_decrease' a signal 'empty' ma
                                -- hodnotu '1' (bola dosiahnuta dolna hranica citaca),
                                -- automat zmeni stav na 'main_finished'
```

```
        sm_main <= main_finished;
      END IF;
    WHEN main_finished =>
      sm_main <= main_wait;   -- ak je automat v stave 'main_finished' zmeni stav na
                              -- 'main_wait' a cely proces sa moze zopakovat
  END CASE;
 END IF;
END PROCESS;

data <= sucet;                -- interny signal 'sucet' je
priradeny vystupnemu signalu 'data'

-- popis priradenia hodnot signalu 'sucet' PROCESS(clk) BEGIN
 IF clk'event AND clk = '1' THEN
   IF sm_main = main_wait THEN
     sucet <= (OTHERS => '0'); -- pri stave automatu 'main_wait' je hodnota vsetkych bitov
                              -- signalu '0'
   ELSIF sm_main = main_increase THEN
     sucet <= sucet + STEP;   -- v stave 'main_increase' sa v kazdom hodinovom takte hodnota
                              -- signalu 'sucet' zvysi o hodnotu kroku STEP
   ELSIF sm_main = main_decrease THEN
     sucet <= sucet - STEP;   -- v stave 'main_decrease' sa v kazdom hodinovom takte hodnota
                              -- signalu 'sucet' zvysi o hodnotu kroku STEP
   ELSE
     sucet <= sucet;          -- v kazdom inom stave sa zachova hodnota signalu, tento prikaz
                              -- zabezpeci vytvorenie registra na ulozenie hodnoty 'sucet'
   END IF;
 END IF;
END PROCESS;

-- popis priradenia hodnot signalu 'empty' PROCESS(clk) BEGIN
 IF clk'event AND clk = '1' THEN
   IF ((sucet = 2*STEP) AND increase = '0') THEN
                              -- ak hodnota suctu je rovna dvojnasobku parametra STEP poces
                              -- dekrementacie, potom signal
     empty <= '1';            -- 'empty' ma hodnotu '1'
   ELSE
     empty <= '0';            -- inak hodnotu '0'
   END IF;
 END IF;
END PROCESS;

-- popis priradenia hodnot signalu 'full' PROCESS(clk) BEGIN
 IF clk'event AND clk = '1' THEN
   IF (sucet = ((INCR*STEP) - (2*STEP)) AND increase = '1') THEN
                              -- ak je hodnota suctu o dvojnasobok STEP mensia ako je maximalna
                              -- pozadovana hodnota citaca (INCR*STEP) pocas inkrementacie
     full  <= '1';            -- 'full' ma hodnotu '1'
   ELSE
     full  <= '0';            -- inak hodnotu '0'
   END IF;
 END IF;
END PROCESS;

END cviko;
```

example_tb.vhd

```
-- example_tb.vhd
-- VHDL subor pre demostracne cvicenie s programom ModelSim
-- autor: Martin Simka

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 testbench IS
 GENERIC(
   D_WIDTH : integer := 8;            -- pocet bitov vystupu
   STEP    : integer := 5;           -- velkost kroku citaca
   INCR    : integer := 4;            -- pocet inkrementacii citaca (min. 2!!!)
   ClockPeriod    : TIME    := 10 ns
 );
END ENTITY testbench;

ARCHITECTURE test_example OF testbench IS

COMPONENT example IS
 GENERIC(
```

```
    D_WIDTH : integer;                  -- pocet bitov vystupu
    STEP    : integer;                  -- velkost kroku citaca
    INCR    : integer                   -- pocet inkrementacii citaca
                                        -- minimalna hodnota je 2 !!!
  );
  PORT(
  clk       : IN   STD_LOGIC;           -- hodinovy signal
  start     : IN   STD_LOGIC;           -- startovaci signal
  reset     : IN   STD_LOGIC;           -- reset signal
  data      : OUT  STD_LOGIC_VECTOR(D_WIDTH-1 DOWNTO 0)
                                        -- vystupny datovy signal (velkost je ohranicena na 8 %%@
bitov)
  );
END COMPONENT;

SIGNAL clk, start, reset : STD_LOGIC;
SIGNAL data              : STD_LOGIC_VECTOR(D_WIDTH-1 DOWNTO 0);

-- Clock procedure
PROCEDURE wait_clock(CONSTANT clk_ticks:integer) IS
  variable i : integer := 0;
BEGIN
  FOR i IN 1 TO clk_ticks*2 LOOP
    WAIT UNTIL clk'EVENT;
  END LOOP;
END wait_clock;

BEGIN

-- Unit Under Test
UUT: example
GENERIC MAP (
  D_WIDTH   => D_WIDTH,
  STEP      => STEP,
  INCR      => INCR
  )
PORT MAP (
  clk       => clk,
  start     => start,
  reset     => reset,
  data      => data
  );

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

stimulus: PROCESS
-- Initialization process

VARIABLE error_det       : BOOLEAN;
VARIABLE error           : BIT;
VARIABLE L               : LINE;

PROCEDURE init IS
BEGIN
    reset    <= '1';
    start    <= '0';
    wait_clock(1);
    reset    <= '0';
END init;

BEGIN

  init;
  WAIT UNTIL clk = '1';
  IF (data /= 0) THEN
      ASSERT error_det REPORT "mismatch after reset";
      error := '1';
  END IF;
  start <= '1';
  wait_clock(1);
  start <= '0';
  wait_clock(1);

  FOR i IN 0 TO INCR LOOP
    WRITE(L, string'("signal 'data' has value "));
```

```
    WRITE(L, data);
    WRITELINE(OUTPUT,L);
    IF (data /= i*STEP) THEN
      ASSERT error_det REPORT "mismatch during incrementation";
      error := '1';
    ELSE
      REPORT "OK";
      error := '0';
    END IF;
    wait_clock(1);
  END LOOP;

  FOR i IN 1 TO INCR LOOP
    WRITE(L, string'("signal 'data' has value "));
    WRITE(L, data);
    WRITELINE(OUTPUT,L);
    IF (data /= (INCR-i)*STEP) THEN
      ASSERT error_det REPORT "mismatch during decrementation";
      error := '1';
    ELSE
      REPORT "OK";
      error := '0';
    END IF;
    wait_clock(1);
  END LOOP;

  IF (error = '1') THEN -- message about results of comparison
    REPORT "ERROR OCCURED";
  ELSE
    REPORT "NO ERROR OCCURED";
  END IF;

  WAIT;

END PROCESS stimulus;
END ARCHITECTURE test_example;
```

do.do

```
restart -f
force -freeze sim:/example/clk 1 0, 0 {50 ns} -r 100
force -freeze sim:/example/reset 1 0, 0 100
force -freeze sim:/example/start 0 0, 1 100, 0 200
run 2500
```

wave.do

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate -color Cyan -format Logic /example/clk
add wave -noupdate -format Logic /example/start
add wave -noupdate -color {Violet Red} -format Logic /example/reset
add wave -noupdate -format Literal -radix unsigned /example/data
add wave -noupdate -color Gold -format Literal /example/sm_main
add wave -noupdate -format Logic /example/increase
add wave -noupdate -format Logic /example/full
add wave -noupdate -format Logic /example/empty
add wave -noupdate -color Blue -format Literal -radix unsigned /example/sucet
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {0 ns} 0}
configure wave -namecolwidth 150
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 0
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 1
configure wave -griddelta 40
configure wave -timeline 0
update
WaveRestoreZoom {0 ns} {2625 ns}
```

sim.do

```
quit -sim
```

```
vcom_capture -work work -2002 -explicit -novitalcheck -no1164 -novital example.vhd
vsim work.example
view signals wave
do wave.do
do do.do
```

tsim.do

```
quit -sim
vcom_capture -work work -2002 -explicit -novitalcheck -no1164 -novital example.vho
vsim -sdftyp example_vhd.sdo work.example
view signals wave
do wave.do
do do.do
```

sim_tb.do

```
quit -sim
vcom_capture -work work -2002 -explicit -novitalcheck -no1164 -novital example.vhd example_tb.vhd
vsim work.testbench
view signals wave
do wave.do
restart -f
run 200
```