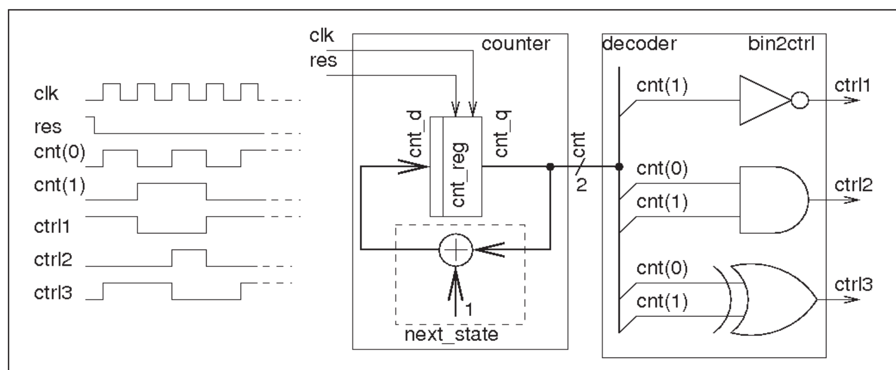
Implementace čítačů v číslicových systémech

Ing. Jakub Šťastný Ph.D.
ASICentrum, s. r. o.
FPGA Laboratoř, FEL ČVUT

Úvod

Čítač je fundamentálním obvodovým blokem nezbytným pro návrh většiny číslicových systémů. Blok čítače je v číslicových obvodech používán v řadě aplikací; nejtypičtějším použitím je čítač užitý jako časovač – blok, který počítá události (například náběžné hrany hodin) na svém vstupu. S čítačem se můžeme setkat ovšem i v jiných podobách, například jako s řadičem pro generování určité sekvence řídicích signálů, s generátorem adresy pro paměť, či čítačem instrukcí programu v procesoru (*Program Counter*). Blok čítače můžeme chápat i jako zjednodušený stavový automat, je-li doplněn o převodník stavu čítače na požadované výstupní signály automatu. Vlastní převodník přitom může degenerovat i na jednoduché „dráty" mezi výstupem čítače a řídicími signály, je-li stav čítače vhodně kódován. Příklad užití čítače jako generátoru sekvence signálů lze nalézt na *obrázku 1*.



Obr. 1 Čítač užitý jako stavový automat pro generování sekvence signálů. V levé části obrázku vidíme příklady časových průběhů hodnot logických signálů na vstupu čítače a na jeho výstupu (hodinový signál clk, resetovací vstup res a výstupy čítače cnt(0) a cnt(1)) spolu s požadovanými průběhy generovaných signálů – ctrl1, 2 a 3. V pravé části je pak rozkresleno schéma čítače spolu s dekodérem pro generování příslušných signálů.

Základní podoba binárního čítače je velmi dobře známá a probíraná ve všech kurzech číslicového návrhu a zmiňovaná i každou knihou zaměřenou na návrh číslicových obvodů, viz například [1], kapitola 2. Už méně známé a zmiňované jsou alternativní možnosti implementace čítače spolu s jeho vlastnostmi; to je překvapivé vzhledem k fundamentálnímu významu bloku čítače pro číslicové systémy.

Potřebujeme-li navrhnout čítač pro konkrétní aplikaci, základní vlastnost, kterou je třeba vzít v úvahu, je kódování stavu čítače. Kódování stavu ovlivňuje téměř všechny parametry výsledného návrhu (velikost, maximální pracovní hodinovou frekvenci, spotřebu elektrické energie a počet současně se měnících bitů na sběrnici na výstupu čítače – maximální Hammingovu vzdálenost [2] dvou sousedních stavů čítače MHVS). Vhodná volba kódování je klíčovým rozhodnutím a právě jí je věnován náš příspěvek. Dalším důležitým kritériem pro návrh čítače je volba mezi synchronním čítačem (*synchronous counter*, všechny registry klopí ve stejný okamžik), nebo asynchronním čítačem (*ripple counter*, registry klopí postupně). Základní rozdíly mezi oběma čítači a argumenty pro volbu jednoho či druhého postupu budou také diskutovány v textu.

Jako příklad je v textu užit jednoduchý blok čítače procházejícího osmi stavy; pro jednotlivé alternativy je prezentován i VHDL kód návrhu a implementační parametry. Ty jsou pak v závěru článku shrnuty a vzájemně srovnány. Pro simulaci i implementaci bylo použito volně dostupné návrhové prostředí ISE Webpack [3]; jednotlivé bloky byly implementovány do obvodu xc5vlx30-3ff324. Začínající návrháři mohou nalézt detailní návod jak pracovat s návrhovým prostředím v knize [1], kapitola 2.

V celém textu označujeme počet registrů udržujících stav čítače jako N, počet stavů jako N_s. Jako f_{clk_max} označujeme maximální dosažitelnou pracovní frekvenci čítače, $T_{clk_min}=1/f_{clk_max}$ je pak minimální perioda hodinového cyklu.

Řešení popsaná v textu jsou použitelná jak při návrhu číslicových obvodů na programovatelných hradlových polích, tak při návrhu zákaznických integrovaných obvodů. Pro jednotlivé možnosti implementace číslicové logiky uvádíme v závěru seriálu kromě odhadů velikosti vlastní logiky a délky kritické cesty v čítači i méně často diskutovanou vlastnost – stručnou analýzu možností „ruční editace" návrhu čítače v případě nutnosti provedení tzv. ECO úpravy (*Engineering Change Order*, viz [4]) už vyráběného integrovaného obvodu.

Příspěvek je z prostorových důvodů rozdělen do několika dílů; v tomto příspěvku bude prezentována plná implementace binárního čítače spolu se synchronním binárním a Johnsonovým čítačem. V dalším příspěvku potom ukážeme implementaci Grayova čítače, čítače v kódu 1 z N a binárního asynchronního čítače spolu se závěrečným shrnutím jejich vlastností.

Základní implementace

Před rozborem jednotlivých alternativ implementace čítačů nejprve shrňme základní funkce, které může čítač mít; spolu s výkladem nechť čtenář sleduje výpis v příkladu 1:

Asynchronní a synchronní reset (*asynchronous/synchronous reset*) – reset aktivní buď vždy, nebo jen v okamžiku příchodu náběžné hrany hodin; čítač je většinou nastaven do stavu 00...000 (je-li binární vzestupný) nebo do stavu např. 11...111 (binární čítající sestupně), případně ve speciálních případech může být obvod resetován i do jiného stavu. Poznamenejme zde, že reset je nezbytnou součástí implementace a není rozumné ho vynechávat, více viz [1], kapitola 5. V případě použití asynchronního resetu se na signálu nesmí vyskytovat žádné hazardní stavy (*glitche*), nesmí být tedy generován přímo z obecné kombinační logiky. V příkladu 1 je funkce asynchronního resetu ovládána vstupem *async_res*, synchronního *sync_res*.

Synchronní a asynchronní přednastavení (*synchronous/asynchronous preload*) – stavový registr čítače je nastaven na hodnotu přivedenou na k tomu určenou vstupní sběrnici obvodu a aktivuje se příslušným řídicím signálem. Tím se také liší přednastavení od resetu. Reset čítač nastavuje vždy do stejného počátečního stavu. Synchronní přednastavení je v příkladu 1 řízeno signálem *sync_ld*, asynchronní *async_ld*. Hodnota, do které má být čítač přednastaven, je přivedena na sběrnici *ld_val*. Ani na signálu pro řízení asynchronního přednastavení se nesmí vyskytovat žádné hazardní stavy, jejich přítomnost by porušila stav čítače.

Čítání (*counting*) – stav čítače je u binárního čítače inkrementován nebo dekrementován s daným krokem (většinou 1, ale může být jiný), případně u jiného kódování je realizován přechod na následující/předchozí stav. Při čítání můžeme procházet jak úplnou sekvencí stavů čítače (pro Nbitový binární čítač je počet stavů $N_s=2^N$), tak sekvencí redukovanou. O $N_s$stavovém čítači často hovoříme jako o „čítači modulo N_s“. Demonstrační čítač má vstupní signály *cnt_en* – povolení čítání, *cnt_up* – směr čítání (nahoru/dolů), *cnt_step* – krok čítání. Počet stavů, kterými čítač může procházet, je definován vstupní sběrnicí *cnt_limit*, dosažení daného počtu stavů je pak indikováno pulzem na výstupním signálu *cnt_over*. Všimněte si také implementace detekce podtečení čítače – přechodu přes nulu v případě čítání dolů. Je zde využito vlastností binárního kódu a podtečení detekováno pomocí detekce situace, kdy je požadovaný příští stav „negativní“ (nejvyšší bit je v log. 1) a současný stav „pozitivní“ (nejvyšší bit je v log. 0).

Všimněte si v příkladu 1 implementace vynulování čítače po dosažení posledního stavu; reset je implementován jako synchronní. Bylo by hrubou chybou čítač nulovat pomocí asynchronního resetu, například takto:

```
reg_cnt : PROCESS (clk, async_res, cnt_over_i, async_ld, ld_val)
BEGIN
  IF async_res='1' OR cnt_over_i='1' THEN
    cnt_q <= (OTHERS => '0');
  ELSIF async_ld='1' THEN
    cnt_q <= unsigned(ld_val);
  ELSIF clk'EVENT AND clk='1' THEN
    cnt_q <= cnt_d;
  END IF;
END PROCESS reg_cnt;
```

Komparátor stavu čítače proti hodnotě *cnt_limit* bude téměř jistě na svém výstupu produkovat statické a dynamické hazardy a zákmit typu 0-1-0 by s jistotou vedl k poruše správné funkce čítače (stavový registr čítače by byl vynulován, případně by mohly být vynulovány jen některé registry ze stavového registru).

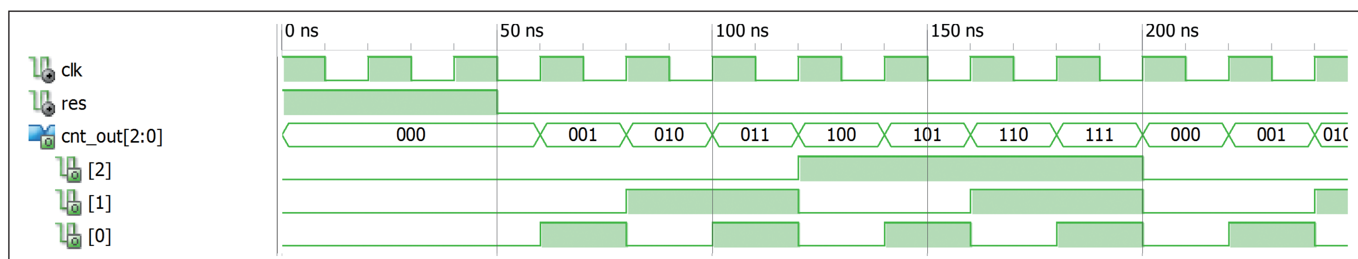
Příklad 1: Maximalistická verze binárního čítače se všemi běžnými funkcemi.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY binary_counter IS
    GENERIC (
      N   : natural := 16
    );
    PORT (
      async_res : IN  std_logic;
      async_ld  : IN  std_logic;
      clk       : IN  std_logic;

      sync_res  : IN  std_logic;
      sync_ld   : IN  std_logic;
      ld_val    : IN  std_logic_vector (N-1 DOWNTO 0);

      cnt_en    : IN  std_logic;
      cnt_up    : IN  std_logic;
      cnt_step  : IN  std_logic_vector (N-1 DOWNTO 0);
      cnt_limit : IN  std_logic_vector (N-1 DOWNTO 0);

      cnt_over  : OUT std_logic;
      cnt_unde  : OUT std_logic;
      cnt_out   : OUT std_logic_vector (N-1 DOWNTO 0)
    );
END ENTITY binary_counter;

ARCHITECTURE rtl OF binary_counter IS
  SIGNAL cnt_nx     : unsigned (N-1 DOWNTO 0);
  SIGNAL cnt_d      : unsigned (N-1 DOWNTO 0);
  SIGNAL cnt_q      : unsigned (N-1 DOWNTO 0);
  SIGNAL cnt_over_i : std_logic;
  SIGNAL cnt_unde_i : std_logic;
BEGIN
```

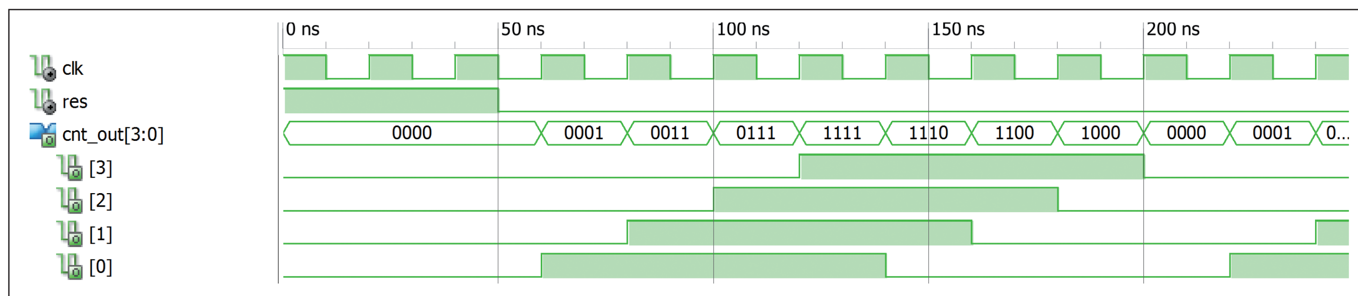
```
  reg_cnt : PROCESS (clk, async_res, async_ld, ld_val)
  BEGIN
    IF async_res='1' THEN
      cnt_q <= (OTHERS => '1');
    ELSIF async_ld='1' THEN
      cnt_q <= unsigned(ld_val);
    ELSIF clk'EVENT AND clk='1' THEN
      cnt_q <= cnt_d;
    END IF;
  END PROCESS reg_cnt;

  cnt_nx  <= cnt_q+unsigned(cnt_step)  WHEN cnt_en = '1' AND cnt_up = '1'  ELSE
             cnt_q-unsigned(cnt_step)  WHEN cnt_en = '1' AND cnt_up = '0'  ELSE
             cnt_q;
  cnt_over_i <= '1'  WHEN cnt_nx >= unsigned(cnt_limit)           ELSE
                '0';
  cnt_unde_i <= '1'  WHEN cnt_nx(N-1) = '1' AND  cnt_q(N-1) = '0' ELSE
                '0';
  cnt_d   <= cnt_nx              WHEN sync_res   = '1' AND sync_ld = '0' AND cnt_over_i = '0' ELSE
            (OTHERS => '0')      WHEN sync_res   = '1' ELSE
            unsigned(ld_val)     WHEN sync_ld    = '1' ELSE
            unsigned(cnt_limit) WHEN cnt_unde_i = '1' ELSE -- underflow
            (OTHERS => '0'); -- overflow

  cnt_over <= cnt_over_i;
  cnt_unde <= cnt_unde_i;
  cnt_out  <= std_logic_vector(cnt_q);
END ARCHITECTURE rtl;
```

Synchronní binární čítač

Binární kód je všem návrhářům dobře známý. Jeho výhodou je minimální počet registrů nutných pro implementaci čítače procházejícího N_s stavy, $N=ceil(log_2(N_s))$, kde *ceil* je funkce zaokrouhlující svůj argument nahoru na celé jednotky. Dále je snadné provádět nad výstupem binárního čítače nejrůznější aritmetické operace (například porovnávání – >,< atd.), jejich implementace v binárním kódu je velmi intuitivní a běžně používaná. Výhodou je i to, že čítač čítající v binárním kódu může mít libovolný počet stavů. Nevýhodou synchronního binárního čítače je potřeba většího množství kombinační logiky pro implementaci sčítačky pro generování dalšího stavu a větší zpoždění v této logice, které omezuje f_{clk_max}. Poslední významnější nevýhodou je skutečnost, že sousední stavy binárního čítače se mohou lišit až v *N* bitech (MHVS je *N*); kombinační logická



Obr. 2 Příklad běhu čítače, sekvence stavů 000, 001, 010, 011, 100, 101, 110, 111.



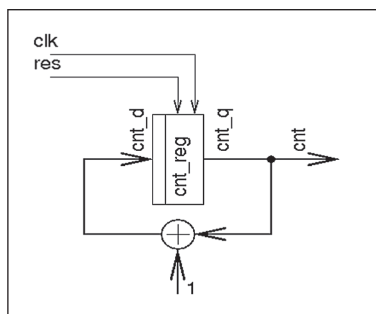
Obr. 3 Příklad běhu čítače, sekvence stavů je 0000, 0001, 0011, 0111, 1111, 1110, 1100, 1000.

funkce, která by dekódovala stavy čítače a generovala podle nich příslušné výstupy jako např. blok *decoder* na obrázku 1, pak bude téměř jistě produkovat statické či dynamické hazardy (*glitche*) na svém výstupu.

Pokud bychom chtěli upravovat čítač pomocí ECO změny, můžeme chtít buď jen zvýšit počet stavů (přidat další stavy na konec sekvence), nebo sekvenci „rozšířit“ vložením nových stavů „doprostřed“. Kombinační logická funkce binárního čítače patří mezi složitější z pohledu ECO změn; většinou lze prodloužit sekvenci přidáním dalších stavů, ale není možné vložit stavy dovnitř do sekvence čítače (čítač by pak už nečítal v binárním kódu).

Příklad 2: Binární čítač – RTL kód a schéma

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ENTITY cnt IS
  GENERIC (
      N   : natural := 4
  );
  PORT (
    res      : IN  std_logic;
    clk      : IN  std_logic;
    cnt_out : OUT std_logic_vector (N-1 DOWNTO 0)
  );
END ENTITY cnt;
ARCHITECTURE rtl_bin OF cnt IS
  SIGNAL cnt_d : unsigned (N-1 DOWNTO 0);
  SIGNAL cnt_q : unsigned (N-1 DOWNTO 0);
BEGIN
  reg_cnt : PROCESS (clk, res)
  BEGIN
      IF res='1' THEN
        cnt_q <= (OTHERS => '0');
      ELSIF clk'EVENT AND clk='1' THEN
        cnt_q <= cnt_d;
      END IF;
  END PROCESS reg_cnt;
  cnt_d    <= cnt_q + 1;
  cnt_out <= std_logic_vector(cnt_q);
END ARCHITECTURE rtl_bin;
```



Příklad 2 uvedený níže demonstruje už jen zjednodušenou verzi čítače z příkladu 1, sekvenci stavů demonstruje *obrázek 2*.

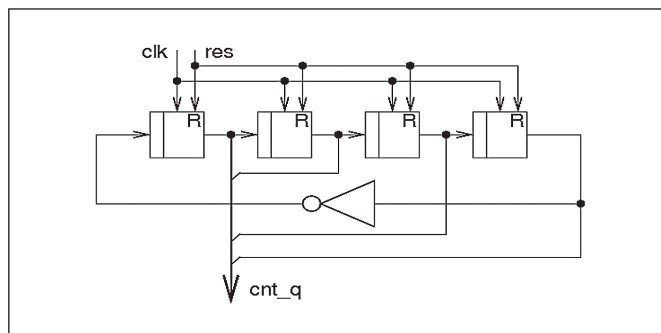
Synchronní Johnsonův čítač

Stav a výstup Johnsonova čítače je kódován v Johnsonově nebo tzv. plazivém kódu. Kód se konstruuje jednoduše – první stav je zakódován samými nulami; pro další stavy se stavový registr posouvá vlevo a zprava nasouvá jednička až do okamžiku, kdy jsou všechny registry nastavené do log. 1. Potom se celý proces opakuje s tím rozdílem, že zprava nasouváme log. 0 až do okamžiku, kdy jsou všechny registry stavu čítače nastavené do log. 0. Pak celý postup opakujeme. Příklad sekvence v Johnsonově kódu je uveden níže v *obrázku 3*.

Výhodou Johnsonova kódu je to, že MHVS je jen 1, proto je Johnsonův čítač často používán například v generátorech hodinových signálů (navazující kombinační logika bloku *decoder* generující vlastní hodinové signály má za dodržení dalších dodatečných podmínek výstup bez statických i dynamických hazardů). Nevýhodou je, že počet registrů udržujících stav čítače je zde $N=N_s/2$; dále je možná jen implementace čítače o sudém počtu stavů. Všimněme si zde, že mezi registry čítače v podstatě není žádná kombinační logika, jen jeden invertorm který invertuje výstup z posledního registru a vytváří „plazivou“ sekvenci nul a jedniček. To umožňuje dosáhnout vysoké hodnoty f_{clk_max}. Do čítače implementovaného jako Johnsonův lze proto snadno vložit pomocí ECO změny další stavy a prodloužit tak sekvenci čítání.

Příklad 3: Johnsonův čítač – RTL kód a schéma

```
ARCHITECTURE rtl_johnson OF cnt IS
  SIGNAL cnt_d : std_logic_vector (N-1 DOWNTO 0);
  SIGNAL cnt_q : std_logic_vector (N-1 DOWNTO 0);
BEGIN
  reg_cnt : PROCESS (clk, res)
  BEGIN
      IF res='1' THEN
        cnt_q <= (OTHERS => '0');
      ELSIF clk'EVENT AND clk='1' THEN
        cnt_q <= cnt_d;
      END IF;
  END PROCESS reg_cnt;
  cnt_d    <= cnt_q(N-2 DOWNTO 0)&NOT(cnt_q(N-1));
  cnt_out <= cnt_q;
END ARCHITECTURE rtl_johnson;
```



Literatura:
[1] Jakub Šťastný. FPGA Prakticky, BEN Praha 2011.
[2] Hamming Distance. http://en.eikipedia.org/wiki/Hamming_distance
[3] ISE WebPack. http://www.xilinx.com/tools/webpack.htm
[4] Engineering Change Order. http://en.wikipedia.org/wiki/Engineering_Change_Order
[5] Steve Golson. One-hot state machine design for FPGAs, 3rd PLD Design conference, Santa Clara, 1993.
[6] Steve Golson. State machine design techniques for Verilog and VHDL, 1994.