B i b l i o g r a f i c k á c i t a c e

HUZLÍK, P. *Vzorové úlohy ve VHDL*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2008. 80 s. Vedoucí bakalářské práce Ing. Radovan Holek, CSc.

Prohlášení

„Prohlašuji, že svou bakalářskou práci na téma "Vzorové úlohy ve VHDL" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.
Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne : 2.6.2008 Podpis: _ _ _ _ _ _ _ _ _

Poděkování

Za odbornou pomoc děkuji mému vedoucímu bakalářské práce Ing. Radovanu Holkovi, CSc.

1. ÚVOD

Tato bakalářská práce navazuje na semestrální projekt a zabývá se jazykem VHDL a obvody FPGA a CPLD firmy Xilinx. Tato práce má ale především za cíl popsat, jak zacházet s vývojovým prostředím WebPack, kde chci ukázat, jak realizovat nový projekt a především popsat různé metody návrhu úlohy v tomto vývojovém prostředí. Je totiž velmi důležité jakou metodu návrhář zvolí. Na konec jsou také v této práci uvedeny některé vzorové příklady v jazyku VHDL.

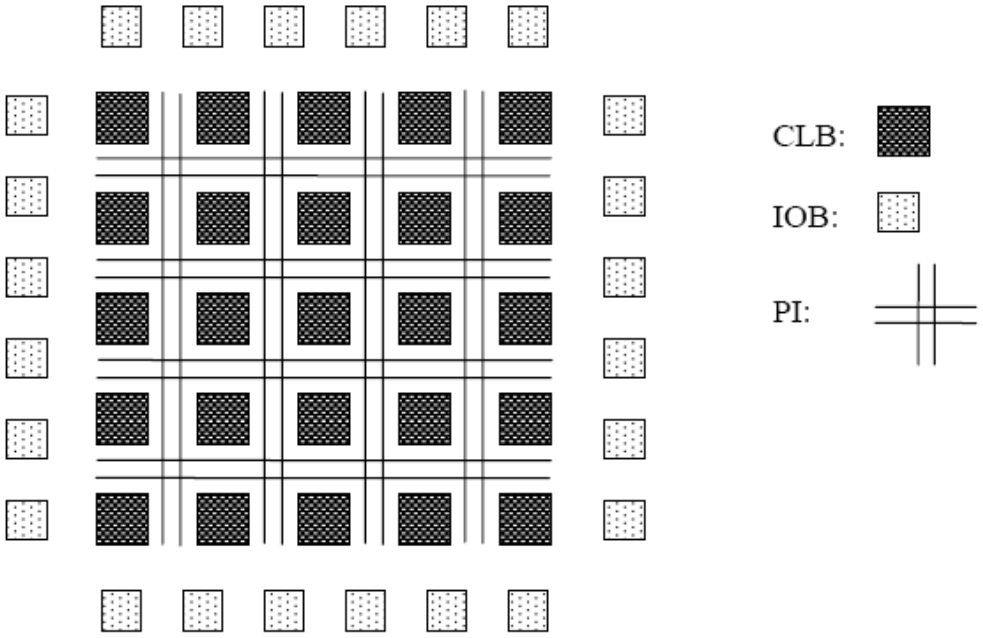
1.1 Obvody PLD

Obecně se Programovatelné součástky PLD (Programmable Logics Device) dají dále rozdělit na CPLD (Comlexed PLD) a FPGA (Field Programmable Gate Array). Struktura PLD obvodu se skládá především z bloku AND a OR. Podle toho, který z těchto bloků je možné naprogramovat můžeme dále PLD rozdělit na:

- strukturu PROM (Programmable Read Only Memory) – blok AND je zapojen pevně a blok OR je programovatelný.
- strukturu PAL (Programmable Array Logic) – blok AND je programovatelný, stupeň OR je zapojen pevně.
- strukturu PLA (Programmable Logic Array) – oba bloky jsou programovatelné.

1.2 Obvody FPGA

Na obr. 1 je zobrazena bloková struktura obvodů FPGA, která je tvořena polem konfigurovatelných logických bloků CLB (Configurable Logic Block). Tyto bloky lze přirovnat k podblokům CPLD. Tyto podbloky, neboli logické buňky se dále dělí ještě na menší části. Bloky CLB jsou propojeny propojovací strukturou PI (Programmable Interconnect). Bloky CLB jsou dále obklopeny vstupně - výstupními bloky IOB (Input / Output Block).



Obr. 1. *Základní bloková struktura obvodů FPGA.* [1]

V praxi v podstatě neexistuje aplikace, která by nešla FPGA obvodem realizovat. Tyto obvody jsou ale velmi složité a jejich nevýhodou je vysoká cena. Další nevýhodou FPGA obvodu je to, že si nepamatují naprogramovanou konfiguraci, oproti obvodům CPLD.

Některé FPGA obvody od firmy Xilinx typu Spatran:

- Spartan – XL:-nepodporuje systém WebPack a jeho architektura je typu XC4000
- Spartan – II:- podporuje systém WebPack a jeho architektura je typu Virtex
- Spartan – IIE:- podporuje systém WebPack a jde o rychlejší verzi Spartan–II

1.3 Obvody CPLD

Obvody CPLD, neboli Complex PLD obsahují několik bloků, které jsou složeny z makrobuněk (struktura PAL, PLA). Tyto makrobuňky (Macrocells) jsou mezi sebou propojeny programovatelnou strukturou. Obvody jsou vyrobeny převážně v provedení EECMOS (nebo SRAM) s EEPROM pamětí, která je

integrována uvnitř. Největší výhodou oproti obvodům FPGA je tedy to, že po připojení napájení okamžitě fungují. Diky EEPROM paměti je tudíž lze naprogramovat, mazat a přeprogramovávat. Což u obvodů FPGA není možné. Další výhodou CPLD obvodů je jejich nízká spotřeba. Napájecí napětí je 5V, dále 3,3V, ale také 2,5V a nižší. Nejčastěji se používíají obvody od firmy Xilinx, například XC9500XL (technologie Flash memory) nebo o něco rychlejší XCR3000XL. Další nespornou výhodou těchto obvodů je jejich nízká cena.

1.4 Srovnaní FPGA / CPLD

Ikdyž jsou FPGA a CPLD obvody na první pohled velice podobné součástky, tak unitř pracují naprosto odlišně. Odlišnost spočívá v rozdílné struktuře součástek. Ve zkratce se da říct, že obvody FPGA jsou větší, složitější, pomalejší a dražší. Narozdíl od obvodů CPLD, které jsou naopak menší, rychlejší a levnejší. Nové obvody FPGA jsou již také velmi rychlé, ale jejich cena je pomerně vysoká a další velkou nevýhodou je to, že je nelze naprogramovat.

1.5 VHDL

VHDL zkratka vznikla jako akronym názvu Very High Speed Integrated Circuits Hardware Description Lenguage). Jde o programovací jazyk, který slouží pro návrh a simulaci programovatelných hradlových polí. Slouží především pro programovaní obvodů CPLD a FPGA.

1.5.1 Historie VHDL

Vývoj jazyka VHDL započal v roce 1981 v rámci výzkumného projektu VHSIC (Very High Speed Integrated Circuits) ministerstva obrany Spojených států. Původně byl tento jazyk vyvinut pro modelovaní a simulaci systému. Jazyk VHDL vznikl jako standart IEEE v roce 1987 a od té doby prošel různými revizemi. [3]

Mimo jazyk VHDL se taky používá jazyk Verilog, který je podobný jako jazyk VHDL, ale používá se především v asijských zemích. V Evropě se ujal především jazyk VHDL.

Mezi základní vlastnosti jazyka VHDL patří:

- **otevřený standart** (Open Standart) : - k použití není třeba licence
- **(Device Independent)** : - práce na návrhu je možná, aniž bychom zvolili cílový obvod .
- **přenositelnost** (Portability) : - je možná simulace, syntéza a implementace obvodu na základě téhož zdrojového kódu.

Různé způsoby navržení modelu:

- **Postup zdola nahoru** (buttom – up) : - v prvním kroku se vytvoří dílčí bloky modelu, které se potom seskupují do větších celků.

- **Postup shora dolů** (top – down) : - nejprve se nadefinuje funkce celého navrhovaného systému. A následně se vyčlení bloky. Ty pak mohou zpracovávat různí konstruktéři. Proces trvá do té doby, až se získají dostatečně jednoduché bloky. Jejich funkci lze popsat behaviorálním popisem. Tento způsob je vhodný u moderních systémů CAD.

- **Model plochého typu** (flat) : - konstrukce zde vypadá jako jeden monolitický blok.

1.5.2 Základy VHDL

Komentář se značí dvěma pomlčkami viz. následovně: -- takto vypadá komentář.V jazyce VHDL se nerozlišují malá a velká písmena.

Struktura modelu jazyka VHDL se dělí na dvě části:

1) Deklarace entity (Entity declaration):

Klíčovým slovem `PORT` se deklarují brány (Ports). Dále rozlišujeme různé mody:

- `IN` – vstup

- `OUT` – výstup
- `BUFFER` – vystup se zpětnou vazbou
- `INOUT` – obousměrný vývod

Rozlišujeme také typy dat, a to například:

- `bit`
- `bit_vektor`
- `stg_logic_1164`
- `stg_logic_vector`
- pro použití těchto typů musíme také použít klauzule `LIBRARY ieee` a `USE`

Příklad deklarace entity s názvem EqComp4: [2]

```
-- EqComp4
ENTITY EqComp4 IS
     PORT(a,b:IN bit_vector(3 DOWNTO 0);       -- vstupy
          Equals: OUT bit);                    -- výstup
END EqComp4;
```

PZN: - rozdíl mezi příkazem `DOWNTO` a `TO`:

(x……`DOWNTO`…….0) (0…….`TO`……..x)

x_n…….. x_3, x_2 , x_1 , x_0 x_0, x_1, x_2, x_3, ……..…...x_n

2) Tělo (popis) architektury (Architecture body):

Příklad popisu architektury: [2]

```
ARCHITECTURE Dataflow OF EqComp4 IS
BEGIN
     Equals <= '1' WHEN (a = b) ELSE '0';
END Dataflow;
```

Tělo architektury můžeme popsat různými styly:

a) Behaviorální styl (Behavioral):

Tento styl je asi nejpoužívanější . Je charakterizován pomocí konstraktu `PROCESS`. Tento konstrakt je použit v programu viz. níže a za tímto konstraktem následuje tzv. seznam citlivosti (Senzitivity list). Jde o signály, při jejichž změně se vyvolá další změna některého signálu. Změna tedy spouští provedení signálu.

Rozlišujeme také přiřazení signálu. A to na odložené přiřazení (`Equals <=´1´`) a bezprostřední přiřazení (proměnné `xvar := ´1´`).
Příklad programu na behaviorální styl: [2]

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY EqComp4 IS
     PORT(a,b: IN std_logic_vector(3 DOWNTO 0); -- vstupy
          Equals: OUT std_logic);           -- výstup
END EqComp4;

ARCHITECTURE Behavioral1 OF EqComp4 IS BEGIN
     Comp: PROCESS (a,b) BEGIN
       IF a = b THEN Equals <= '1';
                ELSE Equals <= '0';
       END IF;
     END PROCESS Comp;
END Behavioral1;
```

b) Styl popisující tok dat (Dataflow):

Tento styl je popsán v příkladu popisu architektury. Jde o vlastní případ behaviorálního stylu. Mimo prostých příkazů se používají příkazy `CASE – WHEN` a dále příkazy `WHIT – SLECT – WHEN`.

c) Strukturální styl (Structural):

Tento styl je založen na principu vkládaní komponent (Component) do kódového útvaru netlist. Nejčastější použití strukturálního stylu je pro popisovaní dílčích bloků na nižší úrovni viz příklad níže: [2]

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY EqComp4 IS
     PORT(a,b: IN std_logic_vector(3 DOWNTO 0);-- vstupy
          Equals: OUT std_logic);            -- výstup
END EqComp4;

USE work.Gatespkg.ALL
ARCHITECTURE Struct OF EqComp4 IS
     SIGNAL x: std_logic_vector(0 to 3);
BEGIN
     u0: Xnor2 PORT MAP (a(0),b(0),x(0));
     u1: Xnor2 PORT MAP (a(1),b(1),x(1));
     u2: Xnor2 PORT MAP (a(2),b(2),x(2));
     u3: Xnor2 PORT MAP (a(3),b(3),x(3));
     u4: And4 PORT MAP (x(0),x(1),x(2),x(3),Equals);
END Struct;
```

Zjednodušeně řečeno, můžeme strukturální popis považovat za jakýsi návrh schématu.

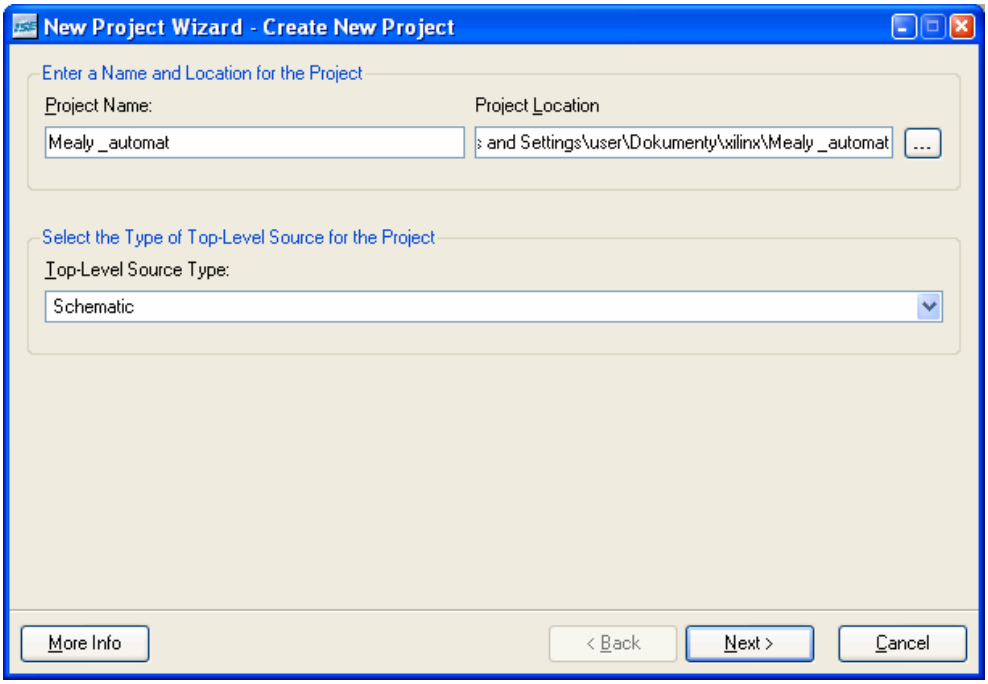
2. POPIS SW - XILINX ISE WEBPACK 8.2.03i

ISE SW umožňuje kompletní návrh SW pro CPLD a FPGA. Je vhodný tehdy, pokud není k dispozici základní programové vybavení od výrobce obvodu. Ceny vývojových prostředků jsou totiž poměrně vysoké. Nespornou výhodou ISE SW je to, ze je volně dostupný na stránkách Xilinx. Jedná se o omezenou verzi kompletního SW pro FPGA. Omezení spočívá hlavně v počtu hradlových polí, který je menší. Omezení je ale například i v tom, že ISE SW neobsahuje oproti kompletnímu SW některé součástky. ISE SW se ovšem chová jako plnohodnotný SW.

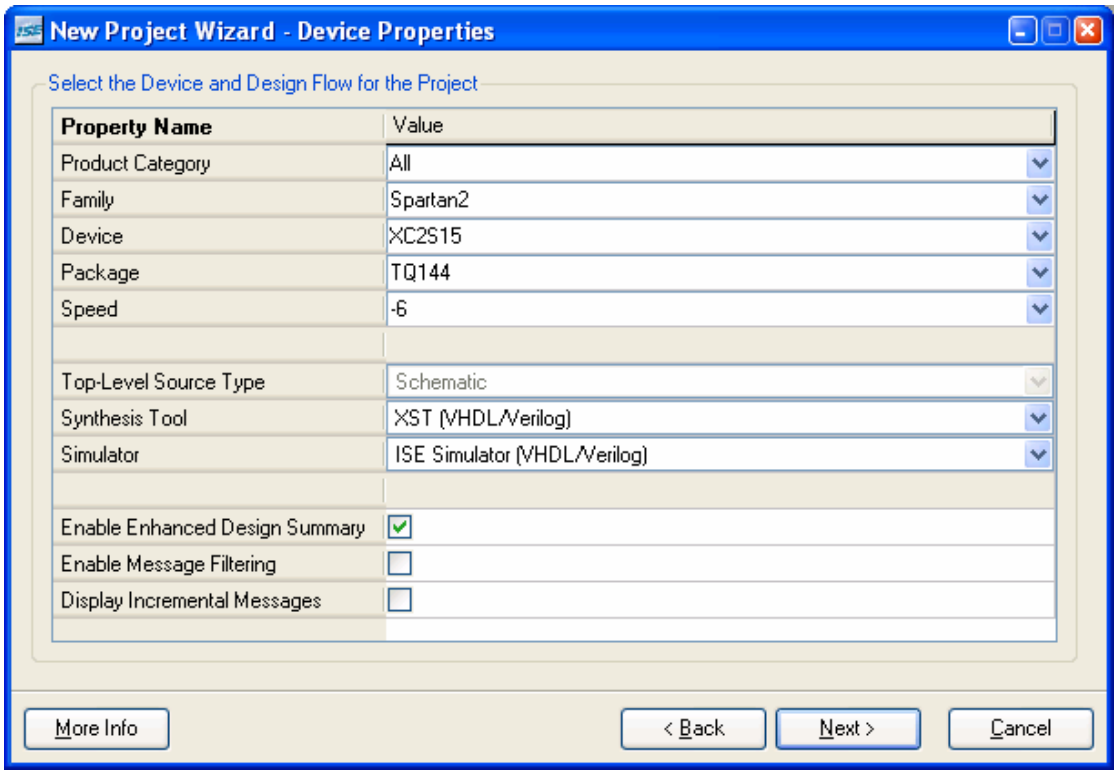
V následující kapitole je zdokumentován zcela obecný postup, jak vytvořit nový projekt v návrhovém prostředí. Následně jsou zde zdokumentovány i různé metody. Je zde ukázáno jak vytvořit projekt pomoci stavového diagramu, schématu, a pomocí VHDL. Nakonec je uvedeno, jak odsimulovat navržený projekt.

2.1 NOVÝ PROJEKT:

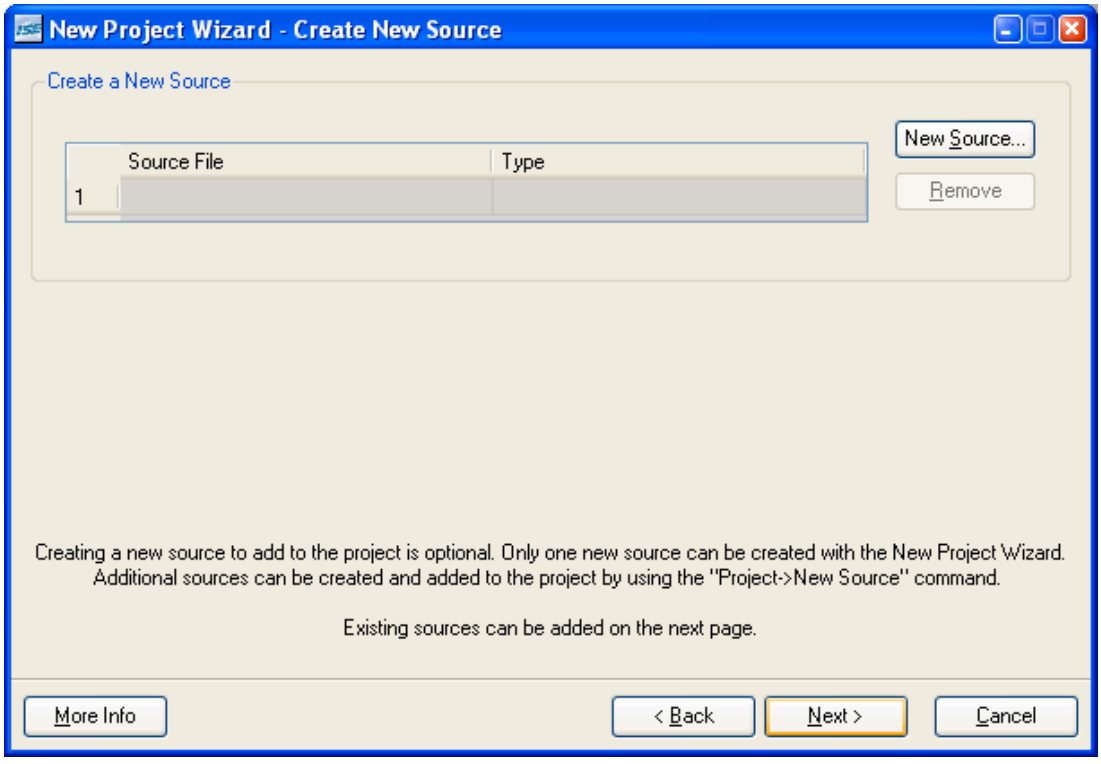
Nejprve zvolte „File“ – „New project“.



Napište název projektu a pokračujte tlačítkem „Next“.

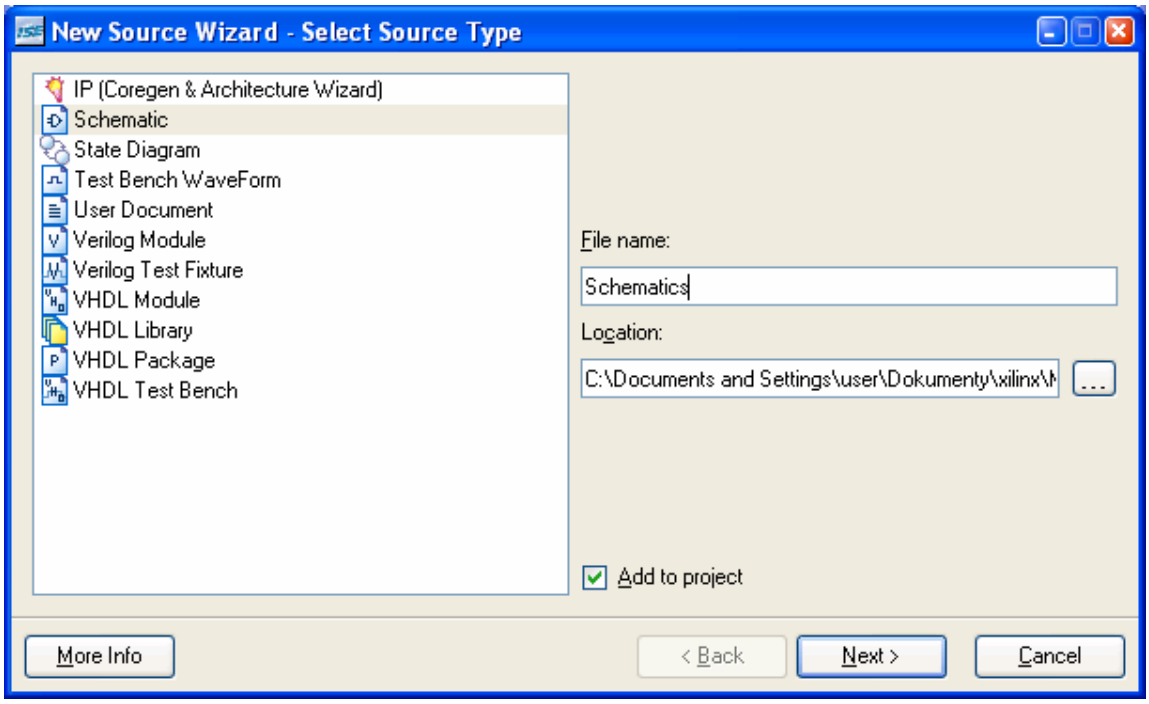


Tlačítko „Next“.

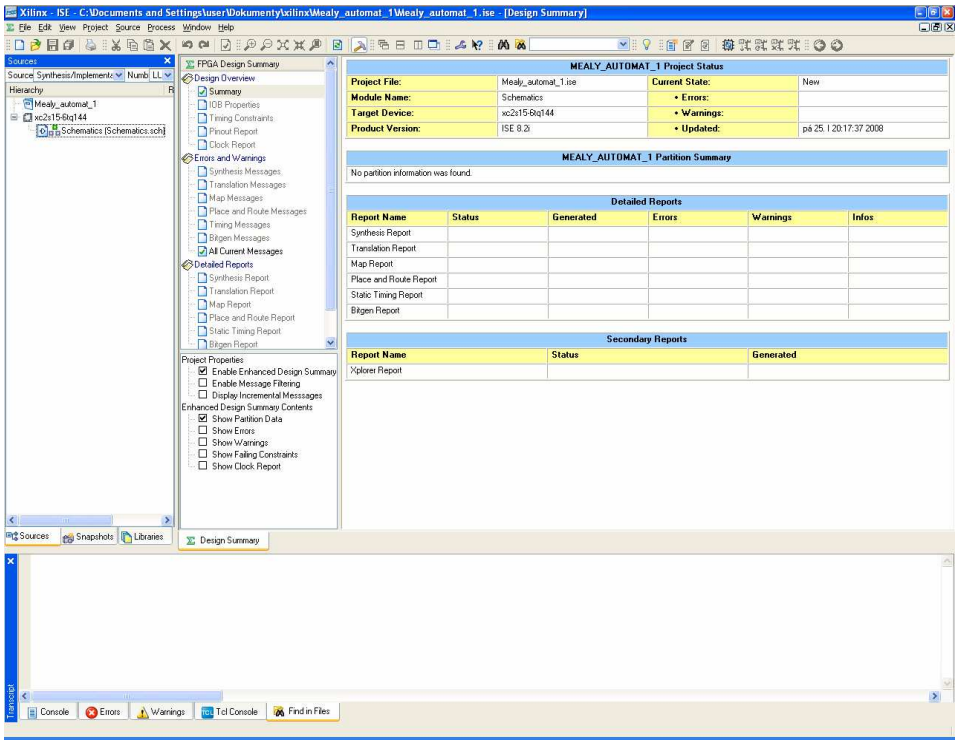


Tlačítko „Next“.

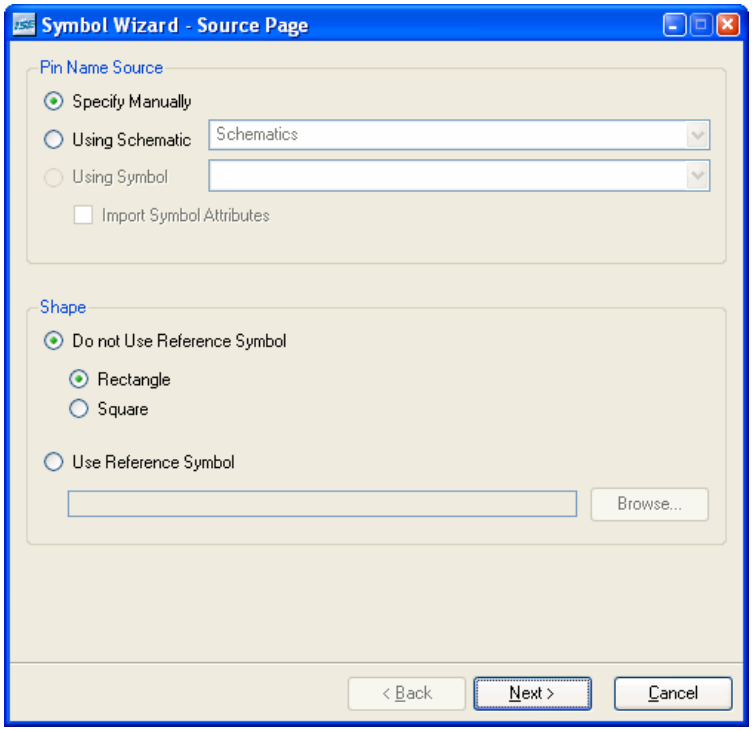
Dále je ukázáno, jak vytvořit hlavní součástku.



Tlačítko „Next“ a dále tlačítko „Finish“ .

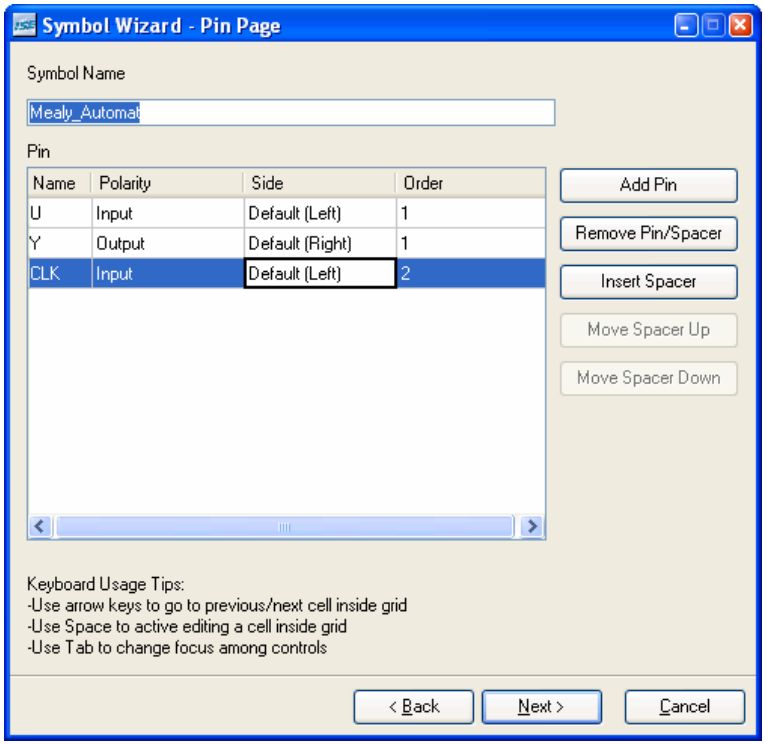


Dále zvolte „Tools“ – „Symbol Wizard“.

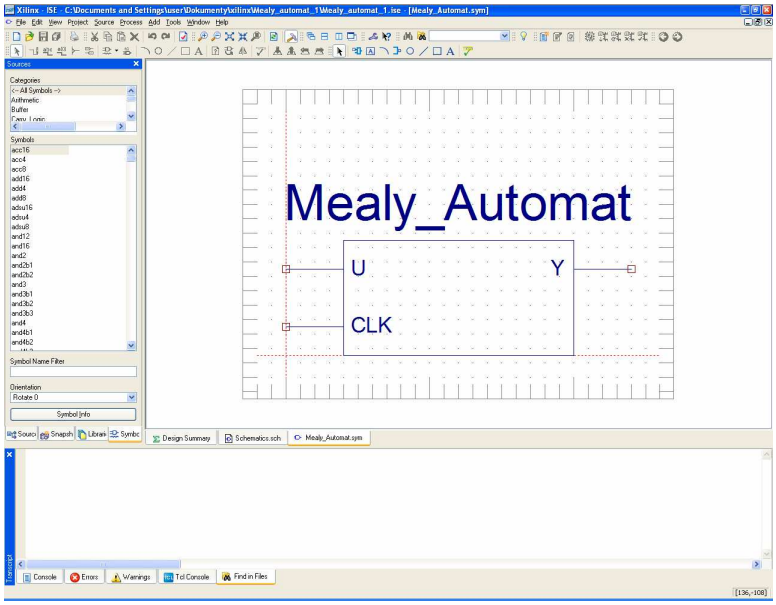


Tlačítko „Next“.

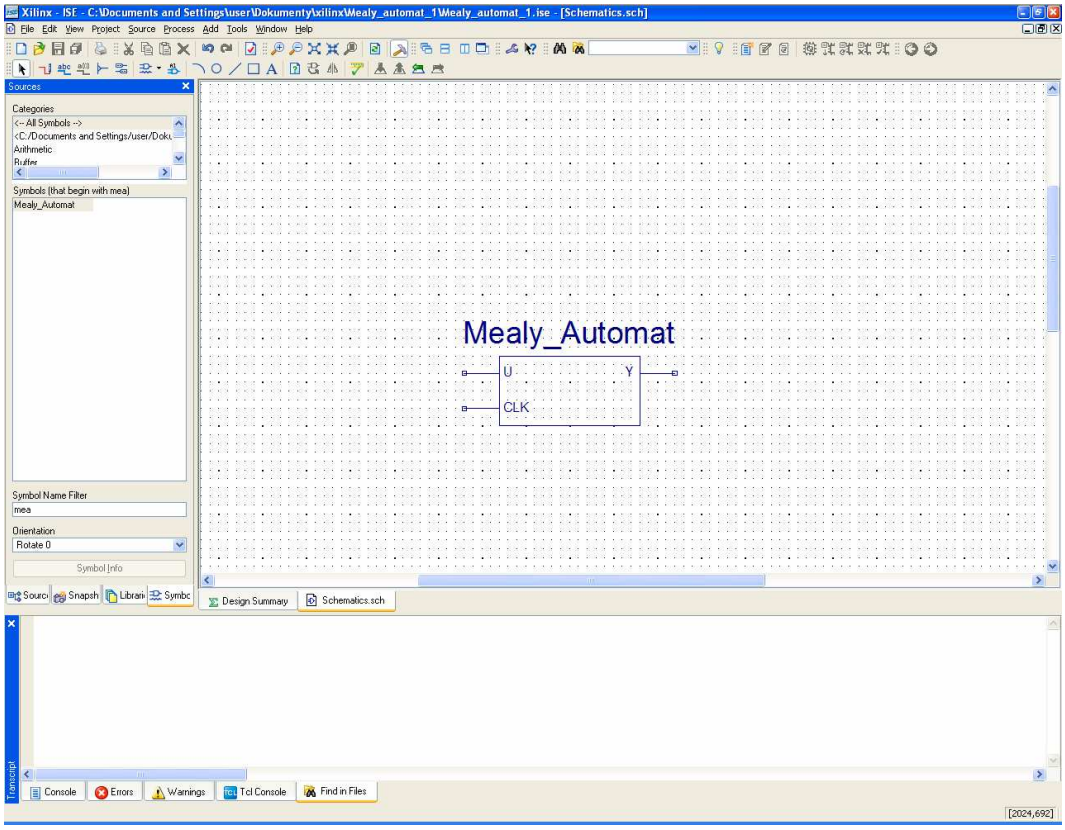
Následuje přiřazení vývodů.



Tlačitko “Next”, “Next”.

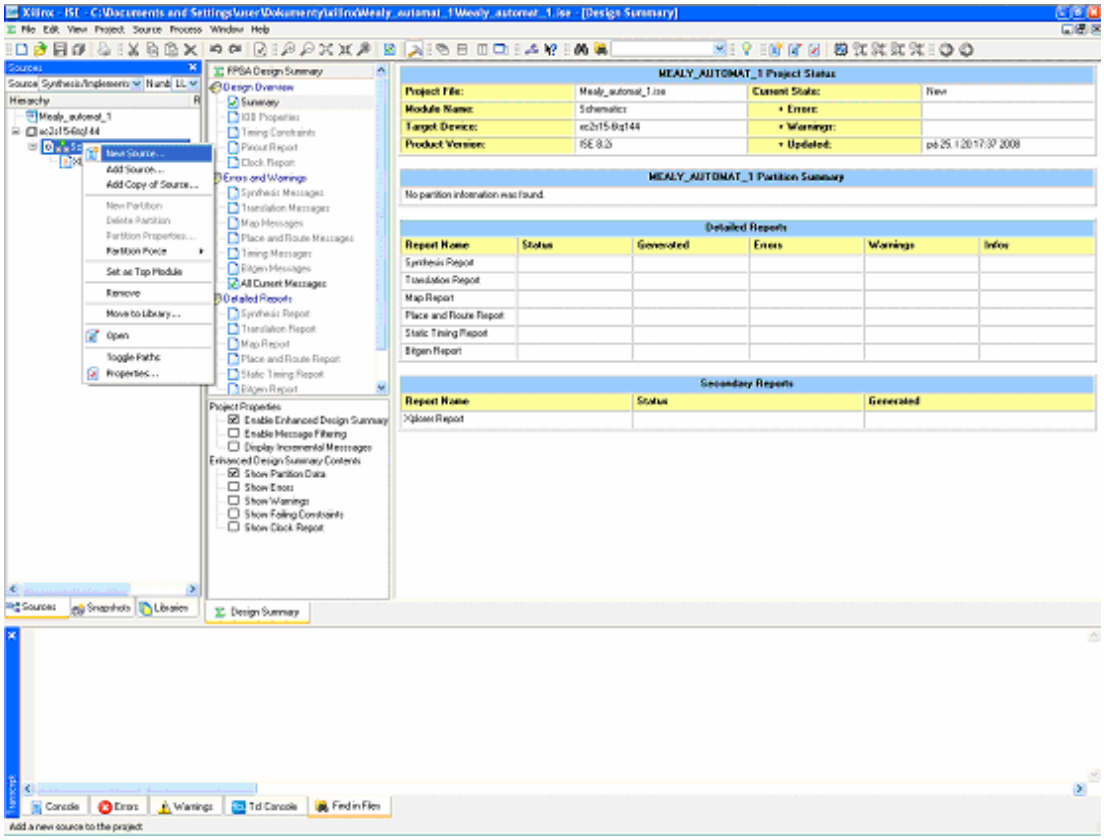


Přidejte nově vytvořenou součástku do schématu.

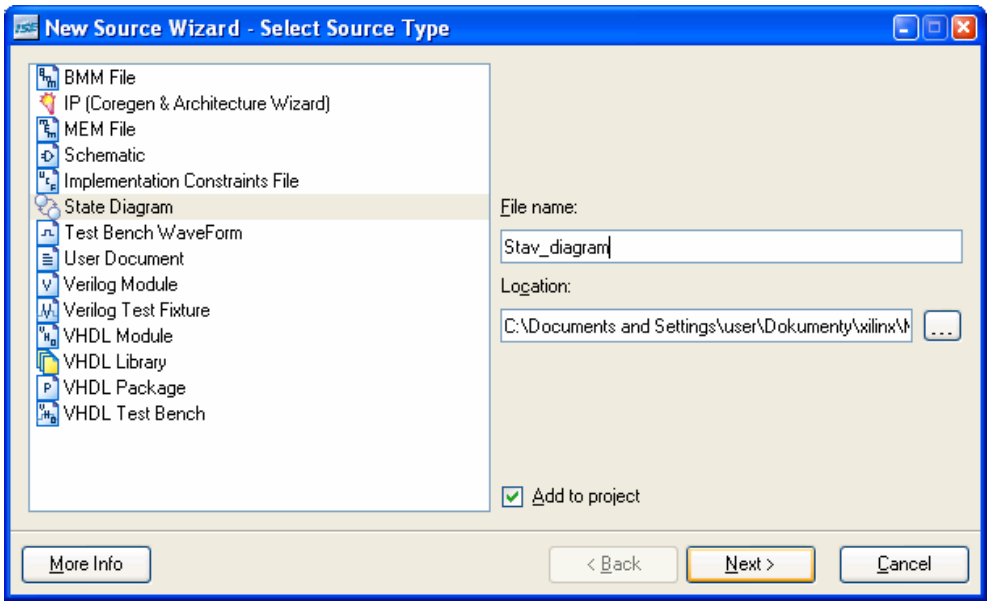


Hlavní součástka je již vytvořena. Dále lze postupovat pomocí některé z metod – stavový diagram , schéma, VHDL.

2.2 VYTVOŘENÍ POMOCÍ STAVOVÉHO DIAGRAMU

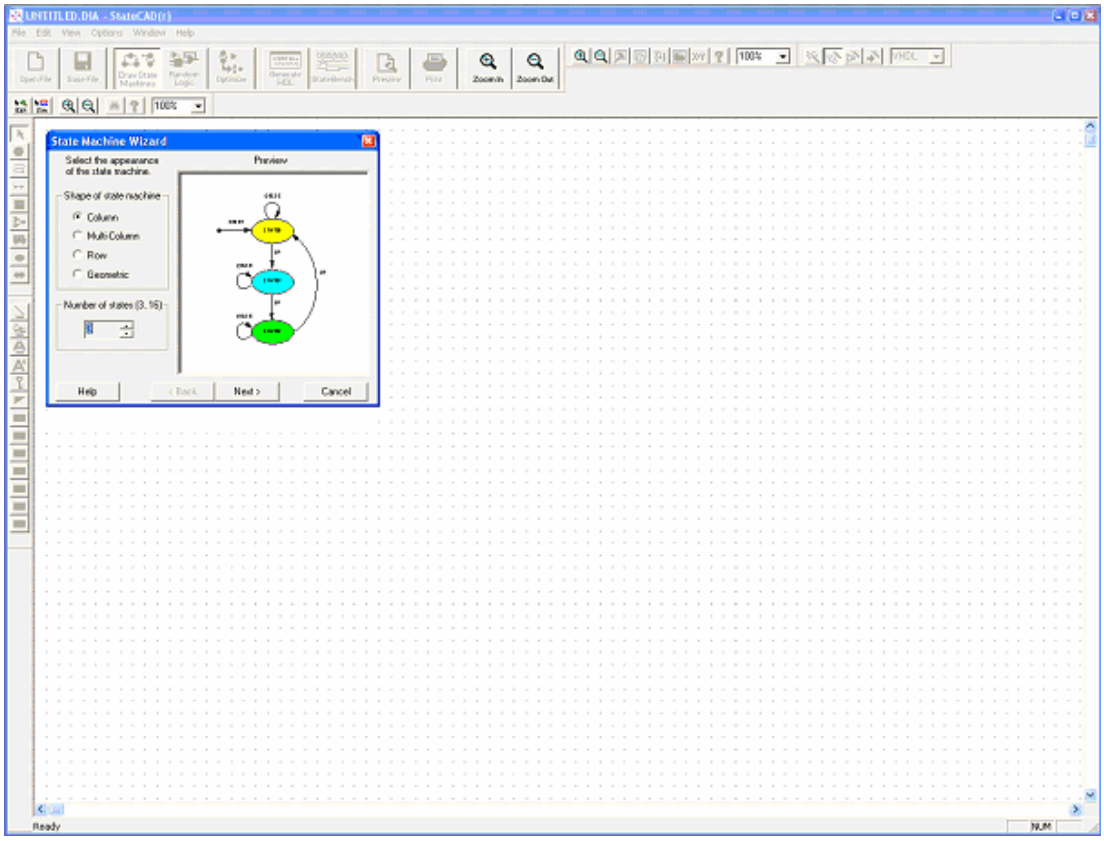


Přidejte zdroj do schématu.



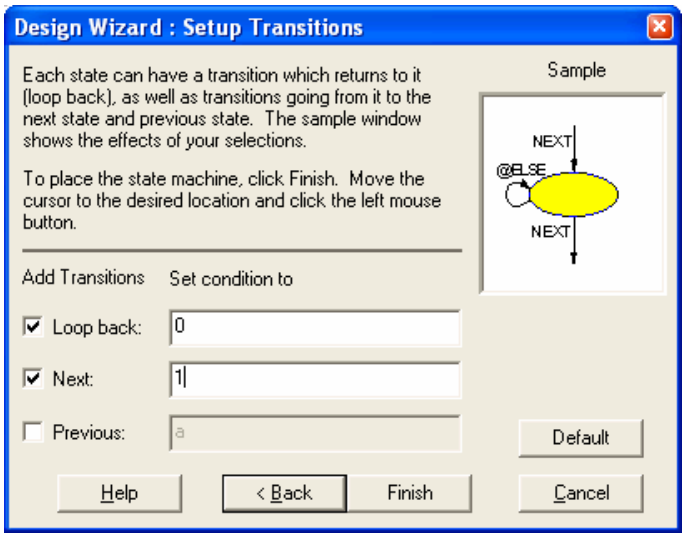
Stiskněte „Next“ pro vytvoření souboru.

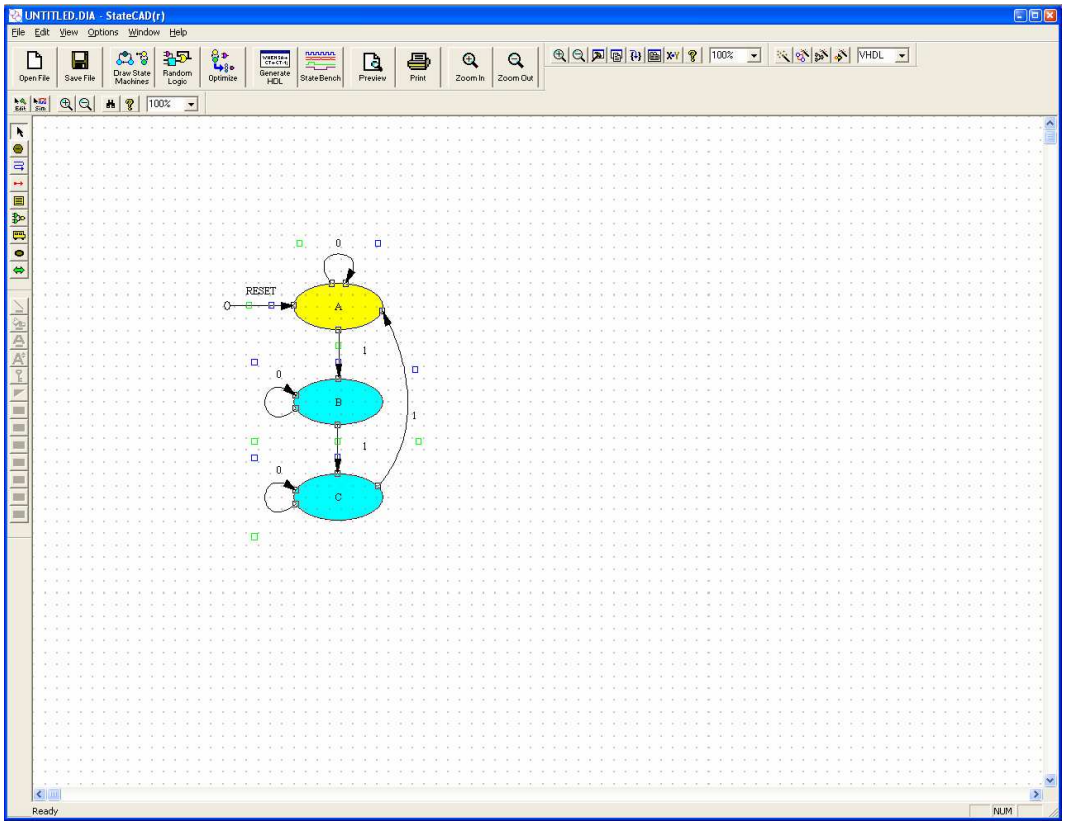
Otevře se okno pro vytváření stavového diagramu – viz. níže.



Zvolte, zda se jedná o synchronní nebo asynchronní.

Nastavte podmínky přechodu.



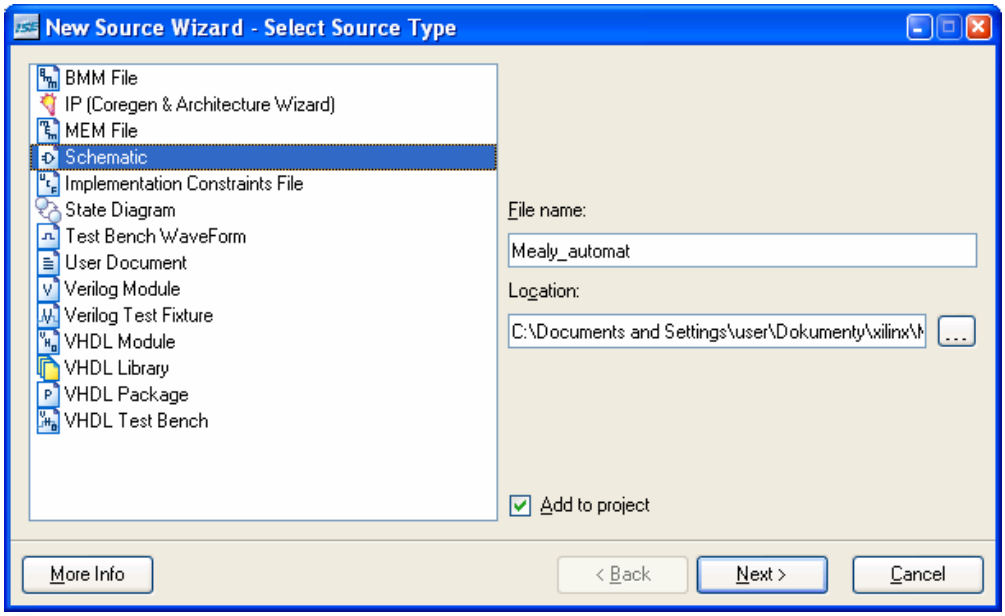


Simulaci vytvořeného stavového diagramu lze otestovat pomoci tlačítka „State Bench“.

Vygenerujte VHDL program pomocí tlačítka „Generate HDL“.

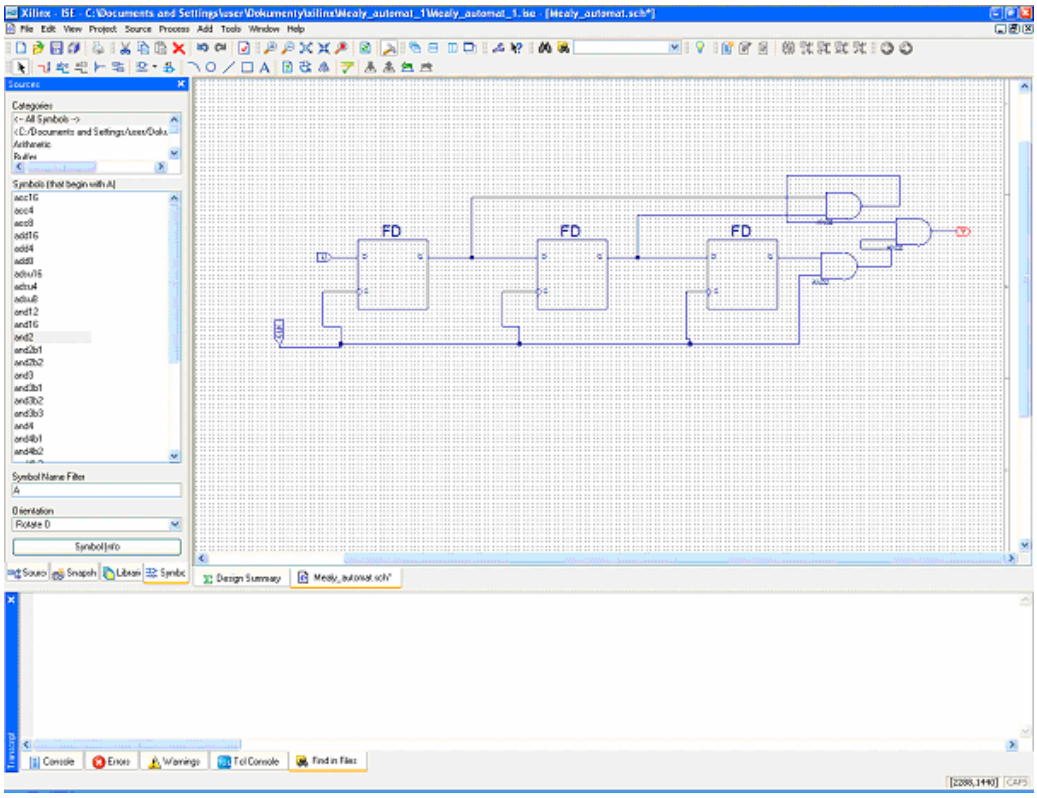
2.3 VYTVOŘENÍ POMOCÍ SCHÉMAT

Zvolte přidat nový zdroj a zvolte typ „schematics“.

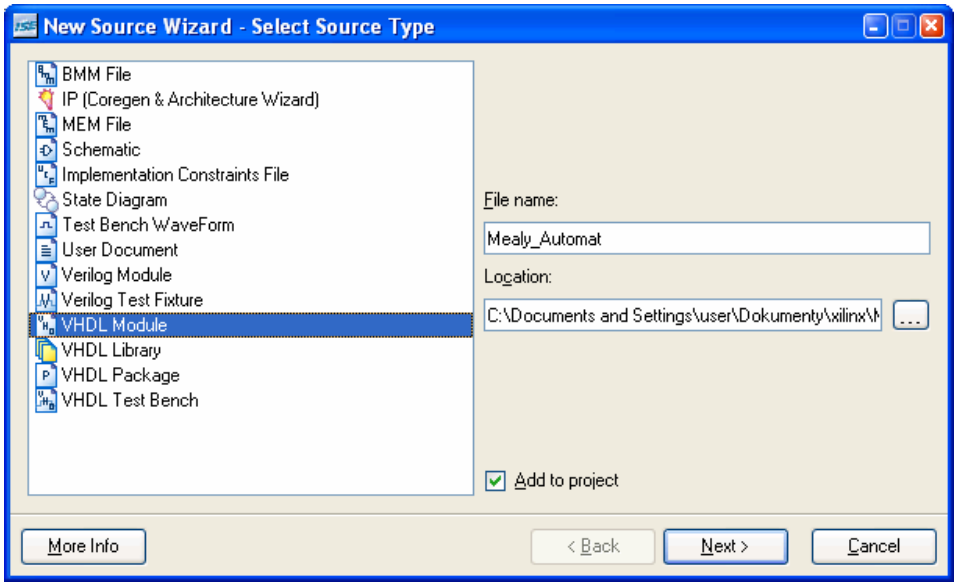


Dokončete vytvoření souboru „Next“, „Next“.

Nakreslete schéma v schematickém editoru.

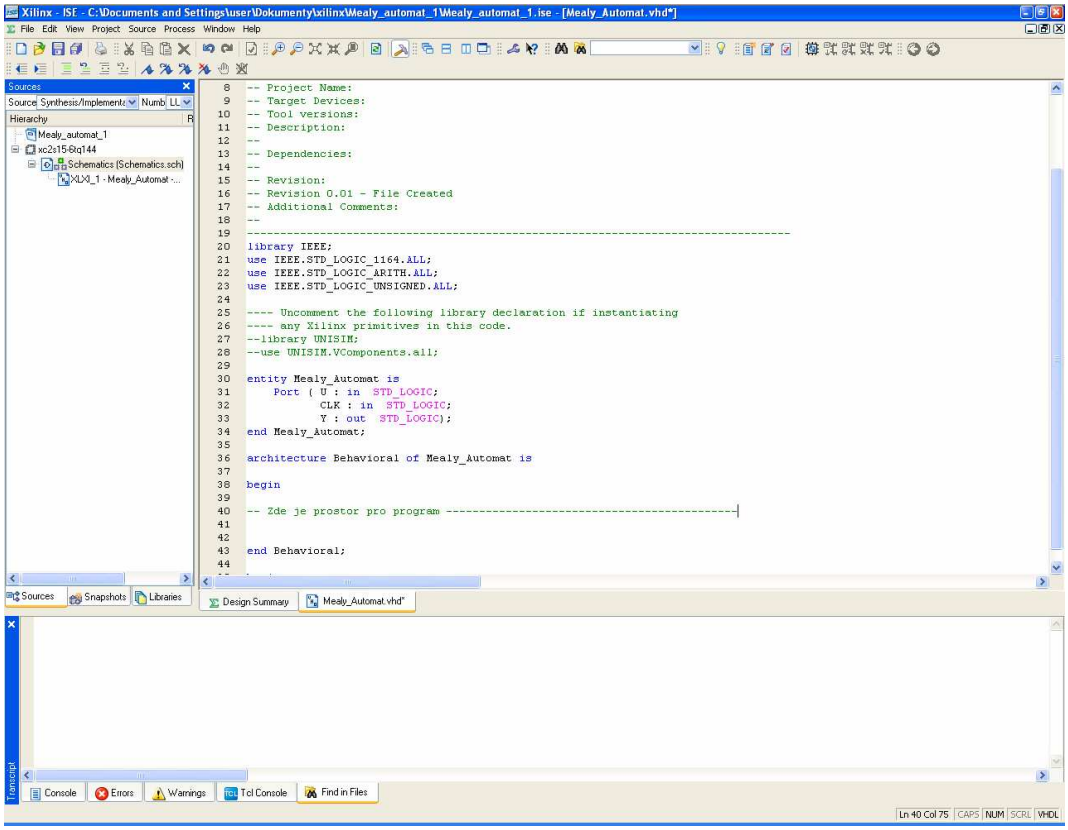


2.4 VYTVOŘENÍ AUTOMATU POMOCÍ VHDL



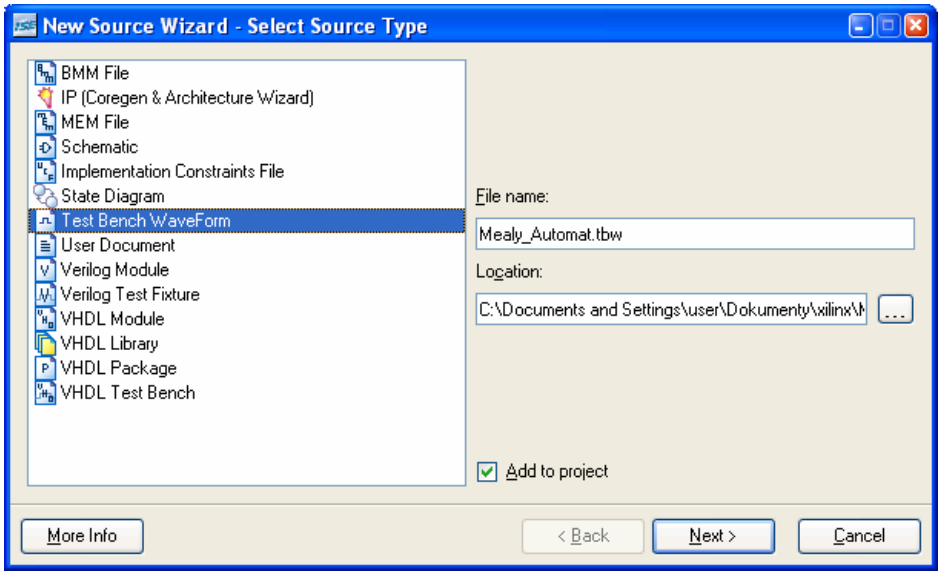
Přiřadíme vstupy a výstupy.

Dále lze již napsat program.

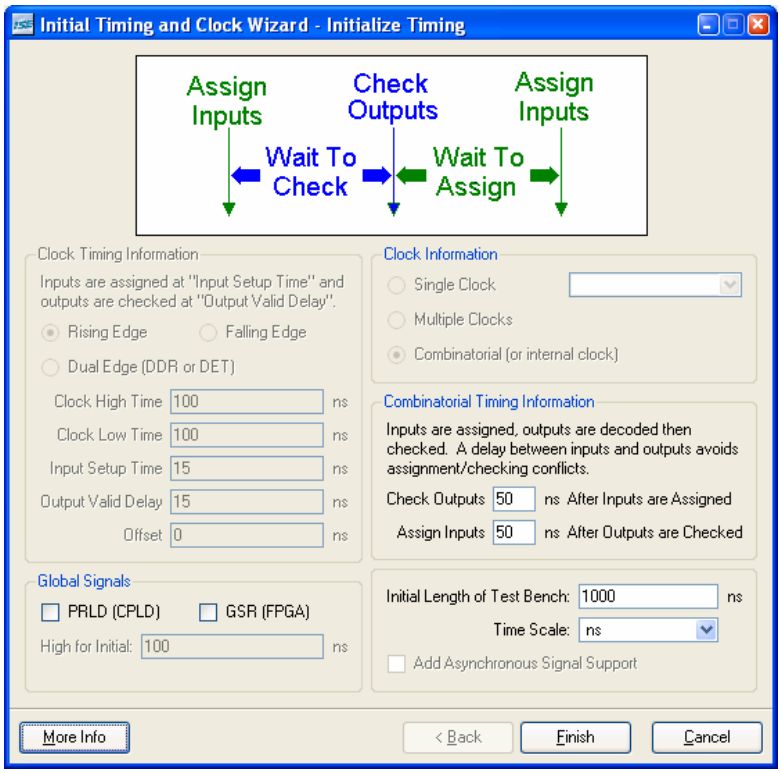


2.5 SIMULACE NAVRŽENÉHO PROJEKTU

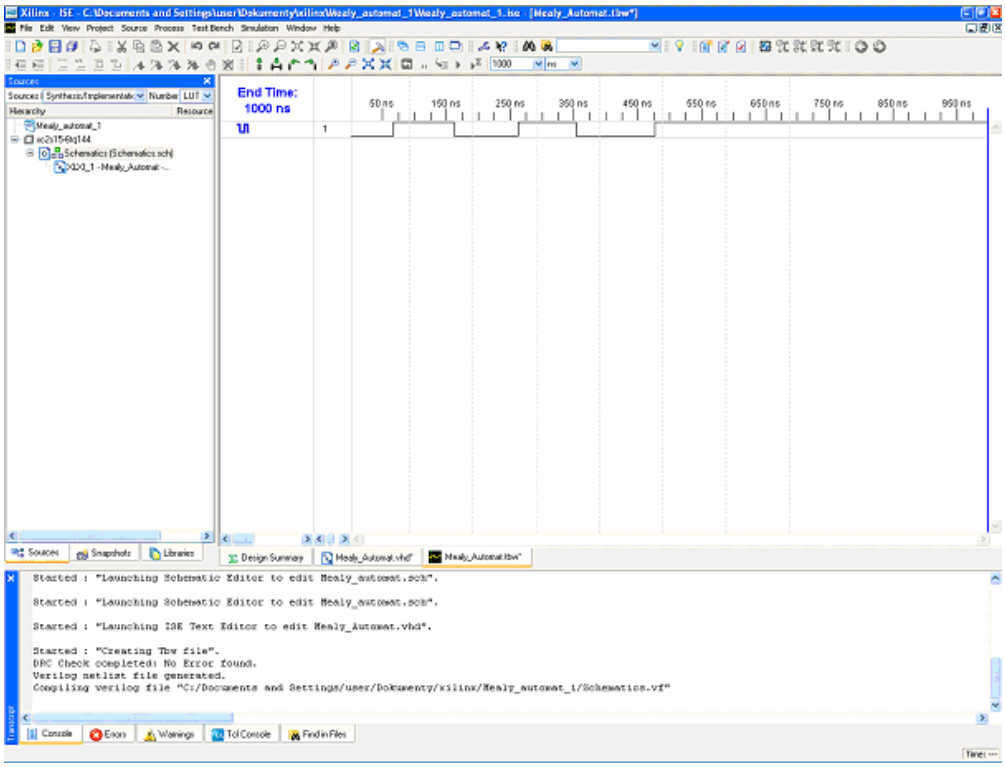
Přidáme další zdroj a zvolíme „Test bench“.



Nastavíme parametry pro odladění.



V dalším okně lze již simulovat chování např. Mealyho automatu – časové průběhy.



Poslední fází je naprogramování CPLD/FPGA hradlového pole:

1. Přiřadíme vývody programovatelného logického pole
2. Přeložíme SW
3. Naprogramujeme CPLD/FPGA

2.6 IMPLEMENTACE SW DO HRADLOVÉHO POLE (PROGRAMOVÁNÍ CPLD/FPGA)

Vytvořený zdrojový kód aplikace je přenesen z PC do hradlového pole přes rozhraní JTAG (Point Test Action Group). JTAG je port, který nepotřebuje žádné další nastavení a po připojení ihned funguje. JTAG port má označení IEEE1149.1 a firma Xilinx tento port dodává v SW pod označením Boundary Scan (metoda hraničního testu). Tímto rozhraním můžeme daný obvod i testovat. JTAG kabel je k PC připojen přes paralelní port a využívá jen některé piny (DIN, CLK,TMS IN, CTRL, PROG, DONE,Vcc a GND). [4]

Existují i další metody, jako například MultiLINX port a SPROM paměť. Tyto metody mají ovšem nevýhodu v tom, že se musí nastavovat MODE zařízení a u CPLD zařízení navíc tyto metody nelze použít. [4]

3. VZOROVÉ ÚLOHY PRO VÝUKU LOGICKÝCH SYSTÉMŮ:

Zde je přehled vybraných úloh ve VHDL: [5]

Moorův stavový automat

Mealyho stavový automat

Generátor průběhů

„Chytrý“ generátor průběhů

Vzdálenost definující sčítačka a odčítačka

Čítač nul (kombinační verze)

Čítač nul (sekvenční verze)

Nápojový automat (statická verze)

Nápojový automat (verze počítání pěticentů)

Sčítačka předvídaného přenosu

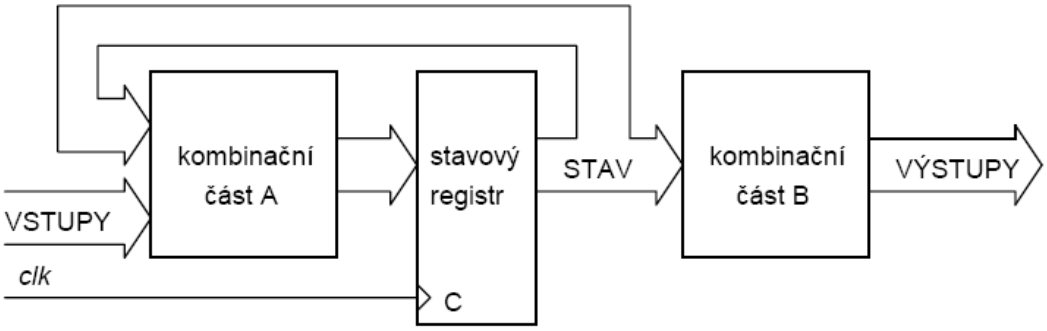
Sériovo – Paralelní převodník (počítající bity)

Sériovo – Paralelní převodník (posouvání bitů)

Programovatelné logické pole (PLA)

3.1 Moorův stavový automat:

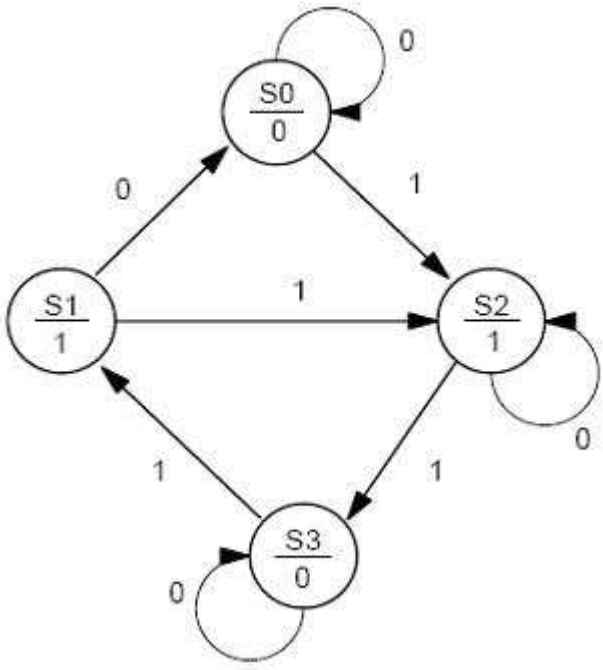
Obecně výstupní signály závisí jen na stavu automatu. Moorův automat má více stavů než Mealyho automat.



Obr. 2: *Obecné blokové schéma Moorova automatu.* [1]

3.1.1 Stavový diagram:

Na obr. 1 je stavový diagram jednoduchého Moorova automatu, který má jeden vstup X, čtyři vnitřní stavy S0 až S3 a jeden výstup Z.



Obr. 3: *Stavový diagram Moorova stavového automatu.* [5]

současný stav	následující stav		Výstup Z
	X=0	X=1	
S0	S0	S2	0
S1	S0	S2	1
S2	S0	S3	1
S3	S0	S1	0

Tab. 1: *Tabulka stavů a výstupu Z.*

3.1.2 VHDL:

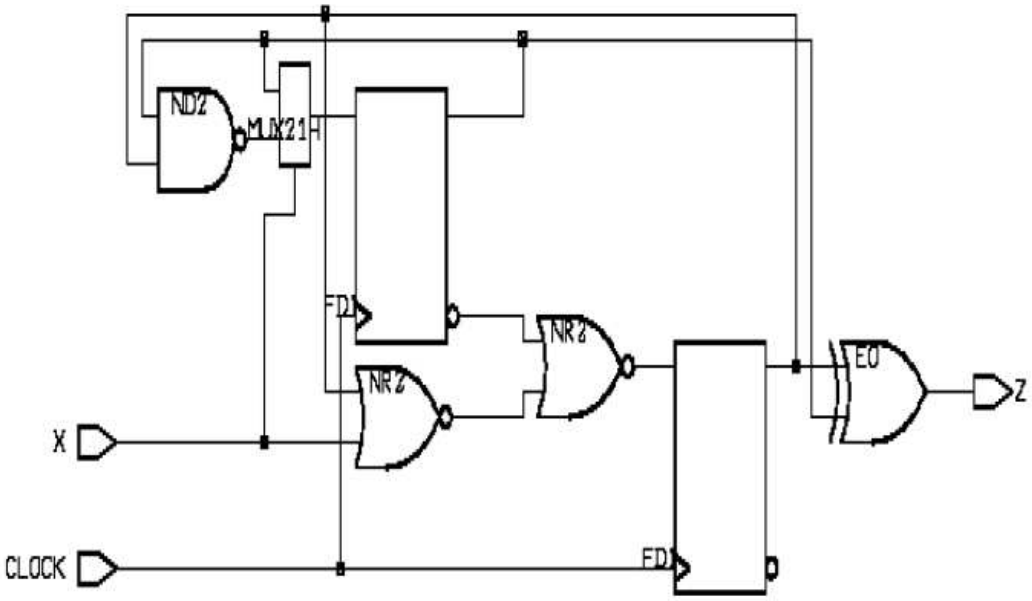
Program je vytvořeno pomocí stavového diagramu (viz. obr. 3) a také podle tabulky stavů (viz. tab. 5). Program se skládá ze dvou částí. První část popisuje kombinační část automatu a druhá část popisuje synchronizaci (záznamy stavů).

```
entity MOORE is
     port(X, CLOCK: in BIT;
     Z: out BIT);
end;

architecture BEHAVIOR of MOORE is
     type STATE_TYPE is (S0, S1, S2, S3);
     signal SOUCASNY_STAV, DALSI_STAV: STATE_TYPE;
begin


-- kombinační část

COMBIN: process(SOUCASNY_STAV, X)
begin
     case SOUCASNY_STAV is
          when S0 =>
               Z <= '0';
               if X = '0' then
                    DALSI_STAV <= S0;
               else
                    DALSI_STAV <= S2;
               end if;
          when S1 =>
               Z <= '1';
               if X = '0' then
```

```
                    DALSI_STAV <= S0;
               else
                    DALSI_STAV <= S2;
               end if;
          when S2 =>
               Z <= '1';
               if X = '0' then
                    DALSI_STAV <= S2;
               else
                    DALSI_STAV <= S3;
               end if;
          when S3 =>
               Z <= '0';
               if X = '0' then
                    DALSI_STAV <= S3;
               else
                    DALSI_STAV <= S1;
               end if;
     end case;
end process;


-- synchronizovaná (souběžná) část

SYNCH: process
     begin
          wait until CLOCK'event and CLOCK = '1';
          SOUCASNY_STAV <= DALSI_STAV;
     end process;
end BEHAVIOR;
```

Př. 1: *Příklad Moorova stavového automatu ve VHDL kódu.*

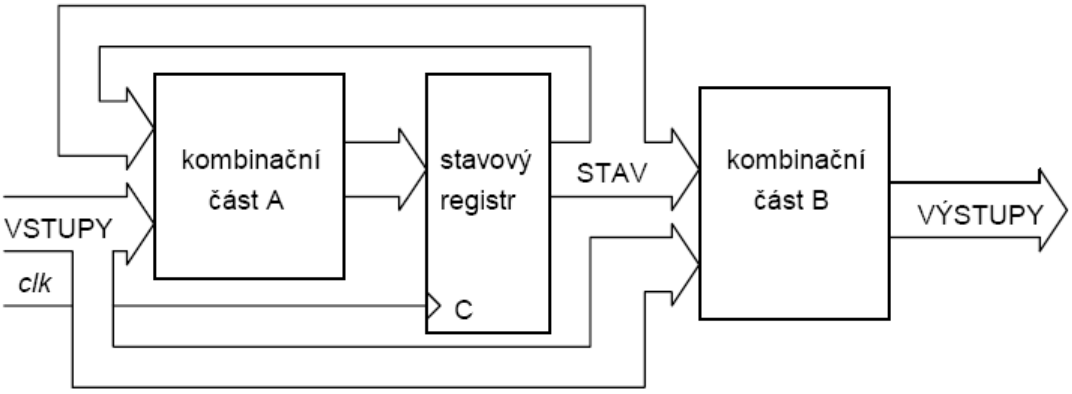
3.1.3 Schéma:



Obr. 4: *Schéma zapojení Moorova automatu.* [5]

3.2 Mealyho stavový automat:

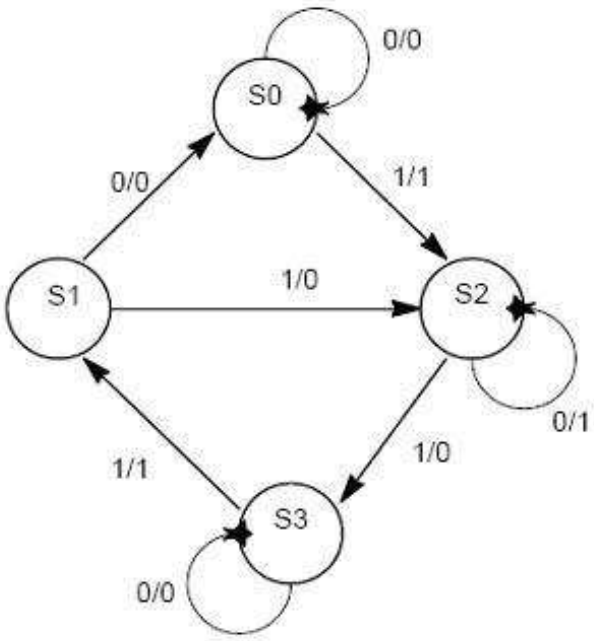
Obecně výstupní signály u tohoto automatu závisí i na vstupních signálech.



Obr. 5: *Obecné blokové schéma Mealyho automatu.* [1]

3.2.1 Stavový diagram:

Na obr. 6 je stavový diagram jednoduchého Mealyho automatu s jedním vstupem a jedním výstupem.



Obr. 6: *Stavový diagram Mealyho stavového automatu.* [5]

současný stav	následující stav		Výstup Z	
	X=0	X=1	X=0	X=1
S0	S0	S2	0	1
S1	S0	S2	0	0
S2	S2	S3	1	0
S3	S3	S1	0	1

Tab. 2: *Tabulka stavů a výstupu Z.*

3.2.2 VHDL:

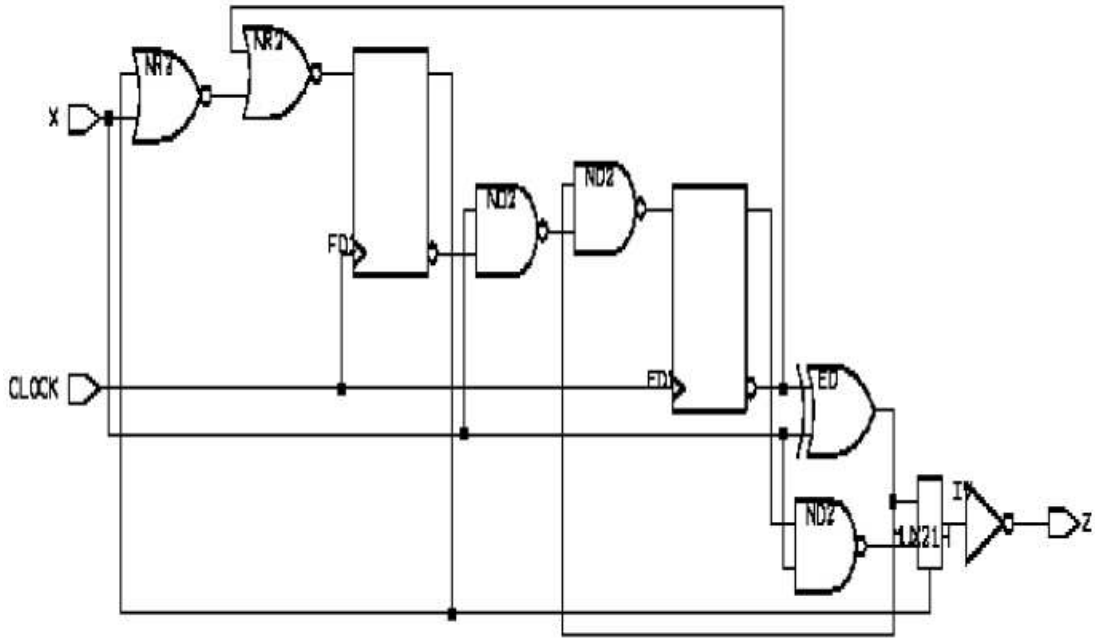
Program jak je opět popsán ve dvou částech, stejně jako u předchozího Moorova automatu.

```
entity MEALY is
     port(X, CLOCK: in BIT;
          Z: out BIT);
end;

architecture BEHAVIOR of MEALY is
     type STATE_TYPE is (S0, S1, S2, S3);
     signal SOUCASNY_STAV, DALSI_STAV: STATE_TYPE;
begin


-- kombinační část

COMBIN: process(SOUCASNY_STAV, X)
begin
     case SOUCASNY_STAV is
          when S0 =>
               if X = '0' then
                 Z <= '0';
                 DALSI_STAV <= S0;
               else
                 Z <= '1';
                 DALSI_STAV <= S2;
               end if;
          when S1 =>
               if X = '0' then
                 Z <= '0';
```

```
                DALSI_STAV <= S0;
            else
                Z <= '0';
                DALSI_STAV <= S2;
            end if;
        when S2 =>
            if X = '0' then
                Z <= '1';
                DALSI_STAV <= S2;
            else
                Z <= '0';
                DALSI_STAV <= S3;
            end if;
        when S3 =>
            if X = '0' then
                Z <= '0';
                DALSI_STAV <= S3;
            else
                Z <= '1';
                DALSI_STAV <= S1;
            end if;
        end case;
    end process;


-- synchronizovaná (souběžná) část

SYNCH: process
    begin
        wait until CLOCK'event and CLOCK = '1';
        SOUCASNY_STAV <= DALSI_STAV;
    end process;
end BEHAVIOR;
```

Př. 2: *Příklad Mealyho stavového automatu ve VHDL kódu.*

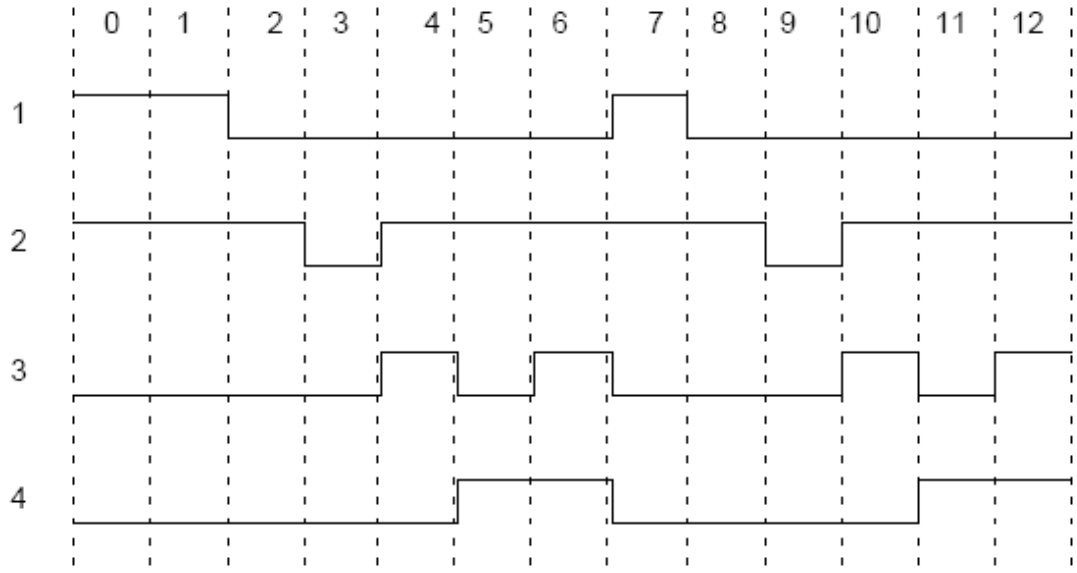
3.2.3 Schéma:



Obr.7: *Schéma zapojení Mealyho stavového automatu.* [5]

3.3 Generátor průběhů:

Tento příklad ukazuje, jak použít paměť ROM k zavedení generátoru průběhů. Uvažme, že chceme vytvořit na výstupu křivky (viz. obr. 8). Nejprve si musíme zajistit dostatečně velikou paměť ROM k získání výstupních signálů (4 bity), a dostatečně „hlubokou“, aby zachovaly veškeré postupné kroky (od 0 do 12 až k výsledným 13). Dále musíme definovat ROM, tak že každý postupný krok je zaznamenán vstupem v ROM. Nakonec musíme vytvořit čítač, který bude cyklicky načítat časové kroky (ROM adresy), které budou vytvářet křivky v každém časovém kroku. [5]



***Obr. 8**: Příklad tvorby průběhů.* [5]

3.3.1 VHDL:

Program zahrnuje ROM, čítač a jednoduché obnovovací procesy.

```
package ROMS is
     -- ROM 4x13 – obsahuje průběhy
     constant ROM_WIDTH: INTEGER := 4;
     subtype ROM_WORD is BIT_VECTOR (1 to ROM_WIDTH);
     subtype ROM_RANGE is INTEGER range 0 to 12;
     type ROM_TABLE is array (0 to 12) of ROM_WORD;
     constant ROM: ROM_TABLE := ROM_TABLE'(
     "1100",   -- krok 0
     "1100",   -- krok 1
     "0100",   -- krok 2
     "0000",   -- krok 3
     "0110",   -- krok 4
     "0101",   -- krok 5
     "0111",   -- krok 6
     "1100",   -- krok 7
     "0100",   -- krok 8
     "0000",   -- krok 9
     "0110",   -- krok 10
     "0101",   -- krok 11
     "0111");  -- krok 12
end ROMS;

use work.ROMS.all;
entity WAVEFORM is
     port(CLOCK: in BIT;
          RESET: in BOOLEAN;
          WAVES: out ROM_WORD);
end;

architecture BEHAVIOR of WAVEFORM is
     signal STEP: ROM_RANGE;
begin

     TIMESTEP_COUNTER: process – čitač kroku
     begin
          wait until CLOCK'event and CLOCK = '1';
          if RESET then              -- detekce resetu
               STEP <= ROM_RANGE'low;       -- restart
          elsif STEP = ROM_RANGE'high then  -- konec?
               STEP <= ROM_RANGE'high;
            -- STEP <= ROM_RANGE'low; --pokračování
                                         průběhu
```

```
            else
                 STEP <= STEP + 1;  -- další krok
            end if;
      end process TIMESTEP_COUNTER;
      WAVES <= ROM(STEP);
end BEHAVIOR;
```

Př. 3: *Příklad generátoru průběhů ve VHDL kódu.* [5]

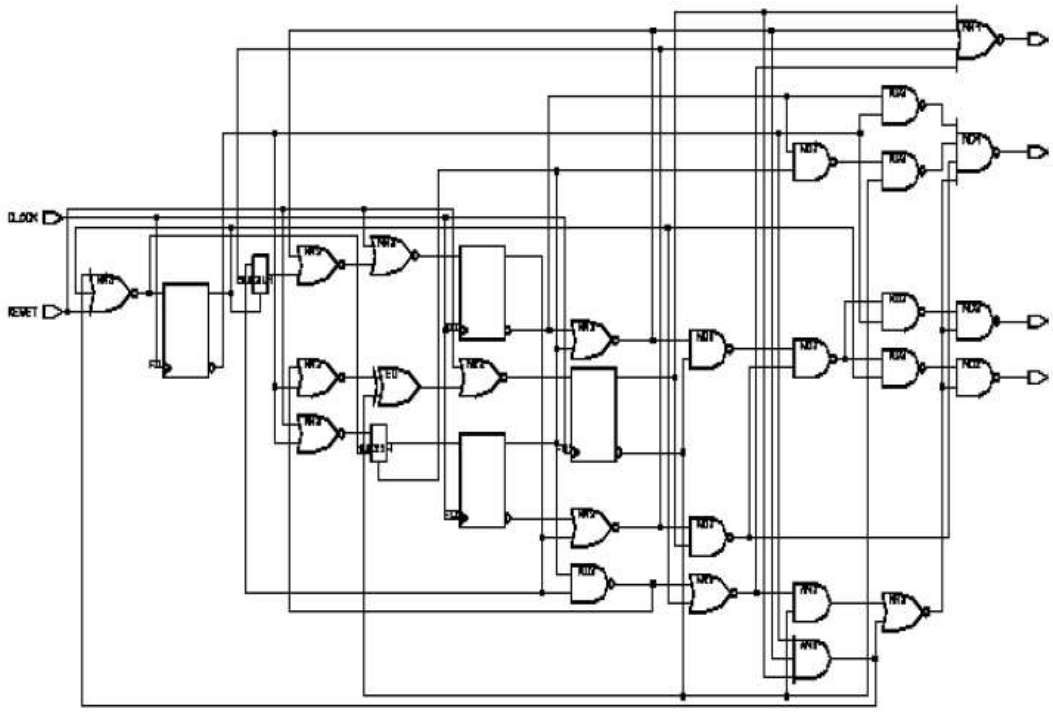
3.3.2 Schéma:

Pokud čítač `STEP` dosáhne na konec v paměti ROM, tak se čítač zastaví, vygeneruje poslední hodnotu a potom čeká na obnovení. Chceme-li, aby se sekvence automaticky zopakovala, vyjměte formuli :

`STEP <= ROM_RANGE´high` [5]

A místo toho použijte následující příkaz (vysvětleno v příkladě):

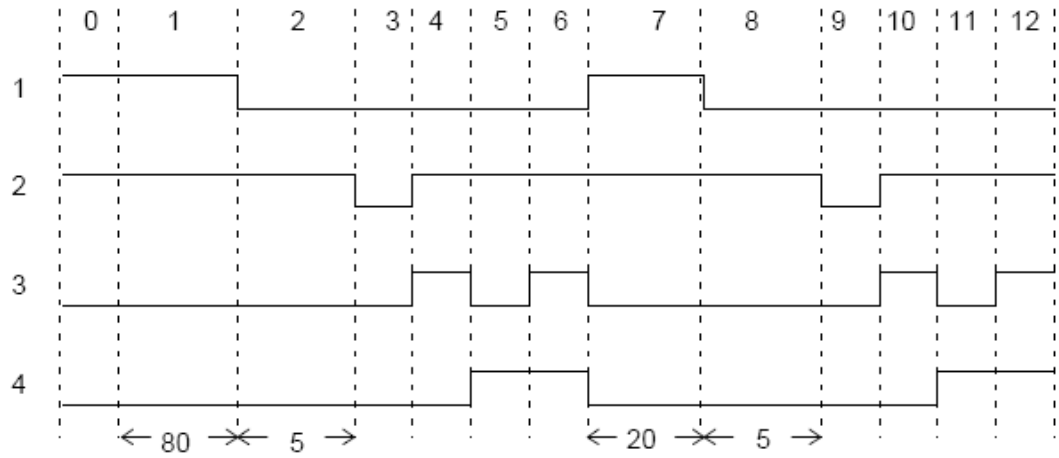
`STEP<= ROM_RANGE´low` [5]



Obr .9: *Schéma zapojení generátoru průběhů.* [5]

3.4 „Chytrý“ generátor průběhů:

Tento generátor křivek je vylepšenou verzí předchozího příkladu. Tento „chytrý“ generátor průběhů je schopný udržet průběh na jakémkoliv bodě po několik hodinových cyklů. Na obr. 10 je znázorněna tvorba průběhu, kde je zachyceno několik časových momentů v násobných hodinových cyklech. [5]



Obr. 10: *Tvorba průběhu pro příklad „chytrého“generátoru průběhů.* [5]

3.4.1 VHDL:

Program je podobný jako v předchozím příkladu. Je zde ale použita `D_ROM`, z důvodu uchování délky každého časového kroku. Hodnota `1` určuje, že odpovídající časový krok by měl být dlouhý jeden časový cyklus, hodnota 80 určuje, že časový krok by měl být dlouhý 80 cyklů. Další zlepšení vzhledem k předcházejícímu příkladu je zpoždění čítače, který spočítá hodinové cykly mezi kroky.

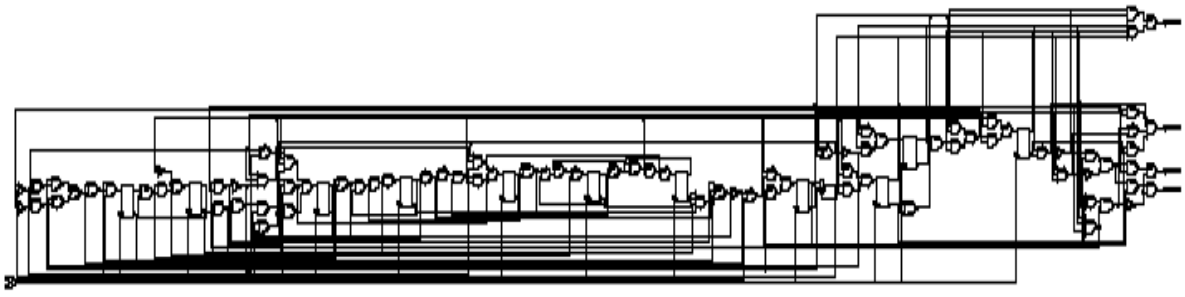
Při projektování tohoto příkladu, vybraný přiřazený signál (*a selected signal assignment*), určuje hodnotu `NEXT_STEP` čítač. [5]

```
package ROMS is
-- W_ROM 4x13 – obsahuje průběhy
     constant W_ROM_WIDTH: INTEGER := 4;
     subtype W_ROM_WORD is BIT_VECTOR (1 to W_ROM_WIDTH);
     subtype W_ROM_RANGE is INTEGER range 0 to 12;
     type W_ROM_TABLE is array (0 to 12) of W_ROM_WORD;
     constant W_ROM: W_ROM_TABLE := W_ROM_TABLE'(
          "1100",   -- krok 0
          "1100",   -- krok 1
          "0100",   -- krok 2
          "0000",   -- krok 3
          "0110",   -- krok 4
          "0101",   -- krok 5
          "0111",   -- krok 6
          "1100",   -- krok 7
          "0100",   -- krok 8
          "0000",   -- krok 9
          "0110",   -- krok 10
          "0101",   -- krok 11
          "0111");  -- krok 12

-- paměť D_ROM 7x13 obsahuje zpoždění
     subtype D_ROM_WORD is INTEGER range 0 to 100;
     subtype D_ROM_RANGE is INTEGER range 0 to 12;
     type D_ROM_TABLE is array (0 to 12) of D_ROM_WORD;
     constant D_ROM: D_ROM_TABLE := D_ROM_TABLE'(
          1,80,5,1,1,1,1,20,5,1,1,1,1);
end ROMS;

use work.ROMS.all;
entity WAVEFORM is
     port(CLOCK: in BIT;
          RESET: in BOOLEAN;
          WAVES: out W_ROM_WORD);
end;

architecture BEHAVIOR of WAVEFORM is
     signal STEP, NEXT_STEP: W_ROM_RANGE;
     signal DELAY: D_ROM_WORD;
begin

-- určení hodnoty v dalším kroku

     NEXT_STEP <= W_ROM_RANGE'high when
          STEP = W_ROM_RANGE'high
     else
          STEP + 1;
```

```
TIMESTEP_COUNTER: process
begin
     wait until CLOCK'event and CLOCK = '1';
     if RESET then              -- detekce resetu
          STEP <= 0;            -- Restart průběhu
     elsif DELAY = 1 then
          STEP <= NEXT_STEP;    -- další krok
     else
          null;                 -- čekání na zpoždění
     end if;
end process;

-- zpoždění mezi dvěma kroky

DELAY_COUNTER: process
begin
wait until CLOCK'event and CLOCK = '1';
if RESET then                   -- detekce resetu
DELAY <= D_ROM(0);              -- Restart
elsif DELAY = 1 then
DELAY <= D_ROM(NEXT_STEP);
else
DELAY <= DELAY - 1;
end if;
end process;
WAVES <= W_ROM(STEP);           -- výstupní hodnota průběhu
     end BEHAVIOR;
```

Př. 4: *Příklad „chytrého" generátoru křivek ve VHDL kódu.* [5]

3.4.2 Schéma:



Obr. 11: *Schéma zapojení „chytrého" generátoru průběhů.* [5]

3.5 Vzdálenost definující sčítačka a odčítačka:

Jazyk VHDL umožňuje vytvářet funkce pro použití pole operandů jakékoliv velikosti. Tento příklad ukazuje obvod sčítačky a odčítačky který, pokud je nutné, je přizpůsoben velikosti jejích operandů. [5]

3.5.1 VHDL:

Sčítačku je nadefinována pro dvě neomezená bitová pole (typu `BIT_VECTOR`) v jednom balíku, nazvaném `MATH`. Pokud je použito neomezené bitové pole jako subprogram, platná omezení pole jsou brána z aktuálně platných hodnot v subprogramu. [5]

```
package MATH is
     function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
          return BIT_VECTOR;

end MATH;

package body MATH is
     function ADD_SUB(L, R: BIT_VECTOR; ADD: BOOLEAN)
               return BIT_VECTOR is
          variable CARRY: BIT;
          variable A, B, SUM:
               BIT_VECTOR(L'length-1 downto 0);
     begin
          if ADD then
               -- operace "add"
               A := L;
               B := R;
               CARRY := '0';
          else
               -- operace "subtract"

               A := L;
               B := not R;
               CARRY := '1';
          end if;
          for i in 0 to A'left loop
```

```
                SUM(i) := A(i) xor B(i) xor CARRY;
                CARRY := (A(i) and B(i)) or
                         (A(i) and CARRY) or
                         (CARRY and B(i));
          end loop;
          return SUM; -- výsledek
     end;
end MATH;
```

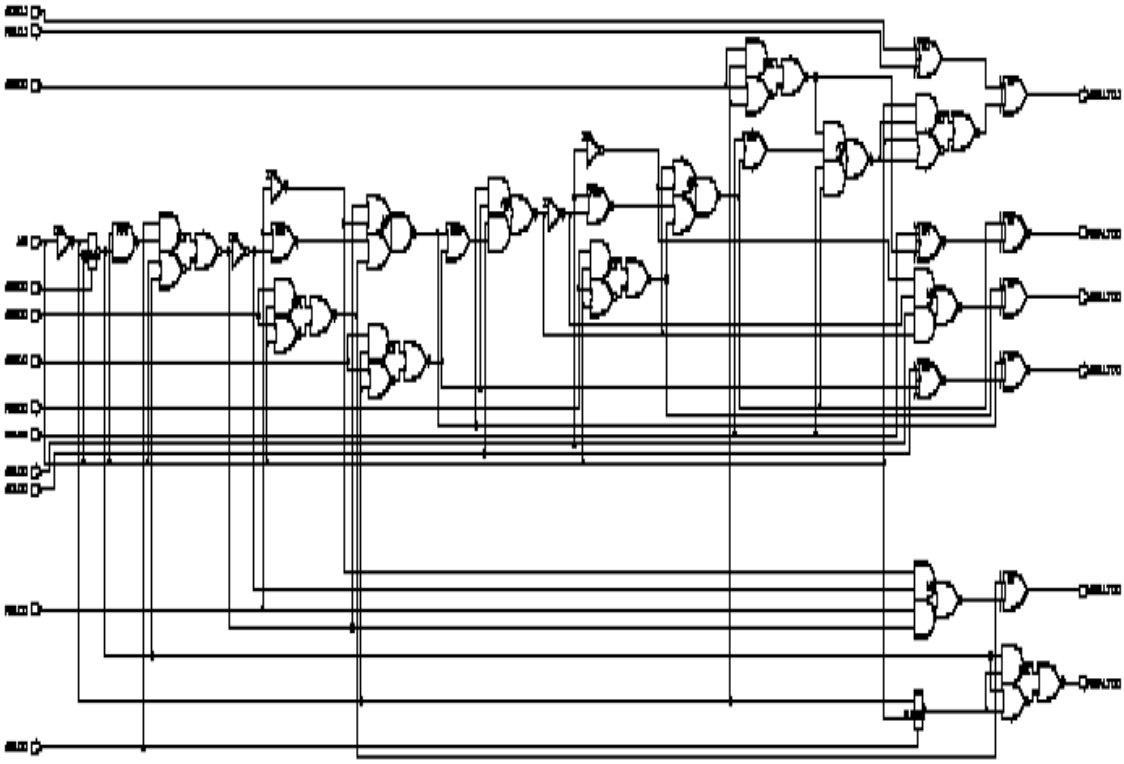
Př. 5: *Příklad balíku* `MATH` *ve VHDL kódu* [5]

Další příklad ukazuje jak použít sčítačku nadefinovanou v balíku `MATH`. V tomto příkladu jsou vektorové argumenty deklarovány k funkcím `ARG1` a `ARG2` jako `BIT_VECTOR` (1 až 6). Tato definice způsobí, že `ADD_SUB` bude fungovat se 6 bitovým polem. V příkazu `ADD_SUB` jsou deklarovány dvě dočasné proměnné `A` a `B`. Tyto proměnné jsou nadefinovány tak, aby byly stejně dlouhé jako `L` (a nutně, `R`), ale mají svůj index omezení normalizován na `L´length-1 downto 0`. Po té, co jsou tyto argumenty normalizované, může se vytvořit přenosový sčítač použitím cyklu `for`. Všimněme si, že neexistují žádné jednoznačné odkazy ve funkci `ADD_SUB` k tomu, jak dlouhé má být pole. Místo toho jsou použity znaky VHDL pole `´left` a `´lenght.` Tyto znaky umožňují této funkci pracovat při jakékoli délce pole. [5]

```
use work.MATH.all;

entity EXAMPLE is
     port(ARG1, ARG2: in BIT_VECTOR(1 to 6);
          ADD: in BOOLEAN;
          RESULT : out BIT_VECTOR(1 to 6));
end EXAMPLE;

architecture BEHAVIOR of EXAMPLE is
begin
     RESULT <= ADD_SUB(ARG1, ARG2, ADD);
end BEHAVIOR;
```

Př. 6: *Příklad zavedení 6 bitové sčítačky ve VHDL kódu* [5]

3.5.2 Schéma:



Obr. 12: *Schéma zapojení vzdálenosti definující sítačka a odčítačka*. [5]

3.6 Čítač nul (kombinační verze):

Tento příklad ukazuje konstrukční problém, kde je dána hodnota o velikosti 8 bitů, a tento obvod určuje dvě věci :

- není dána více než jedna sekvence, kde 0 je v hodnotách zadána
- číslo nul v sekvenci (pokud jsou nějaká). Tento výpočet musí být proveden během jednoho cyklu taktu.

Obvod vytváří dva výstupy. A to počet nalezených nul (`zeros`), a indikaci chyb (`error`). Legální vložené hodnoty mohou mít nejvýše jednu návaznou sérii nul. Hodnota, která se skládá zcela z jedniček je definována jako legální. Pokud je hodnota ilegální, čítač nul se vynuluje. Například hodnota `00000000` je legální a má osm nul. Hodnota `11000111` je legální a má tři nuly a hodnota `00111100` je nelegální. Následující příklad ukazuje popis pro obvod. Skládá se z jednoduchého

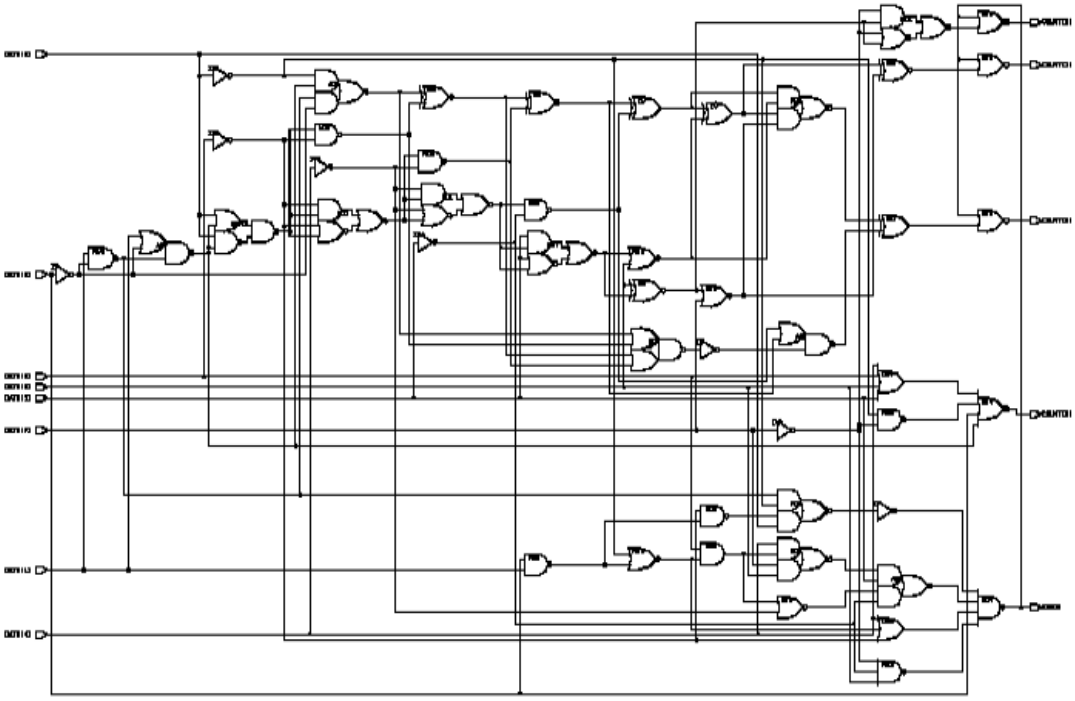
procesu se smyčkou `for`, která se zopakuje přes každý bit v dané hodnotě. Při každém opakování dočasná proměnné `INTEGER` (`TEMP_COUNT`) počítá počet nul, s nimiž se setkala. Dvě dočasné proměnné `BOOLEAN` (`SEEN_ZERO` a `SEEN_TRAILING`), ač původně chybné (`FALSE`), jsou znovu nastaveny na hodnotu pravdivou (`TRUE`). Pokud je nalezen začátek a konec první sekvence nul. Pokud je nalezena nula po ukončení první sekvence nul (po `SEEN_TRAILING` je `TRUE`), čítač nul je vymazán na 0, a `ERROR` je nastaven na `TRUE` a smyčka `for` je ukončena. Tento příklad představuje kombinační (paralelní) přístup k čítání nul. [5]

3.6.1 VHDL:

```
entity COUNT_COMB_VHDL is
     port(DATA: in BIT_VECTOR(7 downto 0);
          COUNT: out INTEGER range 0 to 8;
          ERROR: out BOOLEAN);
end;

architecture BEHAVIOR of COUNT_COMB_VHDL is
begin
     process(DATA)
          variable TEMP_COUNT : INTEGER range 0 to 8;
          variable SEEN_ZERO, SEEN_TRAILING : BOOLEAN;
     begin
          ERROR <= FALSE;
          SEEN_ZERO := FALSE;
          SEEN_TRAILING := FALSE;
          TEMP_COUNT := 0;
          for I in 0 to 7 loop
               if (SEEN_TRAILING and DATA(I) = '0') then
                    TEMP_COUNT := 0;
                    ERROR <= TRUE;
                    exit;
               elsif (SEEN_ZERO and DATA(I) = '1') then
                    SEEN_TRAILING := TRUE;
               elsif (DATA(I) = '0') then
                    SEEN_ZERO := TRUE;
                    TEMP_COUNT := TEMP_COUNT + 1;
               end if;
          end loop;
          COUNT <= TEMP_COUNT;
     end process;
end BEHAVIOR;
```

Př. 7: *Příklad čítače nul (kombinační verze) ve VHDL kódu.* [5]

3.6.2 Schéma:



Obr. 13:. *Schéma zapojení čítače nul (kombinační verze).* [5]

3.7 Čítač nul (sekvenční verze):

Další příklad představuje sekvenční (sériový) přístup neboli sekvenční taktovou variantu oproti předchozímu přikladu.. Obvod nyní přijímá 8 bitovou hodnotu sériově, jeden bit za jeden takt. Vstupy jsou DATA a CLK. A Další dva vstupy jsou RESET (vymaže obvod) a READ (způsobí, že obvod začne znovu přijímat bitová data). Výstupy obvodu jsou tři, IS_LEGAL (je TRUE , pokud měla data legální hodnotu), COUNT_READY (je TRUE u prvního legálního bitu nebo pokud všech 8 bitů bylo zpracováno) a COUNT - čísla nul (pokud IS_LEGAL je TRUE). Místo výstupního COUNT je deklarován modem BUFFER, takže může být během procesu čten. Výstupy OUT mohou být pouze zapisovány, ne čteny. [5]

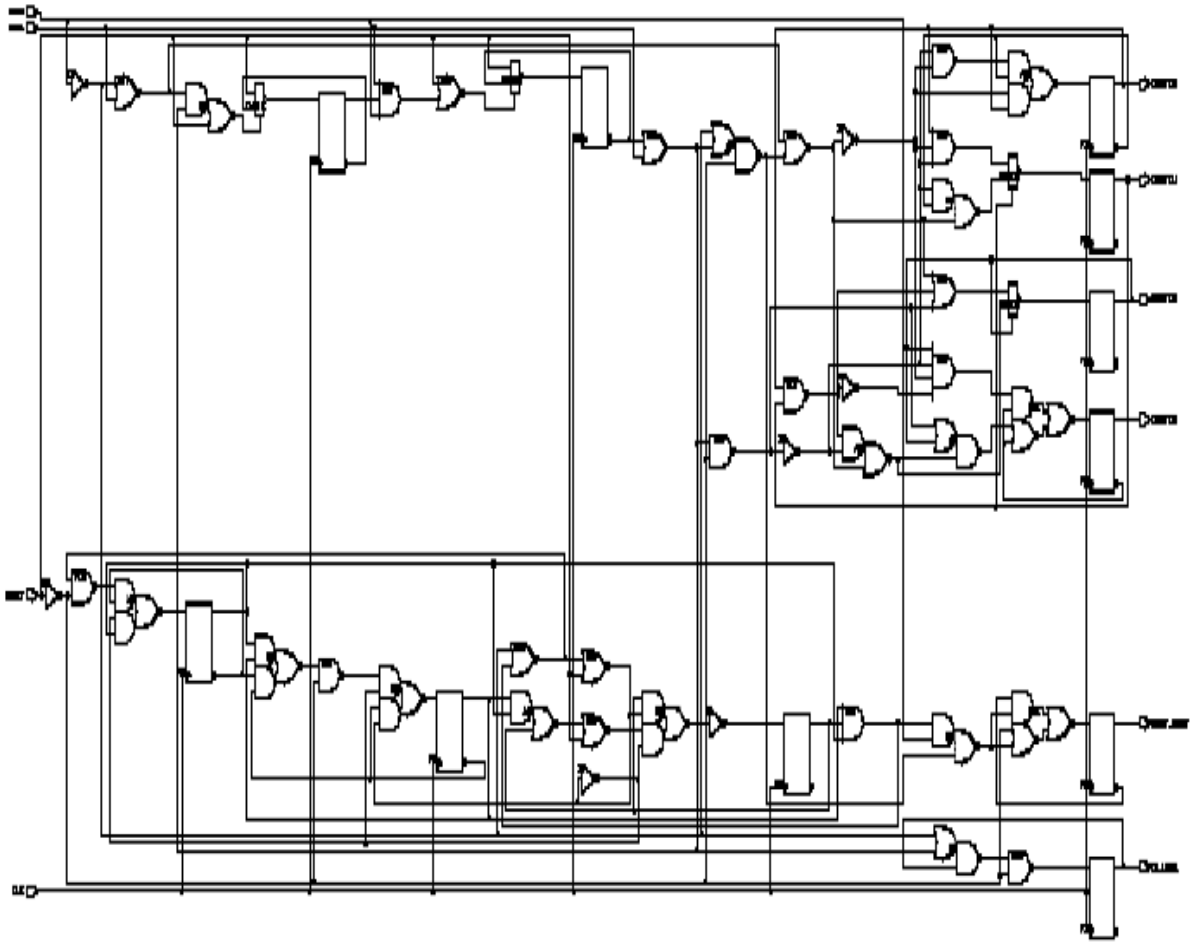
3.7.1 VHDL:

```
entity COUNT_SEQ_VHDL is
     port(DATA, CLK: in BIT;
          RESET, READ: in BOOLEAN;
          COUNT: buffer INTEGER range 0 to 8;
          IS_LEGAL: out BOOLEAN;
          COUNT_READY: out BOOLEAN);
     end;


architecture BEHAVIOR of COUNT_SEQ_VHDL is
begin
     process
          variable SEEN_ZERO, SEEN_TRAILING: BOOLEAN;
          variable BITS_SEEN: INTEGER range 0 to 7;
     begin
          wait until CLK'event and CLK = '1';

          if(RESET) then
               COUNT_READY <= FALSE;
               IS_LEGAL <= TRUE;
               SEEN_ZERO := FALSE;
               SEEN_TRAILING := FALSE;
               COUNT <= 0;
               BITS_SEEN := 0;
          else
               if (READ) then
                 if (SEEN_TRAILING and DATA = '0') then
                    IS_LEGAL <= FALSE;
                    COUNT <= 0;
                    COUNT_READY <= TRUE;
                 elsif (SEEN_ZERO and DATA = '1') then
                    SEEN_TRAILING := TRUE;
                 elsif (DATA = '0') then
                    SEEN_ZERO := TRUE;
                    COUNT <= COUNT + 1;
                 end if;

                 if (BITS_SEEN = 7) then
                    COUNT_READY <= TRUE;
                 else
                    BITS_SEEN := BITS_SEEN + 1;
                 end if;

               end if;
```

```
        end if;
    end process;
end BEHAVIOR;
```

Př. 8: *Příklad čítače nul (sekvenční verze) ve VHDL kódu.* [5]

3.7.2 Schéma:



Obr. 14:. *Schéma zapojení čítače nul (sekvenční verze).* [5]

3.8 Nápojový automat (statická verze):

Zde je uveden příklad pro kontrolní jednotkou pro prodejní nápojový automat. Obvod čte signály z místa pro vházení mince a vysílá signály do jednotky vyměňující informace a jednotky vydávající nápoje. Tento příklad předpokládá, že je vydáván pouze jeden druh nealko nápoje. Jde o taktový model se vstupními signály `CLK` a `RESET`. Cena nápoje je 35 centů. Vstupní signály z místa pro vhazování mincí jsou `NICKEL_IN` (vložen 5 cent), `DIME_IN` (vložen 10 cent) a `QUARTER_IN` (vložen 25 cent). Výstupní signály pro jednotku vracení mincí jsou `NICKEL_OUT` a `DIME_OUT`. Výstupní signál pro jednotku výdeje nápojů jsou `DISPENSE` (vydej nápoj). [5]

3.8.1 VHDL:

```
library synopsys; use synopsys.attributes.all;

entity DRINK_STATE_VHDL is
     port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
          CLK: BIT;
          NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end;

architecture BEHAVIOR of DRINK_STATE_VHDL is
     type STATE_TYPE is (IDLE, FIVE, TEN, FIFTEEN,
               TWENTY, TWENTY_FIVE, THIRTY, OWE_DIME);
     signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
     attribute STATE_VECTOR : STRING;
     attribute STATE_VECTOR of BEHAVIOR : architecture is
                         ”CURRENT_STATE”;


attribute sync_sync_reset of reset : signal is ”true”;
begin
     process(NICKEL_IN, DIME_IN, QUARTER_IN,
             CURRENT_STATE, RESET, CLK)
```

```
begin
     -- počáteční přořazení
     NEXT_STATE <= CURRENT_STATE;
     NICKEL_OUT <= FALSE;
     DIME_OUT <= FALSE;
     DISPENSE <= FALSE;

     -- synchronní reset
     if(RESET) then
          NEXT_STATE <= IDLE;
     Else

     -- přenos stavů a výstupní logika
          case CURRENT_STATE is
               when IDLE =>
                    if(NICKEL_IN) then
                         NEXT_STATE <= FIVE;
                    elsif(DIME_IN) then
                         NEXT_STATE <= TEN;
                    elsif(QUARTER_IN) then
                         NEXT_STATE <= TWENTY_FIVE;
                    end if;

               when FIVE =>
                    if(NICKEL_IN) then
                         NEXT_STATE <= TEN;
                    elsif(DIME_IN) then
                         NEXT_STATE <= FIFTEEN;
                    elsif(QUARTER_IN) then
                         NEXT_STATE <= THIRTY;
                    end if;
```

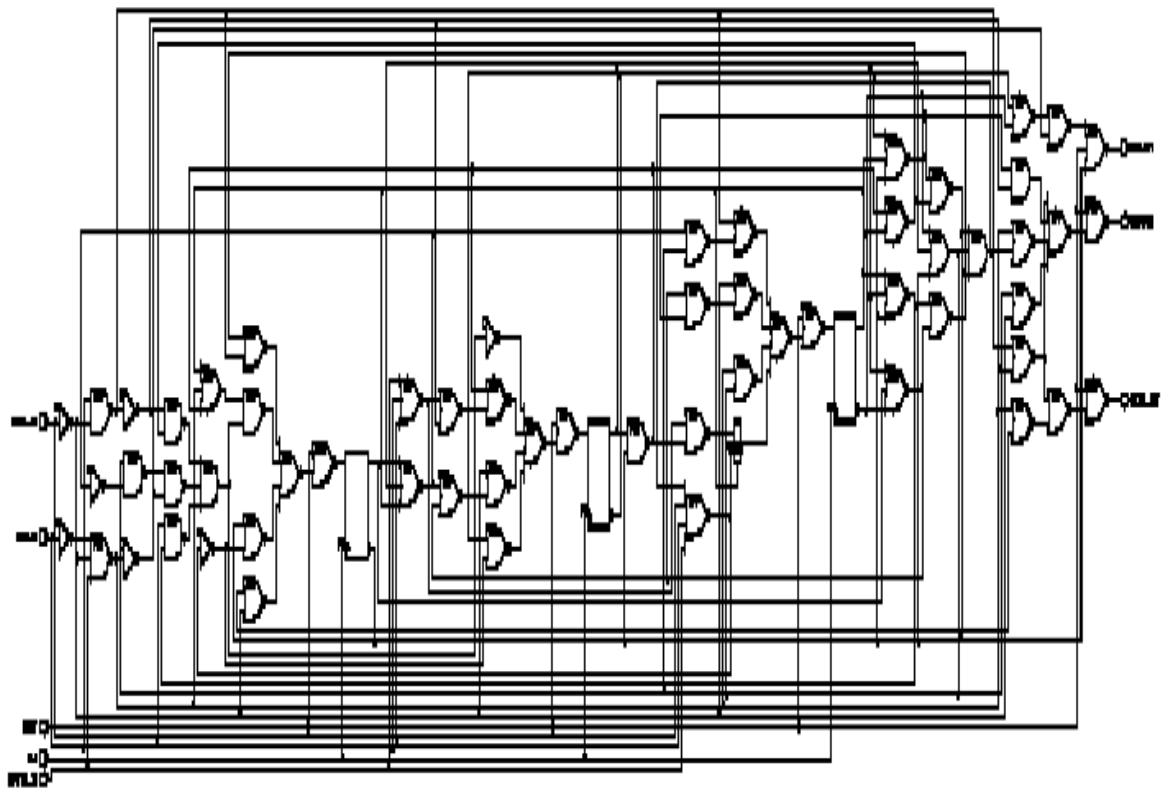
```
when TEN =>
     if(NICKEL_IN) then
          NEXT_STATE <= FIFTEEN;
     elsif(DIME_IN) then
          NEXT_STATE <= TWENTY;
     elsif(QUARTER_IN) then
          NEXT_STATE <= IDLE;
          DISPENSE <= TRUE;
     end if;
when FIFTEEN =>
     if(NICKEL_IN) then
          NEXT_STATE <= TWENTY;
     elsif(DIME_IN) then
          NEXT_STATE <= TWENTY_FIVE;
     elsif(QUARTER_IN) then
          NEXT_STATE <= IDLE;
          DISPENSE <= TRUE;
          NICKEL_OUT <= TRUE;
     end if;

when TWENTY =>
     if(NICKEL_IN) then
          NEXT_STATE <= TWENTY_FIVE;
     elsif(DIME_IN) then
          NEXT_STATE <= THIRTY;
     elsif(QUARTER_IN) then
          NEXT_STATE <= IDLE;
          DISPENSE <= TRUE;
          DIME_OUT <= TRUE;
     end if;
```

```
            when TWENTY_FIVE =>
                  if(NICKEL_IN) then
                        NEXT_STATE <= THIRTY;
                  elsif(DIME_IN) then
                        NEXT_STATE <= IDLE;
                        DISPENSE <= TRUE;
                  elsif(QUARTER_IN) then
                        NEXT_STATE <= IDLE;
                        DISPENSE <= TRUE;
                        DIME_OUT <= TRUE;
                        NICKEL_OUT <= TRUE;
                  end if;

            when THIRTY =>
                  if(NICKEL_IN) then
                        NEXT_STATE <= IDLE;
                        DISPENSE <= TRUE;
                  elsif(DIME_IN) then
                        NEXT_STATE <= IDLE;
                        DISPENSE <= TRUE;
                        NICKEL_OUT <= TRUE;
                  elsif(QUARTER_IN) then
                        NEXT_STATE <= OWE_DIME;
                        DISPENSE <= TRUE;
                        DIME_OUT <= TRUE;
                  end if;

            when OWE_DIME =>
                        NEXT_STATE <= IDLE;
                        DIME_OUT <= TRUE;
      end case;
```

```
     end if;
end process;

     process
     begin
          wait until CLK'event and CLK = '1';
          CURRENT_STATE <= NEXT_STATE;
     end process;
end BEHAVIOR;
```

Př. 9: *Příklad nápojového automatu (statická verze) ve VHDL kódu.* [5]

3.8.2 Schéma:



Obr. 15: *Schéma zapojení nápojového automatu (statická verze).* [5]

3.9 Nápojový automat (verze počítání pěticentů):

Stejně jako tomu bylo v předchozím příkladu, tak i v tomto případě je použito stejných parametrů se stejnými vstupními i výstupními signály. Rozdíl je v tom, že pro tuto verzi čítač počítá množství vhozených pěticentů. Čítač je navýšen o 1, pokud je vklad 5 cent, o 2 pokud je to 10cent, a o 5 pokud je vhozen 25 cent. [5]

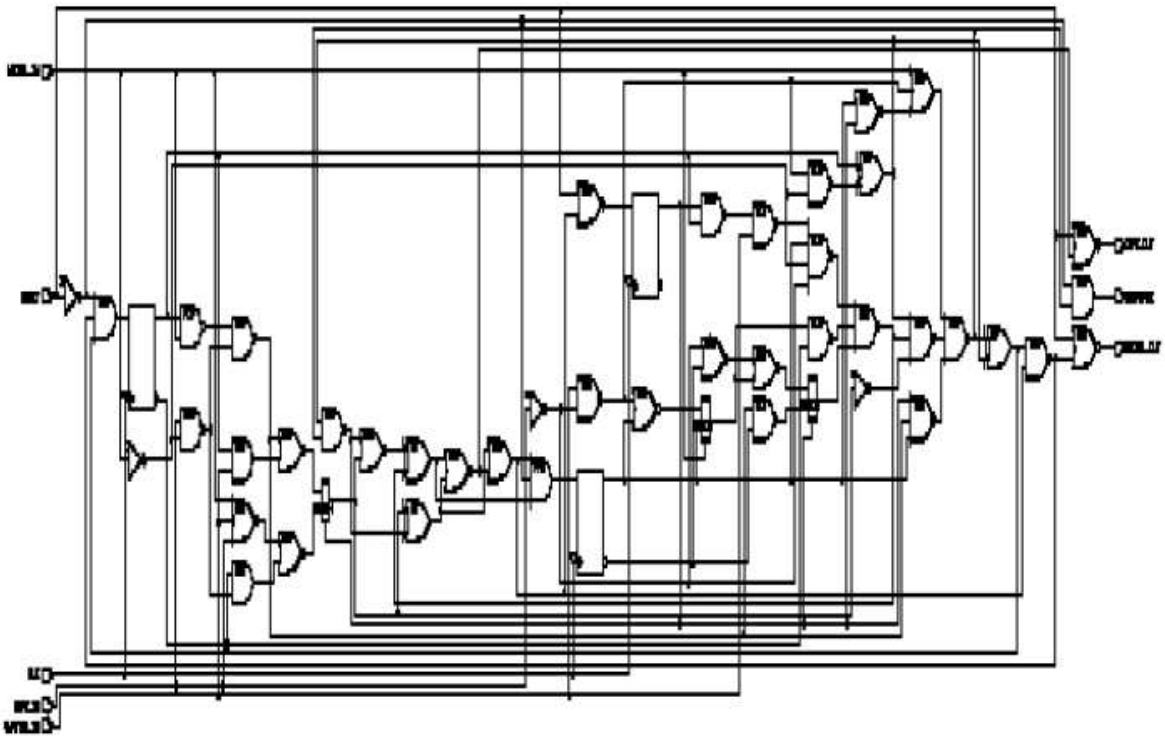
3.9.1 VHDL:

```
entity DRINK_COUNT_VHDL is
     port(NICKEL_IN, DIME_IN, QUARTER_IN, RESET: BOOLEAN;
          CLK: BIT;
          NICKEL_OUT, DIME_OUT, DISPENSE: out BOOLEAN);
end;

architecture BEHAVIOR of DRINK_COUNT_VHDL is
     signal CURRENT_NICKEL_COUNT,
          NEXT_NICKEL_COUNT: INTEGER range 0 to 7;
     signal CURRENT_RETURN_CHANGE, NEXT_RETURN_CHANGE :
     BOOLEAN;
begin

     process(NICKEL_IN, DIME_IN, QUARTER_IN, RESET, CLK,
             CURRENT_NICKEL_COUNT, CURRENT_RETURN_CHANGE)
       variable TEMP_NICKEL_COUNT: INTEGER range 0 to 12;
     begin

          -- počáteční přiřazení
          NICKEL_OUT <= FALSE;
          DIME_OUT <= FALSE;
          DISPENSE <= FALSE;
          NEXT_NICKEL_COUNT <= 0;
          NEXT_RETURN_CHANGE <= FALSE;

          -- synchronní reset
          if (not RESET) then
               TEMP_NICKEL_COUNT := CURRENT_NICKEL_COUNT;

          -- kontrola zda byly vhozeny peníze
          if (NICKEL_IN) then

              TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 1;
```

```
            elsif(DIME_IN) then
                TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 2;
            elsif(QUARTER_IN) then
                TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT + 5;
            end if;

            -- byla vhozena dostatečná suma?
            if(TEMP_NICKEL_COUNT >= 7) then
                TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT - 7;
                DISPENSE <= TRUE;
            end if;

            -- vrácení drobných
            if(TEMP_NICKEL_COUNT >= 1 or
                CURRENT_RETURN_CHANGE) then
            if(TEMP_NICKEL_COUNT >= 2) then
                DIME_OUT <= TRUE;
                TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT - 2;
                NEXT_RETURN_CHANGE <= TRUE;
            end if;
            if(TEMP_NICKEL_COUNT = 1) then
                NICKEL_OUT <= TRUE;
                TEMP_NICKEL_COUNT := TEMP_NICKEL_COUNT - 1;
            end if;
        end if;

        NEXT_NICKEL_COUNT <= TEMP_NICKEL_COUNT;
    end if;
end process;

-- zapamatuj že bylo vráceno a kolik

        process
        begin
            wait until CLK'event and CLK = '1';
            CURRENT_RETURN_CHANGE <= NEXT_RETURN_CHANGE;
            CURRENT_NICKEL_COUNT <= NEXT_NICKEL_COUNT;
        end process;
end BEHAVIOR;
```

Př. 10: *Příklad nápojového automatu (verze počítání pěticentů) ve VHDL kódu.* [5]

3.9.2 Schéma:



Obr. 16: *Schéma zapojení nápojového automatu (verze počítání pěticentů).* [5]

3.10 Sčítačka předvídaného přenosu:

U tohoto příkladu se užívá postup k sestavení 32 bitové sčítačky předvídaného přenosu. Sčítačka je sestavena rozdělením 32 bitového vstupu do osmi dílů po čtyřech . Každý z těchto osmi dílů kalkuluje, rozmnožuje se a vytváří hodnoty za použití procesu PG. Výstup P (propagate) z PG je ´1´ pro pozici bitu, pokud tato pozice rozšiřuje přenos z následné nižší pozice do následné vyšší pozice. Výstup G (generate) je ´1´ pro pozici bitu, pokud tato pozice vytváří přenos do následné vyšší pozice, bez ohledu na předcházející přenos z následné nižší pozice. Systém přečte přenos, rozmnoží a vygeneruje informaci vykalkulovanou z vložených dat. Vykalkuluje přenesenou hodnotu pro pozici každého bitu. Tento systém vytváří přídavné informace z XOR vložených dat a přenášených hodnot. [5]

Kalkulace přenášených hodnot:

Hodnoty přenosu jsou počítány stromem o třech úrovních, ve 4 bitových přenosových blocích.

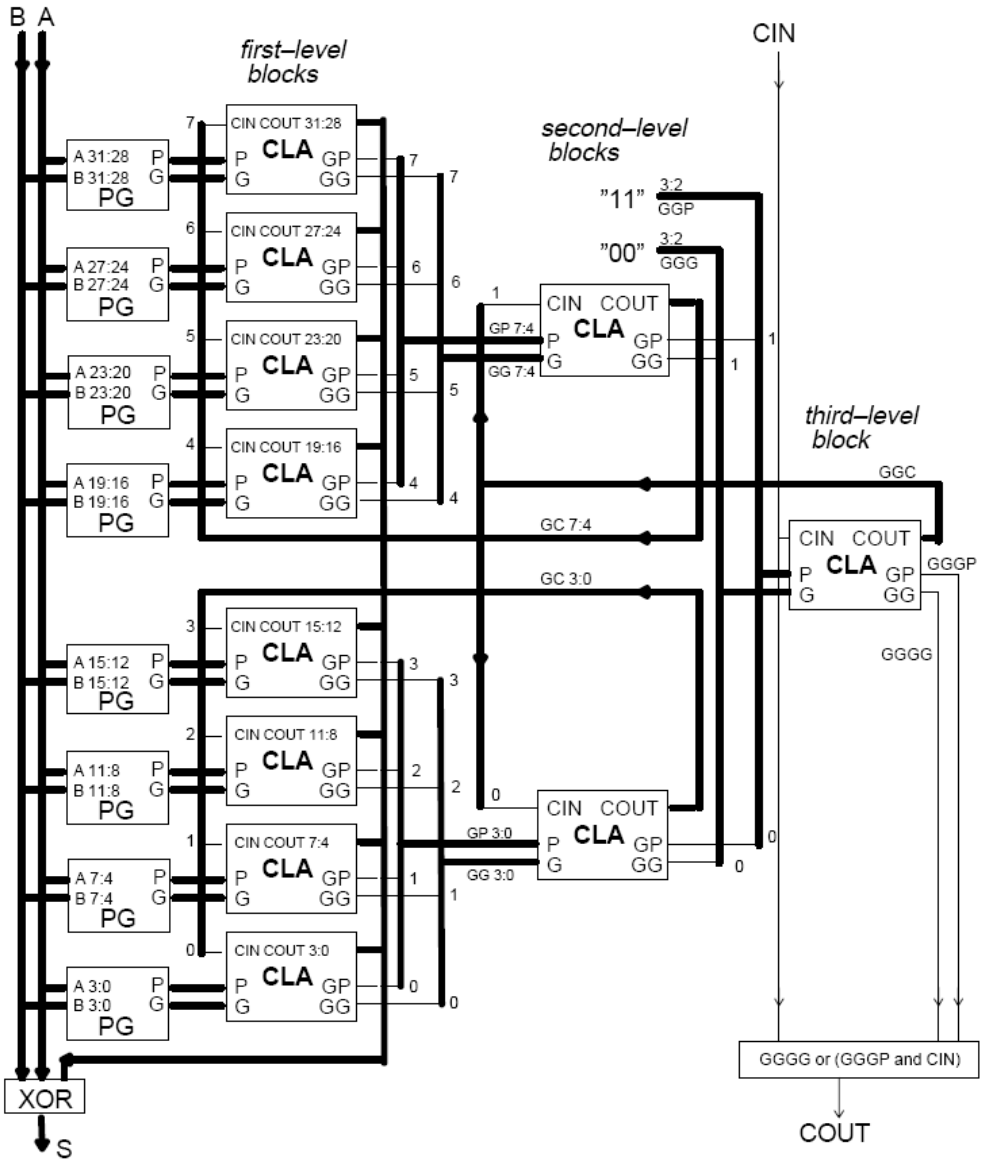
První úroveň stromu kalkuluje 32 přenášených hodnot a rozmnoženou skupinu osmi a generuje hodnoty. Každá z první úrovně skupin rozmnožování a generování hodnot říká, zda 4 bitový díl rozmnožuje a generuje přenášené hodnoty z následující nižší skupiny do následující vyšší. Na první úrovni předvídající bloky čtou přenos vykalkulovaný na druhé úrovni.

Bloky druhé úrovně čtou informace o skupině rozmnožených a generovaných informací z z prvních čtyř bloků, pak vykalkulují svoji vlastní skupinu rozmnožených a generovaných informací. Také čtou přenášené informace o vykalkulované na třetí úrovni, aby mohly vykalkulovat přenosy pro každý z bloků na třetí úrovni.

Blok na třetí úrovni čte informace vygenerované na druhé úrovni, aby mohl vykalkulovat, rozmnožit a vygenerovat informace pro celou sčítačku. Načítá také externí přenos, aby vykalkuloval přenos dat na každém bloku druhé úrovně. Výstup

pro sčítačku je ´1´, pokud třetí úroveň vygenerovalo ´1´, nebo pokud je zmnožení na třetí úrovni ´1´, a externí přenos je ´1´.

Blok sčítačky na třetí úrovni je schopný zpracovat 4 bloky z druhé úrovně. Pakliže jsou jen dva, potom jsou dva bity vyššího řádu kalkulovaného přenosu ignorovány. Tyto dva bity vyššího řádu generovaného vstupu na třetí úrovni jsou nastaveny na nulu `00`, rozmnožené bity vyššího řádu jsou nastaveny na `11`. Toto nastavení způsobuje nepoužitá část k tomu, aby mohly být zmnoženy, ale ne generovány. [5]



Obr. 17: *Diagram bloku.* [5]

3.10.1VHDL:

Program je zapsán pro 4 postupy:

1) `CLA` - 4bitový blok

2) `PG` - kalkuluje prvoúrovňovou informaci zmnožených a generovaných dat

3) `SUM` - kalkuluje sumu XOR vstupů s přenášenými hodnotami vykalkulovanými `CLA`

4) `BITSLICE` - sbírá CLA bloky z první úrovně, kalkulace `PG` a `SUM`.

Tento proces předvádí všechnu práci pro 4 bitovou hodnotu, kromě druhé a třetí úrovně sčítačky. [5]

```
package LOCAL is
     constant N: INTEGER := 4;

     procedure BITSLICE(
          A, B: in BIT_VECTOR(3 downto 0);
          CIN: in BIT;
          signal S: out BIT_VECTOR(3 downto 0);
          signal GP, GG: out BIT);
     procedure PG(
          A, B: in BIT_VECTOR(3 downto 0);
          P, G: out BIT_VECTOR(3 downto 0));
     function SUM(A, B, C: BIT_VECTOR(3 downto 0))
          return BIT_VECTOR;
     procedure CLA(
          P, G: in BIT_VECTOR(3 downto 0);
          CIN: in BIT;
          C: out BIT_VECTOR(3 downto 0);
          signal GP, GG: out BIT);
end LOCAL;

package body LOCAL is


--spočítej součet z výstupu a, b, cin
     procedure BITSLICE(
          A, B: in BIT_VECTOR(3 downto 0);
          CIN: in BIT;
```

```
            signal S: out BIT_VECTOR(3 downto 0);
            signal GP, GG: out BIT) is

        variable P, G, C: BIT_VECTOR(3 downto 0);
      begin
        PG(A, B, P, G);
        CLA(P, G, CIN, C, GP, GG);
        S <= SUM(A, B, C);
      end;
-------------------------------------------------------
      procedure PG(A, B: in BIT_VECTOR(3 downto 0);
           P, G: out BIT_VECTOR(3 downto 0)) is
      begin
           P := A or B;
           G := A and B;
      end;

-- spočítej sumu ze vstupu a přenosu
      function SUM(A, B, C: BIT_VECTOR(3 downto 0))
           return BIT_VECTOR is
      begin
           return(A xor B xor C);
-------------------------------------------------------
      procedure CLA(
           P, G: in BIT_VECTOR(3 downto 0);
           CIN: in BIT;
           C: out BIT_VECTOR(3 downto 0);
           signal GP, GG: out BIT) is

        variable TEMP_GP, TEMP_GG, LAST_C: BIT;
      begin
        TEMP_GP := P(0);
        TEMP_GG := G(0);
        LAST_C := CIN;
        C(0) := CIN;

        for I in 1 to N-1 loop
           TEMP_GP := TEMP_GP and P(I);
           TEMP_GG := (TEMP_GG and P(I)) or G(I);
           LAST_C := (LAST_C and P(I-1)) or G(I-1);
           C(I) := LAST_C;
        end loop;

        GP <= TEMP_GP;
        GG <= TEMP_GG;
      end;
end LOCAL;
```

```
use WORK.LOCAL.ALL;
-------------------------------------------------------
entity ADDER is
     port(A, B: in BIT_VECTOR(31 downto 0);
          CIN: in BIT;
          S: out BIT_VECTOR(31 downto 0);
          COUT: out BIT);
end ADDER;
architecture BEHAVIOR of ADDER is

     signal GG,GP,GC: BIT_VECTOR(7 downto 0);
     -- první úroveň, gen., přenos
     signal GGG, GGP, GGC: BIT_VECTOR(3 downto 0);
     -- druhá úroveň, gen., přenos
     signal GGGG, GGGP: BIT;
     -- třetí úroveň, gen., přenos
begin
     -- spočítej sumu

     -- použij vstup a přenos z první úrovně
     BITSLICE(A( 3 downto 0),B( 3 downto 0),GC(0),
              S( 3 downto 0),GP(0), GG(0));
     BITSLICE(A( 7 downto 4),B( 7 downto 4),GC(1),
              S( 7 downto 4),GP(1), GG(1));
     BITSLICE(A(11 downto 8),B(11 downto 8),GC(2),
              S(11 downto 8),GP(2), GG(2));
     BITSLICE(A(15 downto 12),B(15 downto 12),GC(3),
              S(15 downto 12),GP(3), GG(3));
     BITSLICE(A(19 downto 16),B(19 downto 16),GC(4),
              S(19 downto 16),GP(4), GG(4));
     BITSLICE(A(23 downto 20),B(23 downto 20),GC(5),
              S(23 downto 20),GP(5), GG(5));
     BITSLICE(A(27 downto 24),B(27 downto 24),GC(6),
              S(27 downto 24),GP(6), GG(6));
     BITSLICE(A(31 downto 28),B(31 downto 28),GC(7),
              S(31 downto 28),GP(7), GG(7));

     -- spočítej přenosy z první a druhé úrovně
     process(GP, GG, GGC)
        variable TEMP: BIT_VECTOR(3 downto 0);
     begin
        CLA(GP(3 downto 0), GG(3 downto 0), GGC(0), TEMP,
            GGP(0), GGG(0));
        GC(3 downto 0) <= TEMP;
     end process;
```

```
      process(GP, GG, GGC)
         variable TEMP: BIT_VECTOR(3 downto 0);
      begin
         CLA(GP(7 downto 4), GG(7 downto 4), GGC(1), TEMP,
         GGP(1), GGG(1));
         GC(7 downto 4) <= TEMP;
      end process;

      -- spočítej přenosy z druhé a třetí úrovně
      process(GGP, GGG, CIN)
         variable TEMP: BIT_VECTOR(3 downto 0);
      begin
         CLA(GGP, GGG, CIN, TEMP, GGGP, GGGG);
         GGC <= TEMP;
      end process;

      -- přiřaď nepoužité bity
      GGP(3 downto 2) <= ”11”;
      GGG(3 downto 2) <= ”00”;

      -- spočítej vstupní přenos
      COUT <= GGGG or (GGGP and CIN);
end BEHAVIOR;
```

Př. 11: *Příklad sčítačky předvídaného přenosu ve VHDL kódu.* [5]

Použití:

Při zavádění sčítačky, jsou uplatňovány metody, které předvádějí kalkulaci modelu. Tyto postupy mohou být také napsány jako oddělené entity a použity jako dílčí dosazení, který tvoří hierarchický model. VHDL kompilátor posloupnost entit, ale zhroutí se postup hierarchie sestavené v jednom modelu. Před některými deklaracemi hraničních parametrů stojí klíčové slovo `signal`. Toto klíčové slovo je požadováno pro formální parametry (`out`), pokud skutečnými parametry musí být signály. Výstupní parametr `C` z `CLA` postupu není považovat za signál, takže v postupu není povolen. V takových výzvách mohou být použity pouze signály. Abychom překonali tento problém, jsou používány vedlejší postupy, které uvádějí dočasné proměnné `TEMP`.Princip, je takový, že `TEMP`, které obdrží hodnotu `C` parametru a přidělí mu příslušný signál (obecně užitečná technika). [5]

3.11 Sériovo – Paralelní převodník (počítající bity):

Jedná se zde o model sériově – paralelního převodníku, který zaznamenává sériový proud bitů jako vstupní informace a vytváří osmi bitový výstup.

Model stanovuje tato vstupní data:

`SERIAL_IN` – sériová vstupní data

`RESET` – pokud `´1´` způsobí, že je převodník vymazán, všechny výstupy jsou nastaveny na 0 a převodník je připraven přečíst další sériové slovo.

`CLOCK` – hodnota `RESET` a `SERIAL_IN` je čtena na sousledný přechod tohoto taktu.Výstupy jsou také platné jenom jako sousledný přechod.

Model stanovuje následující výstupy:

`PARALLEL_OUT` – 8 bitová hodnota čtena z `SERIAL_IN` zařízení

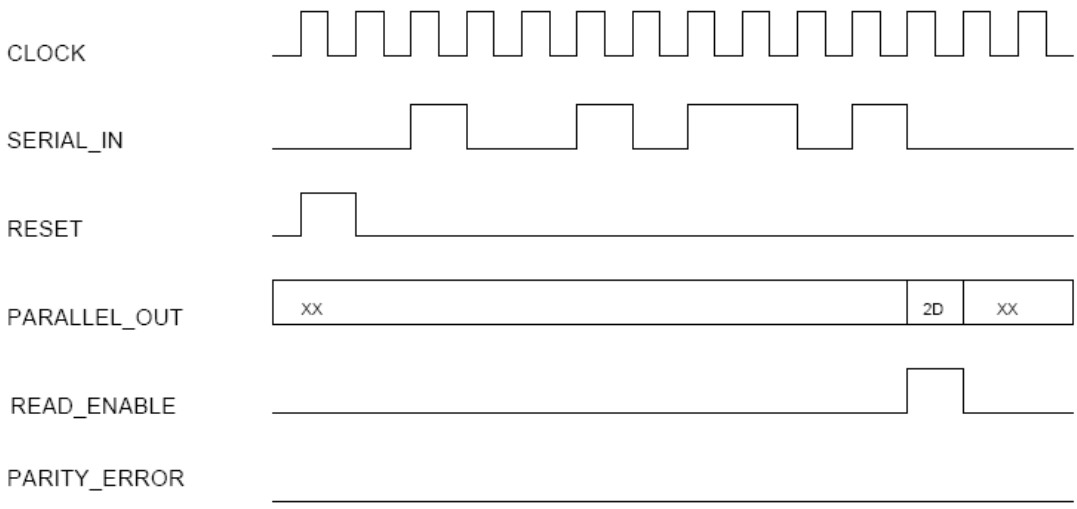
`READ_ENABLE` – pokud je tento výstup `´1´` na sousledném přechodu taktu, mohou být čtena data na `PARALLEL_OUT`

`PARITY_ERROR` – pokud je tento výstup 1 na sousledném přechodu taktu , na portu `SERIÁL_IN` byla nalezena chyba v rovnováze. Pokud je nalezena chyba v rovnováze. Převodník se zastaví, dokud není restartován portem `RESET`. [5]

Vstupní formát:

Pokud nejsou přenášena žádná data sériovým portem, udržuje hodnotu `´0´`. Každá 8 bitová hodnota vyžaduje 10 taktových cyklů k načtení. Po 11 taktovém cyklu může být přečtena paralelní výstupní hodnota. V prvním cyklu je `´1´` umístěna na sériovém vstupu. Toto udělení znamená, že následuje 8 bitová hodnota. Dalších osm cyklů je použito k přenosu každého bitu hodnoty. Nejzásadnější bit je přenášen jako první. 10 a závěrečný cyklus přenáší rovnováhu 8 bitové hodnoty. Musí to být `´0´`, pokud je to sudý počet `´1´`, jsou- li v osmi bitových datech. Pokud převodník nalezne chybu v rovnováze, nastaví výstup `PARITY_ERROR` na

´1´ a čeká dokud nedojde k resetování. Pokud na 11 cyklu stojí READ_ENABLE, výstup je nastaven na ´1´ a osmi bitová hodnota může být načtena z PARALLEL_OUT portu. Pokud SERIAL_IN port má ´1´ na 11 cyklu , další 8 bitová hodnota je okamžitě načtena, jinak převodník čeká dokud se SERIÁL_IN nastaví na ´1´. [5]



Obr. 18: *Vzorek vytváření průběhů převodníku.*[5]

Detaily zavádění-implementace:

Převodník je zde použit jako čtyřstádiový konečný statický automat se synchronním resetem. Pokud je spuštěn reset, je zaveden stav WAIT_FOR_START.

Popis každého stavu je:

WAIT_FOR_START - zůstaň v tomto stavu dokud je ´1´ detekována na sériovém výstupu. Pokud je ´1´ detekována, smaž seznam pomocí PARALLEL_OUT a vytvořte stav pomocí READ_BITS.

READ_BITS - pokud je hodnota CURRENT_BIT_POSITION na čítači 8, všech 8 bitů bylo načteno. Zkontroluje vykalkulovanou rovnováhu s přenášenou rovnováhou a pokud je to v pořádku, přesune se do ALLOW_READ stavu. Jinak přejde do stavu PARITY_ERROR. Pokud ještě nebylo načteno všech

osm bitů, nastaví se vhodný bit ve (PARALLEL_OUT) vyrovnávací paměti na hodnotu SERIAL_IN. Dále vykalkuluje rovnováhu bitů do této chvíle načtenou a přičtěte CURRENT_BIT_POSITION.

ALLOW_READ - toto je stav, kde okolní čte hodnotu PARALLEL_OUT. Pokud je ale tato hodnota načtena, model se vrátí do stavu WAIT_FOR_START.

PARITY_ERROR_DETECTED - v tomto stavu je výstup PARITY_ERROR nastaven na ´1´ a nic dalšího není provedeno. [5]

Tento model má čtyři hodnoty uchované v seznamech:

CURRENT_STATE - pamatuje si stav z předcházejícího taktu

CURRENT_BIT_POSITION - pamatuje si kolik bylo do této chvíle načteno bitů

CURRENT_PARITY - udržuje XOR načtených bitů v chodu

CURRENT_PARALLEL_OUT - skladuje každý paralelní bit, tak, jak byl nalezen.

Model je rozdělen mezi dva procesy (postupy), a to kombinace NEXT_ST zahrnující kombinační logiku a sekvenční SYNCH, která je taktována. [5]

NEXT_ST - proces začíná prvním určením všech chybných hodnot k signálům, které je řídí. Toto určení garantuje, že všechny signály jsou řízeny za všech podmínek. Dále je vytvořen RESET vstup. Pokud není RESET aktivní, pak stanovisko CASE určí současný stav a jeho kalkulaci. Změna stavu je předvedena uvedením další statické hodnoty, kterou potřebujete pro signál NEXT_STATE. [5]

Samotná sériová-paralelní změna je předváděna těmito dvěma příkazy v procesu NEXT_ST :

```
NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <= SERIAL_IN;
NEXT_BIT_POSITION <= CURRENT_BIT_POSITION + 1;
```

První příkaz přiřazuje současný sériový vstup bitů k určitému bitu paralelního výstupu. Druhý příkaz přičte další pozici bitu, který má být určena. [5]

SYNCH zapisuje a aktualizuje uchovávané hodnoty popsané výše. Každý zapsaný signál má dvě části, NEXT_ … a CURRENT_…

NEXT_…signály uchovávají hodnoty vykalkulované v procesu NEXT_ST.

CURRENT_…signály udržují hodnoty vytvořené procesem SYNCH. Signály CURRENT_…uchovávají hodnoty NEXT_…signály jako posledně uchované v taktu. [5]

3.11.1VHDL:

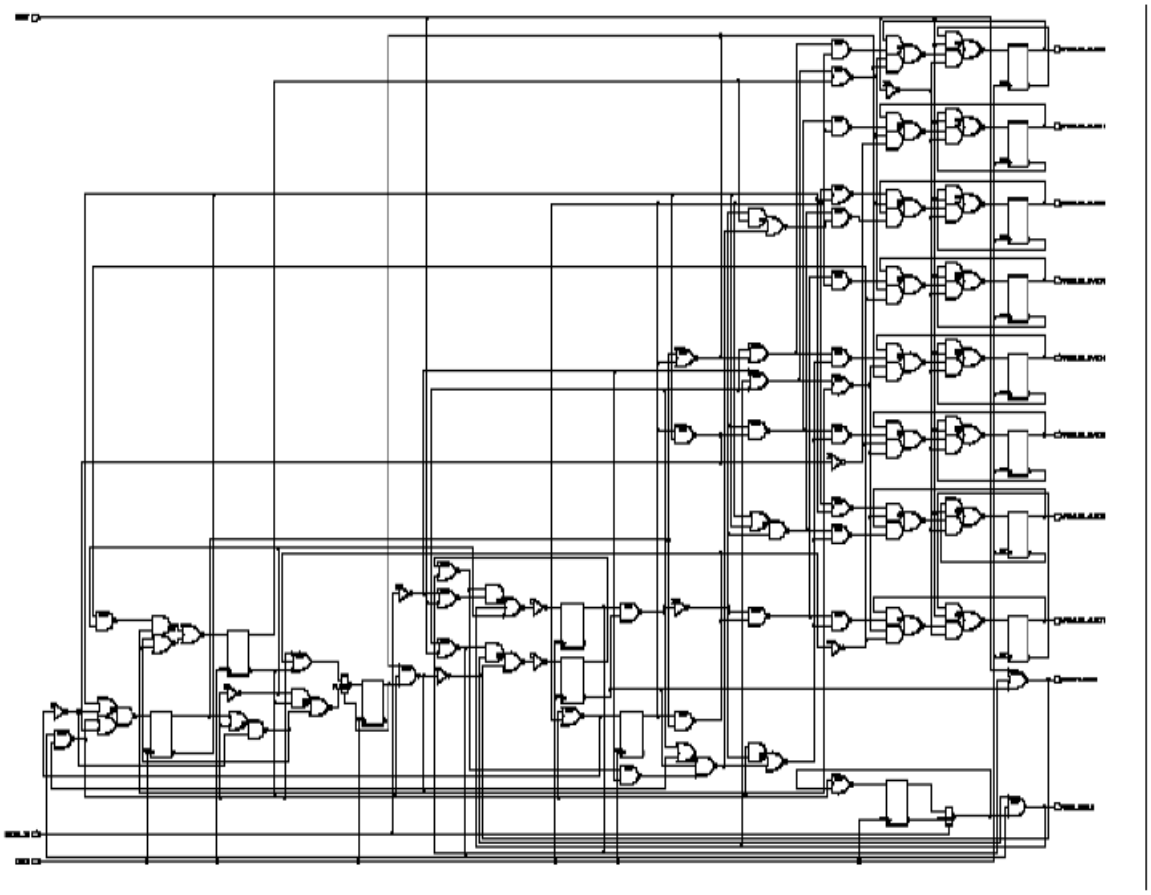
```
-- převodník ze sériového na paralelní kód
package TYPES is

    -- deklarace proměnných
    type STATE_TYPE is (WAIT_FOR_START,
                        READ_BITS,
                        PARITY_ERROR_DETECTED,
                        ALLOW_READ);
    constant PARALLEL_BIT_COUNT: INTEGER := 8;

    subtype PARALLEL_RANGE is INTEGER

        range 0 to (PARALLEL_BIT_COUNT-1);
    subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
    end TYPES;
```

```
use WORK.TYPES.ALL;

entity SER_PAR is    -- deklarování rozhraní
     port(SERIAL_IN, CLOCK, RESET: in BIT;
          PARALLEL_OUT: out PARALLEL_TYPE;
          PARITY_ERROR, READ_ENABLE: out BIT);
end;
architecture BEHAVIOR of SER_PAR is
     signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
     signal CURRENT_PARITY, NEXT_PARITY: BIT;
     signal CURRENT_BIT_POSITION, NEXT_BIT_POSITION:
          INTEGER range PARALLEL_BIT_COUNT downto 0;
     signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
          PARALLEL_TYPE;
begin
NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_BIT_POSITION, CURRENT_PARITY,
                 CURRENT_PARALLEL_OUT)

-- výpočet výstupních stavů
begin
     PARITY_ERROR <= '0';
     READ_ENABLE <= '0';
     NEXT_STATE <= CURRENT_STATE;
     NEXT_BIT_POSITION <= 0;
     NEXT_PARITY <= '0';
     NEXT_PARALLEL_OUT <= CURRENT_PARALLEL_OUT;

     if (RESET = '1') then -- synchronníreset
          NEXT_STATE <= WAIT_FOR_START;
     else
          case CURRENT_STATE is
               when WAIT_FOR_START =>
                    if (SERIAL_IN = '1') then
                         NEXT_STATE <= READ_BITS;
                         NEXT_PARALLEL_OUT <=
                              PARALLEL_TYPE'(others=>'0');
                    end if;
               when READ_BITS =>
                    if (CURRENT_BIT_POSITION =
                              PARALLEL_BIT_COUNT) then
                     if (CURRENT_PARITY = SERIAL_IN) then
                              NEXT_STATE <= ALLOW_READ;
                              READ_ENABLE <= '1';
                     else
                     NEXT_STATE <= PARITY_ERROR_DETECTED;
```

```
                    end if;
                  else
          NEXT_PARALLEL_OUT(CURRENT_BIT_POSITION) <=
                        SERIAL_IN;
          NEXT_BIT_POSITION <=
                        CURRENT_BIT_POSITION + 1;
          NEXT_PARITY <= CURRENT_PARITY xor
                        SERIAL_IN;
                  end if;
          when PARITY_ERROR_DETECTED =>
               PARITY_ERROR <= '1';
          when ALLOW_READ =>
               NEXT_STATE <= WAIT_FOR_START;
       end case;
     end if;
end process;

SYNCH: process

-- uložení proměnných
     begin
          wait until CLOCK'event and CLOCK = '1';
          CURRENT_STATE <= NEXT_STATE;
          CURRENT_BIT_POSITION <= NEXT_BIT_POSITION;
          CURRENT_PARITY <= NEXT_PARITY;
          CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
     end process;

PARALLEL_OUT <= CURRENT_PARALLEL_OUT;¨

end BEHAVIOR;
```

Př. 12: *Příklad sériovo – paralelního převodníku (počítající bity) ve VHDL kódu.* [5]

3.11.2Schéma:



Obr. 19: *Schéma zapojení sériovo – paralelního převodníku (počítající bity).* [5]

3.12 Sériově – Paralelní převodník (posouvání bitů) :

Tento převodník je opět podobný jako předchozí. Má stejnou funkci jako v předešlém případě, ale používá jiný algoritmus, aby provedl konverzi. V předcházejícím použití byl použit čítač, aby určoval bity výstupu, který byl nastaven, pokud byla načtena nová série bitů.V tomto užití je série bitů posunuta na místo. Ještě než se uskuteční konverze, je ´1´ umístěna na nejméně patrnou bitovou pozici. Je-li ´1´ posunuta z nejdůležitější pozice (pozice 0), signál NEXT_HIGH_ BIT je nastaven na ´1´ a konverze je kompletní.Vyjmenování druhého použití je následující. Rozdíly jsou zvýrazněny tučně.Vztahují se k přemístění ….BIT_POSITION signálů, dodání ….HIGH_BIT signálů a změna v cestě NEXT_PARALLEL_OUT je vykalkulována. [5]

Schéma pro zavedení posouvání bitů je mnohem jednodušší než v předchozím případě. Je jednodušší, protože posouvací algoritmus je snazší zavést. S počítaným algoritmem, každý z klopných obvodů (flip-flop), majících PARALLEL_OUT bity, potřebuje systém, který dekóduje hodnotu zachovanou v BIT_POSITION klopných obvodech, aby bylo vidět, kdy a kam je třeba uvést hodnotu SERIAL_IN. Také BIT_POSITION klopného obvodu potřebuje přírůstek k vykalkulování jejich další hodnoty. Naproti tomu nastavitelný algoritmus nevyžaduje žádné přírůstky ani flip- flop, aby udržely BIT_POSITION. Navíc systém před většinou PARALLEL_OUT bitů potřebuje načíst pouze hodnotu předcházejícího flip-flopu, nebo ´0´. Hodnota záleží na tom, zda jsou bity právě načítány. V nastavitelném algoritmu (shifter algorithm) , port SERIAL_IN potřebuje být propojen pouze s nejméně podstatným bitem (číslo 7) PARALLEL_OUT flip-flopu. [5]

Tato dvě použití ilustrují důležitost vytvoření dobře fungujících algoritmů. Oba fungují náležitě, ale nastavitelný algoritmus vytváří rychlejší a efektivnější model. [5]

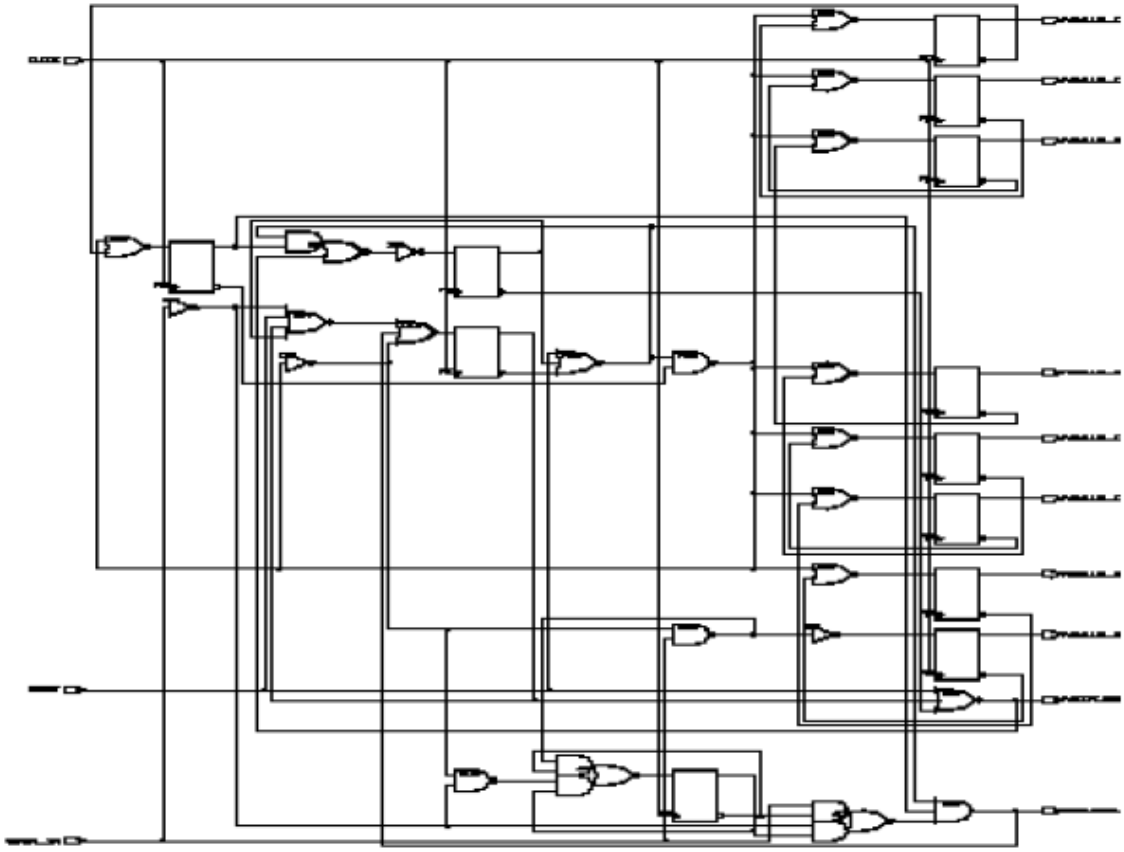
3.12.1 VHDL:

```
package TYPES is
     -- deklaruje proměnné
     type STATE_TYPE is (WAIT_FOR_START,
                         READ_BITS,
                         PARITY_ERROR_DETECTED,
                         ALLOW_READ);
     constant PARALLEL_BIT_COUNT: INTEGER := 8;
     subtype PARALLEL_RANGE is INTEGER
          range 0 to (PARALLEL_BIT_COUNT-1);
     subtype PARALLEL_TYPE is BIT_VECTOR(PARALLEL_RANGE);
end TYPES;

use WORK.TYPES.ALL;

entity SER_PAR is        -- deklaruje rozhraní
     port(SERIAL_IN, CLOCK, RESET: in BIT;
          PARALLEL_OUT: out PARALLEL_TYPE;
          PARITY_ERROR, READ_ENABLE: out BIT);

end;

architecture BEHAVIOR of SER_PAR is
     -- Signály pro uložené signály
     signal CURRENT_STATE, NEXT_STATE: STATE_TYPE;
     signal CURRENT_PARITY, NEXT_PARITY: BIT;
     signal CURRENT_HIGH_BIT, NEXT_HIGH_BIT: BIT;
     signal CURRENT_PARALLEL_OUT, NEXT_PARALLEL_OUT:
          PARALLEL_TYPE;
begin

NEXT_ST: process(SERIAL_IN, CURRENT_STATE, RESET,
                 CURRENT_HIGH_BIT, CURRENT_PARITY,
                 CURRENT_PARALLEL_OUT)
-- výpočet stavů všech výstupů
begin
     PARITY_ERROR <= '0';
     READ_ENABLE <= '0'; -- výstupy a uložené hodnoty
     NEXT_STATE <= CURRENT_STATE;
     NEXT_HIGH_BIT <= '0';
     NEXT_PARITY <= '0';
     NEXT_PARALLEL_OUT <= PARALLEL_TYPE'(others=>'0');
     if(RESET = '1') then -- synchronní reset
          NEXT_STATE <= WAIT_FOR_START;
     else
          case CURRENT_STATE is -- stav procesu
               when WAIT_FOR_START =>
```

```
                    if (SERIAL_IN = '1') then
                         NEXT_STATE <= READ_BITS;
                         NEXT_PARALLEL_OUT <=
                             PARALLEL_TYPE'(others=>'0');
                    end if;
               when READ_BITS =>
                    if (CURRENT_HIGH_BIT = '1') then
                    if (CURRENT_PARITY = SERIAL_IN) then
                         NEXT_STATE <= ALLOW_READ;
                         READ_ENABLE <= '1';
                    else
                     NEXT_STATE <= PARITY_ERROR_DETECTED;
                    end if;
                    else
               NEXT_HIGH_BIT <= CURRENT_PARALLEL_OUT(0);
               NEXT_PARALLEL_OUT <=
                    CURRENT_PARALLEL_OUT(
                    1 to PARALLEL_BIT_COUNT-1) &
                 SERIAL_IN;
               NEXT_PARITY <= CURRENT_PARITY xor
                              SERIAL_IN;
          end if;
       when PARITY_ERROR_DETECTED =>
          PARITY_ERROR <= '1';
       when ALLOW_READ =>
          NEXT_STATE <= WAIT_FOR_START;
     end case;
  end if;
end process;

SYNCH: process

-- uložení hodnot přes jeden hodinový cyklus
     begin
          wait until CLOCK'event and CLOCK = '1';
          CURRENT_STATE <= NEXT_STATE;
          CURRENT_HIGH_BIT <= NEXT_HIGH_BIT;
          CURRENT_PARITY <= NEXT_PARITY;
          CURRENT_PARALLEL_OUT <= NEXT_PARALLEL_OUT;
     end process;
     PARALLEL_OUT <= CURRENT_PARALLEL_OUT;
end BEHAVIOR;
```

Př. 13: *Příklad sériovo –paralelního převodníku (posouvání bitů) ve VHDL kódu.* [5]

3.12.2 Schéma:



Obr. 20: *Schéma zapojení sériovo – paralelního převodníku (posouvání bitů).* [5]

3.13 Programovatelné logického pole (PLA):

Zde je popsán způsob, jak sestavit programovatelné logické pole PLA. Funkce PLA používá vstupní vyhledaný vektor jako index do neměnné PLA tabulky, poté vrací výstupní vektor specifikovaný PLA. PLA tabulka je pole `PLA_ROW`, kde každá řada je pole `PLA_ELEMENT`. Každý element je buď 1, 0, mínus nebo mezera (`´1´,´0´,´ -´ ,´ ´`). Tabulka je rozdělena mezi vstupní plochu a výstupní plochu. Vstupní plocha je specifikována nulami a jedničkami a mínusy. Výstupní plocha je specifikována nulami a jedničkami. Hodnoty obou ploch jsou odděleny mezerou. Ve funkci PLA, je první výstupní vektor poprvé nastaven tak, aby byl tvořen samými nulami. Když vstupní vektor odpovídá vstupní ploše v řadě PLA tabulky, jedničky ve výstupní ploše jsou přiřazeny odpovídajícím bitům ve výstupním vektoru. Shoda je určena tak, že pokud `´0´` nebo `´1´` je ve vstupní ploše, vstupní vektor musí mít stejnou hodnotu na stejné pozici a pokud je ve vstupní ploše mínus nebo mezera, shoduje se jakýmkoliv vstupním vektorem na této pozici. [PLA tabulky a PLA funkce jsou definovány v balíku `LOCAL`. Entita `PLA_VHDL`, která používá `LOCAL`, potřebuje pouze specifikovat svojí PLA tabulku jako konstantu a pak spustit PLA funkci. PLA funkce nezáleží výslovně na velikosti PLA. Změnit velikost PLA můžete změnou nastavení tabulkové konstanty a nastavením konstant `INPUT_COUNT`, `OUTPUT_COUNT` a `ROW_COUNT`.
V následujícím příkladu, jsou je rozsah výstupního portu `4 downto 0 OUT_VECTOR`. Tento příklad je uveden hlavně k předvedení možností VHDL . Je mnohem efektivnější definovat PLA přímo, použitím vstupního PLA formátu pro více informací o PLA vstupním formátu. [5]

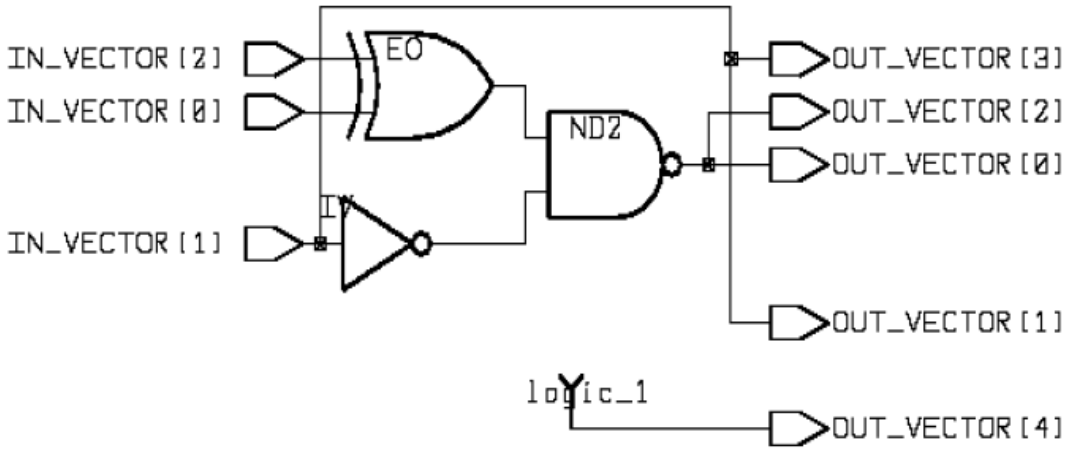
3.13.1VHDL:

```
package LOCAL is
     constant INPUT_COUNT: INTEGER := 3;
     constant OUTPUT_COUNT: INTEGER := 5;
     constant ROW_COUNT: INTEGER := 6;
     constant ROW_SIZE: INTEGER := INPUT_COUNT +
                                   OUTPUT_COUNT + 1;
     type PLA_ELEMENT is ('1', '0', '-', ' ');
     type PLA_VECTOR is
          array (INTEGER range <>) of PLA_ELEMENT;
     subtype PLA_ROW is
          PLA_VECTOR(ROW_SIZE - 1 downto 0);
     subtype PLA_OUTPUT is
          PLA_VECTOR(OUTPUT_COUNT - 1 downto 0);
     type PLA_TABLE is
          array(ROW_COUNT - 1 downto 0) of PLA_ROW;

     function PLA(IN_VECTOR: BIT_VECTOR;
             TABLE: PLA_TABLE)
          return BIT_VECTOR;
end LOCAL;


package body LOCAL is
     function PLA(IN_VECTOR: BIT_VECTOR;
                  TABLE: PLA_TABLE)
       return BIT_VECTOR is
     subtype RESULT_TYPE is
         BIT_VECTOR(OUTPUT_COUNT - 1 downto 0);
     variable RESULT: RESULT_TYPE;
     variable ROW: PLA_ROW;
     variable MATCH: BOOLEAN;
     variable IN_POS: INTEGER;

begin
     RESULT := RESULT_TYPE'(others => BIT'( '0' ));

     for I in TABLE'range loop
          ROW := TABLE(I);

          MATCH := TRUE;
          IN_POS := IN_VECTOR'left;
```

```
          -- kontrola shody vstupů
          for J in ROW_SIZE - 1 downto OUTPUT_COUNT loop
               if(ROW(J) = PLA_ELEMENT'( '1' )) then
                    MATCH := MATCH and
                       (IN_VECTOR(IN_POS) = BIT'( '1' ));
               elsif(ROW(J) = PLA_ELEMENT'( '0' )) then
                    MATCH := MATCH and
                       (IN_VECTOR(IN_POS) = BIT'( '0' ));
               else
                    null; -- musí být mínus
               end if;
               IN_POS := IN_POS - 1;
          end loop;

          -- nastav výstupy
          if(MATCH) then
               for J in RESULT'range loop
               if(ROW(J) = PLA_ELEMENT'( '1' )) then
                    RESULT(J) := BIT'( '1' );
               end if;
          end loop;
     end if;
end loop;
return(RESULT);
     end;
end LOCAL;

use WORK.LOCAL.all;
entity PLA_VHDL is
     port(IN_VECTOR: BIT_VECTOR(2 downto 0);
          OUT_VECTOR: out BIT_VECTOR(4 downto 0));
end;

architecture BEHAVIOR of PLA_VHDL is
     constant TABLE: PLA_TABLE := PLA_TABLE'(
          PLA_ROW'("--- 10000"),
          PLA_ROW'("-1- 01000"),
          PLA_ROW'("0-0 00101"),
          PLA_ROW'("-1- 00101"),
          PLA_ROW'("1-1 00101"),
          PLA_ROW'("-1- 00010"));
begin
     OUT_VECTOR <= PLA(IN_VECTOR, TABLE);
end BEHAVIOR;
```

Př. 14: *Příklad programovatelného logického pole ve VHDL kódu.* [5]

3.13.2Schéma:



Obr. 21: *Schéma zapojení programovatelného logického pole.* [5]

4. ZÁVĚR

Tato bakalářská práce měla za cíl se seznámit s jazykem VHDL a obvody FPGA a CPLD. Je nutné si uvědomit, že pro vytvoření vzorových úloh je zapotřebí dobře znát vnitřní zapojení těchto obvodů a také dobře ovládat jazyk VHDL. Je také důležité si vhodně zvolit metodu, jakou se daná úloha bude řešit. Někdy je vhodné za pomocí SW ISE pouze nakreslit schéma nebo použít metodu vytvoření stavového diagramu, což je nejvhodnější například u stavových automatů. Někdy je ovšem výhodnější napsat stořádkový program ve VHDL, než kreslit složité schéma. Některé vzorové úlohy byly odzkoušeny na demonstrační vývojové desce pro programovatelné logické obvody s obvodem Xilinx řady XC9500. Výhodou této demonstrační desky je nízká cena součástek, dostupnost SW vybavení a možnost vývoje a testovaní vzorových úloh. Výhodou obvodů CPLD od firmy Xilinx je nepochybně jejich dobrá cenová dostupnost, rychlost, nízká spotřeba a univerzálnost. Proto tyto obvody mají velkou budoucnost. V jazyku VHDL toho bylo již hodně vymyšleno a je otázkou, zda jde ještě něco převratného vymyslet. Pro návrháře je spíše důležité si uvědomit, jakou metodu při návrhu použít.

5. PŘEHLED POUŽITÉ LITERATURY:

[1] Doc. Ing. Jaromír Kolouch, skripta, Programovatelné logické obvody

[2] Doc. Ing. Jaromír Kolouch, skripta, Programovatelné logické obvody (počítačové cvičení)

[3] Jiří Pinker, Martin Poupa, knížka, Číslicové systémy a jazyk VHDL

[4] Zdeněk Oprchal, diplomová práce, Vývojová deska pro programovatelné automaty

[5] A Examples, skripta, HDL_AA.pdf

6. SEZNAM ILUSTRACÍ:

7. SEZNAM TABULEK:

8. SEZNAM PŘÍKLADŮ: