Návrh počítačových systémů
INP

Studijní opora

Úvod do jazyka VHDL

Lukáš Sekanina verze 1.2006

Tento učební text vznikl za podpory projektu „Zvýšení konkurenceschopnosti IT odborníků – absolventů pro Evropský trh práce“, reg. č. CZ.04.1.03/3.2.15.1/0003. Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

Použité piktogramy


Počítačové cvičení, příklad


Otázka, příklad k řešení

Příklad


Slovo tutora, komentář


Potřebný čas pro studium, doplněno číslicí přes hodiny



Reference

Souhrn

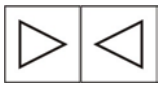

Správné řešení

Obtížná část


Důležitá část

Cíl

Definice


Zajímavé místo (levá, pravá varianta)

Rozšiřující látka, informace, znalosti. Nejsou předmětem zkoušky.

1 Úvod a motivace

Moderní výuka metod návrhu číslicových obvodů se neobejde bez uvedení studentů do základů specializovaných jazyků pro popis obvodů HDL (Hardware Description Language). Mezi tyto jazyky řadíme zejména VHDL a Verilog. V poslední době je trendem využívat pro popis hardware i jazyky, které jsou bližší softwarovým inženýrům, např. Handel-C nebo System-C. Tento text je věnován úvodu do jazyka VHDL, který je asi nejpopulárnější a je od roku 1987 standardem IEEE. Cílem výkladu není předvést všechny možnost jazyka VHDL (k tomu slouží řada kvalitních učebnic a textů a zejména dobře vedené internetové stránky - některé odkazy jsou uvedeny v seznamu literatury), ale vysvětlit základní principy a konstrukce, které tento jazyk odlišují od standardních programovacích jazyků jako Java, C nebo Pascal a které student využije v rámci kurzů zabývajících se počítačovými obvody na FIT VUT v Brně. Výklad bude veden z pohledu studenta, který je již schopen programovat ve standardních programovacích jazycích a má základní představu o činnosti číslicových obvodů. Budeme se zabývat obvodovou implementací jak specializovaných obvodů tak i podsystémů procesorů.

1.1 Implementace algoritmu

V rámci počítačového inženýrství jsou vytvářena výpočetní zařízení, která realizují algoritmy. Připomeňme význam termínu **algoritmus** (z kurzu Základy programování):

Algoritmus je konečná uspořádaná množina úplně definovaných kroků pro vyřešení nějakého problému. Intuitivně algoritmem rozumíme postup, který nás dovede k řešení úlohy. Formálněji vyjádřeno se jedná o přesně definovanou konečnou posloupnost příkazů (kroků), jejichž prováděním pro každé přípustné vstupní hodnoty získáme po konečném počtu kroků odpovídající hodnoty výstupní.

Algoritmus můžeme implementovat dvojím způsobem:

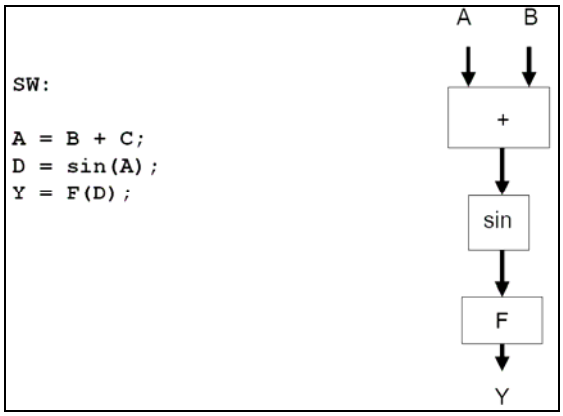
- jako program na univerzálním procesoru, který je nejčastěji prováděn sekvenčně, nebo
- jako specializovaný číslicový obvod.

Následující příklady demonstrují, jak je možné typické softwarové konstrukce implementovat obvodově. Z teorie algoritmů víme, že jsme-li schopni realizovat sekvenci, selekci a iteraci, jsme schopni řešit jakýkoliv algoritmicky řešitelný problém.

1.1.1 Sekvence: softwarové vs hardwarové řešení

Následující obrázek demonstruje obvodovou realizaci sekvence. Operátory (+, sin) a specializované funkce (F) jsou realizovány v hardware pomocí speciálních obvodů (komponent) propojených vodiči. Pořadí umístění komponent v obvodu odpovídá posloupnosti instrukcí v programu. Datová šířka vodičů odpovídá počtu bitů u příslušného datového typu proměnných.

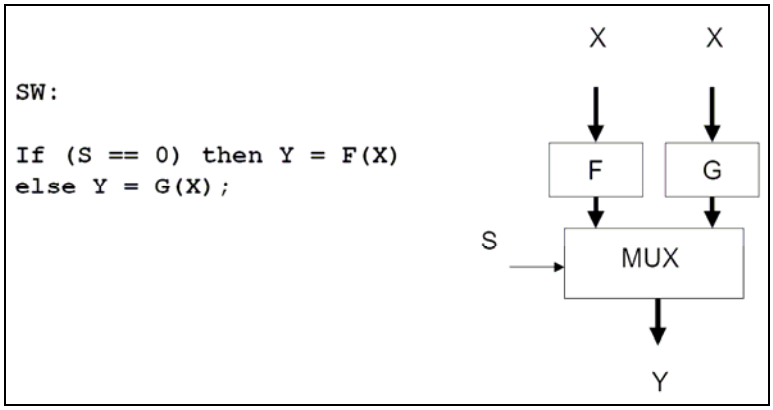
Obr. 1. Realizace sekvence



1.1.2 Selekce: softwarové vs hardwarové řešení

Větvení v programu se typicky implementuje pomocí multiplexorů. Řídicí proměnná je v obvodu připojena na řídicí vstup multiplexoru, který vybírá výstup jedné z komponent F nebo G.

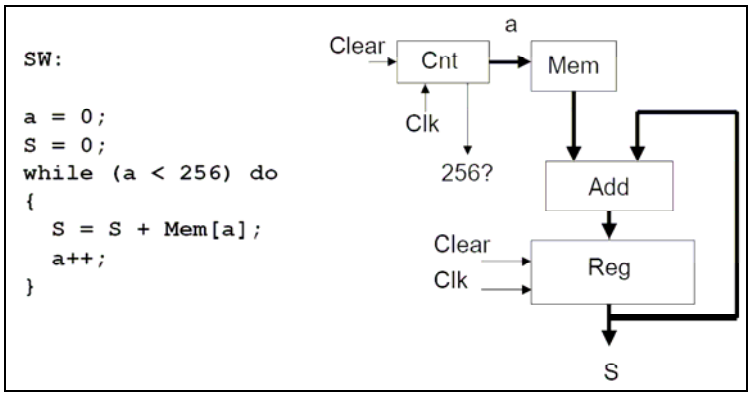
Obr. 2. Realizace selekce



1.1.3 Iterace: softwarové vs hardwarové řešení

Na levé straně následujícího obrázku je uveden program, který sečte hodnoty paměťových buněk paměti o adresách 0 až 255. Proměnná *a* je aktuální adresou v paměti, proměnná *S* představuje aktuální součet. Obě proměnné jsou na začátku programu vynulovány. V každé iteraci je třeba přičíst obsah paměťové buňky s adresou *a* k aktuálnímu součtu *S* a zvýšit počítadlo adres. Obvodové řešení využívá sčítačku Add, která sčítá obsah paměťové buňky s adresou *a* s obsahem registru Reg, který obsahuje aktuální součet. Pomocí signálu Clear je čítač adres Cnt na počátku výpočtu vynulován. Stejně tak je vynulován registr Reg. Důležitou částí systému je hodinový signál Clk, který zajišťuje synchronizaci komponent. Výpočet je ukončen, pokud dojde k přetečení čítače. Celé obvodové řešení je založeno na souběžně pracujících komponentách, které musí být vhodně propojeny a synchronizovány.

Obr. 3. Realizace iterace



1.1.4 Porovnání programové a obvodové implementace

Proč je výhodné vytvořit obvodovou implementaci algoritmu? Důvodů je několik:

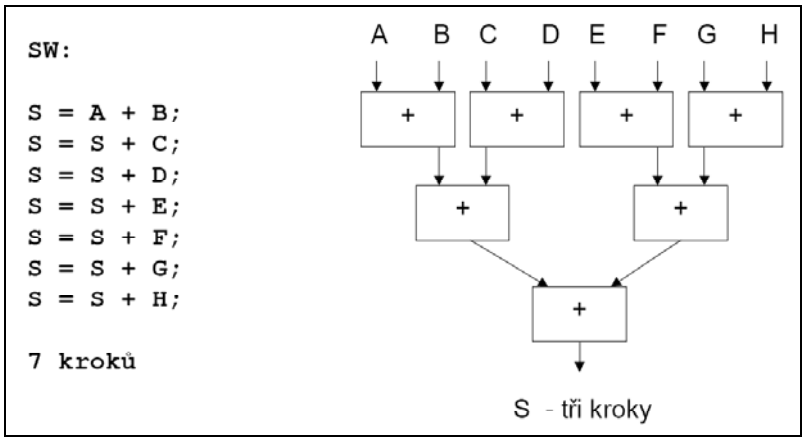
- Obvodová implementace je obvykle výrazně rychlejší než procesor vyrobený stejnou technologií. Neztrácí se čas načítáním instrukcí a prováděním celé řady operací, které jsou nutné pro činnost procesoru, ale nejsou třeba k řešení daného problému. Je možné využít vyššího stupně paralelismu, aplikačně specifických komponent (jako např. rychlé Fourierovy transformace) a speciálního kódování, které se na běžných procesorech nevyskytují.
- Plocha na čipu je obvykle menší, implementují se jen nezbytně nutné komponenty. Např. pokud je potřeba, můžeme efektivně implementovat nějakou netypickou aritmetiku (např. na 22 bitech), kterou bychom na běžném procesoru museli implementovat na 32 bitech.
- Z výše uvedeného důvodu je také obvykle i spotřeba elektrické energie nižší.

Nevýhody obvodové realizace jsou následující:

- Výrobní náklady jsou vyšší než nákup obecně použitelného procesoru.
- Další nevýhodou obvodové realizace je, že její návrh je obvykle obtížnější (a tedy i dražší) než návrh odpovídajícího programu pro procesor.
- Obvodová realizace bývá také méně flexibilnější než řešení využívající procesor. Typicky je obvodová realizace jednoúčelová.

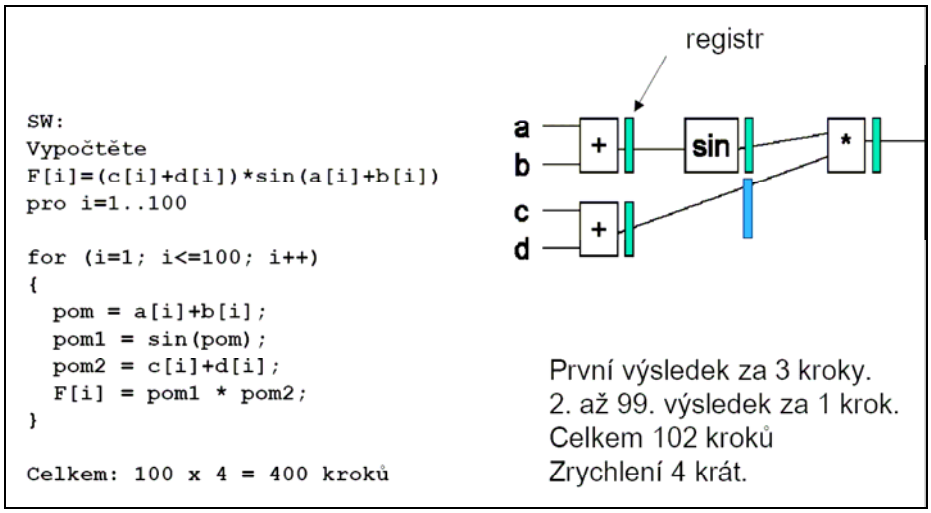
Následující příklady demonstrují dvě základní techniky zvyšování výkonnosti, které se používají v HW – paralelní zpracování a zřetězené zpracování.

Obr. 4. Paralelní zpracování



Sečtení osmi proměnných na sekvenčním procesoru (v sedmi krocích) a na speciální paralelní architektuře, která využívá 7 sčítaček zapojených do stromu, je časově výhodnější v hardware (výpočet trvá 3 kroky). Oproti tomu procesor využívá pouze jednu sčítačku.

Obr. 5. Zřetězené zpracování



Obr. 5 ukazuje výpočet výrazu F[i] pro 100 vstupních vektorů. Procesor provede výpočet za 400 kroků. Pomocí tzv. řetězeného zpracování, kdy je obvodová realizace rozdělena pomocí registrů do synchronizovaných stupňů, bude výpočet proveden během 102 kroků (první výsledek bude k dispozici za 3 kroky, dalších 99 výsledků potom v každém taktu, tj. v jednom kroku).

1.2 Kroky moderního návrhu číslicových obvodů

Návrh číslicového systému probíhá v těchto krocích:

- Vstupem je odladěný zdrojový kód (např. v jazyce VHDL, Verilog, SystemC, HandelC apod.) nebo schéma zapojení, požadavek na celkové zpoždění, plochu, příkon apod.
- Následuje syntéza, v rámci které se popis obvodu transformuje na schéma zapojení na úrovni hradel.
- Následuje mapování tohoto schématu na elementy cílové technologii. Takovým elementem může být hradlo NAND popsané na úrovni tranzistorů, které jsou rozmístěny na čipu přesně dle požadavků výroby. Jiným příkladem elementu cílové technologie je CLB (Configurable Logic Block) v obvodech FPGA (Field Programmable Gate Array).

- Dále jsou tyto elementy vhodně rozmístěny a propojeny.
- V posledním kroku je vygenerována maska pro výrobu integrovaného obvodu nebo konfigurační informace pro FPGA.

Všechny kroky probíhají automaticky vy vývojovém nástroji. Návrhář má možnost zasahovat do jednotlivých etap. Typicky dochází po ukončení každé fáze k ověření činnosti obvodu a kontrole časování.

1.3 Způsoby popisu číslicového obvodu

Číslicový obvod můžeme popsat dvěma základními způsoby: *strukturně* nebo *behaviorálně*.

Strukturní (nebo také strukturální) popis znamená, že si nejdříve nakreslíme schéma zapojení obvodu a posléze toto schéma zapíšeme pomocí konstrukcí daného HDL. Tedy popíšeme jednotlivé komponenty obvodu a jejich propojení pomocí vodičů. Komponenty mohou být rovněž dekomponovány na subkomponenty a samostatně popsány. Lze takto definovat hierarchii komponent. Tento styl popisu má tu výhodu, že je velice blízký finální obvodové realizaci, návrhář má do jisté míry pod kontrolou proces syntézy a časování.

Behaviorální popis (nebo také popis chování) využívá faktu, že chování obvodu můžeme popsat algoritmem. Používáme konstrukce běžné v programovacích jazycích (cykly, procedury, funkce apod.). Při tomto popisu neuvažujeme obvodové detaily, tvorbu zapojení obvodu necháváme na procesu syntézy, který však nemusí být nutně průchodný. Pro některé programové konstrukce totiž neexistuje obvodový ekvivalent (např. pro rekurzi). Tento způsob s výhodou použijeme, pokud nám stačí obvody pouze simulovat a finální implementace není důležitá.

Zde je dobré si uvědomit, že elementární komponenty, které používáme ve strukturním popisu, jsou vždy popsány behaviorálně. Na určité úrovni je vždy třeba ukončit strukturní popis a definovat elementární objekty, se kterými pracujeme a jejichž realizaci již nevysvětlujeme. Např. realizaci logického členu AND zapíšeme jako `z <= a and b;` kdy význam logické operace *and* již nemusíme dodat např. popisem implementace na úrovni tranzistorů.

Popsané způsoby se často kombinují. Kromě uvedených možností existují i speciální způsoby popisu, např. dataflow nebo generický popis, se kterými se seznámíme později.

1.4 Poznámky k procesu syntézy

1.4.1 Behaviorální syntéza

Jedná se o syntézu abstraktního chování, kdy je daný algoritmus popsán na nejvyšší úrovni. K popisu mohou byt použity všechny syntaktické konstrukce, které HDL jazyk poskytuje. Popis je soustředěn na tok dat, omezení vztahující se na vstup a výstup a uživatelská omezení. Výstupem behaviorální syntézy je popis na úrovni meziregistrových přenosů, který má stejné chování jako vstupní obvod. Nevýhodou je, že veškeré optimalizace jsou nechány na nástrojích realizujících syntézu. Jediná možnost, jak řídit výsledek syntézy, je správné

nastavení omezení a nástrojů.

1.4.2 RTL syntéza

Jedná se o syntézu na úrovni meziregistrových přenosů. Vstupní obvod je tedy popsán pomocí registrů, čítačů, automatů atd. Popis se vyznačuje tím, že je od sebe oddělena *datová* a *řidicí* cesta. Řidicí cestou se většinou rozumí FSM automat. Datovou cestou pak různé logické sítě, sčítačky, registry atd. Celá datová cesta je pak řízena pomocí signálů z řidicí cesty. Syntezátor z tohoto popisu vygeneruje popis na úrovni hradel. Výsledek syntézy zároveň obsahuje optimalizovanou datovou cestu, paměti a řídicí struktury.

1.4.3 Logická syntéza

Syntéza na úrovni hradel. Popis je složen pouze z komponent cílové technologie vzájemně propojených pomocí vodičů.

1.5 Shrnutí

V této kapitole byly vysvětleny základní rozdíly mezi programovou a obvodovou realizací algoritmu. Dále byly uvedeny základní konstrukce používané pro paralelizaci řešení na úrovni hardware. V rámci popisu procesu návrhu moderních číslicových obvodů byla osvětlena problematika syntézy.

2 Jazyk VHDL

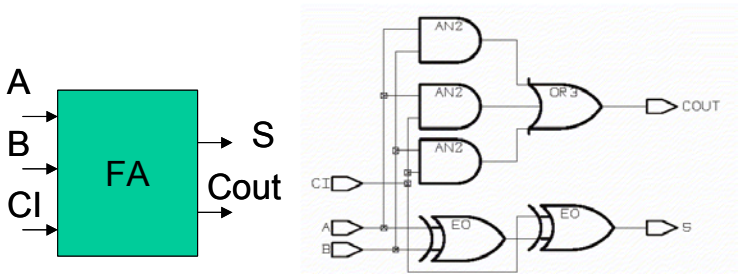
VHDL je akrynom z anglického „Very high speed integrated circuits HDL“. VHDL je standardem IEEE od r. 1987, byl revidován v roce 1997 a je použitelný i pro návrh analogových obvodů. Jedná se o typovaný programovací jazyk. VHDL má prostředky pro popis paralelismu, konektivity a explicitní vyjádření času. Jazyk VHDL se používá jak pro simulaci obvodů, tak i pro popis integrovaných obvodů, které se mají vyrábět. Cílem této kapitoly je představit základní filozofii jazyka VHDL a jeho typické konstrukce.

2.1 Základní konstrukce jazyka

Základní konstrukce jazyka VHDL vysvětlíme na příkladu návrhu úplné jednobitové sčítačky, která pracuje podle vztahů:

```
S = A xor B xor CI
COUT = (A and B) or (A and CI) or (B and CI)
```

kde A a B jsou sčítance, CI je vstupní přenos, S je součet a COUT je výstupní přenos.



Obr. 6. Úplná jednobitová sčítačka: (a) značka (rozhraní), (b) implementace

x+y

2.1.1 Nejjednodušší popis sčítačky

Nejjednodušší popis sčítačky je následující:

```
-- knihovny
library IEEE;
use IEEE.std_logic_1164.all;

-- popis rozhraní
entity FA is
  port (
    A, B, CI  : in  std_logic;
    S, COUT   : out std_logic);
end FA;

-- realizace rozhraní
architecture RTL of FA is
begin
  S <= A xor B xor CI;
  COUT <= (A and B) or (A and CI) or (B and CI);
end RTL;
```

Každý zdrojový kód ve VHDL začíná uvedením knihoven prvků, které budeme používat. Ve většině případů využijeme standardní knihovny IEEE, k jejichž popisu se dostaneme ještě později. Klíčové slovo **library** zpřístupňuje knihovnu IEEE a klausule **use** zpřístupňuje její dílčí části. Řádek s komentářem je označen --.

Dále následuje popis rozhraní navrhovaného obvodu. Za klíčovým slovem **entity** následuje název obvodu (FA) a v části **port** potom deklarace signálů rozhraní. Každý signál má přiřazen identifikátor (např. A), za dvojtečkou najdeme informaci o tom, zda se jedná o vstupní, výstupní či obousměrný vodič (např. **in** označuje vstupní vodič, podrobnosti si uvedeme dále) a typ vodič. Např. std_logic označuje běžný vodič, std_logic_vector potom svazek vodičů (např. sběrnici). Část entity se tedy chová jako černá skříňka, která má vstupy a výstupy, ale u které neznáme její obsah.

V části **architecture** se popisuje, jak má být rozhraní, uvedené v části **entity**, implementováno. Zde je důležité si uvědomit, že jedna entita může být implementována vícero způsoby (stejně jako při psaní programů můžeme různými způsoby implementovat funkci pro řazení). Popis implementace tedy začíná klíčovým slovem **architecture**, následuje zvolený název dané implementace (RTL v našem případě), klíčové slovo **of**, název entity, která se implementuje (FA v našem případě), a klíčové slovo **is**. Mezi **begin** a **end** zapisujeme zvolenou implementaci pomocí strukturního, behaviorálního nebo jiného popisu. Vše, co je uvedeno v této části, se vykonává „paralelně“ (v reálném HW také pracují všechny komponenty paralelně), tudíž nezávisí na pořadí zápisu jednotlivých operací. Před klíčovým slovem **begin** se uvádí deklarace použitých signálů, komponent apod. (což si vysvětlíme později).

V uvedeném příkladě není třeba definovat pomocné signály ani žádné jiné objekty. Vzhledem k tomu, že se jedná o kombinační obvod, provádí se pouze transformace vstupů na výstupy pomocí logických operátorů, popř. s využitím závorek. Všechny symboly uvedené v rozhraní jsou z pohledu VHDL tzv. signály, které budou mít při fyzické realizaci podobu vodičů a které mají daný směr toku dat.

Signál je fundamentální prostředek komunikace mezi jednotlivými částmi obvodu. Může být deklarován pouze v paralelním prostředí entity. Pomocí operátoru <= přiřadíme výstupnímu signálu (např. S) logickou hodnotu na základě výpočtu logického obvodu, který vznikne dle vztahu uvedeného na pravé straně výrazu (např. A xor B xor CI). Hodnota výrazu se přiřadí s určitým zpožděním signálu, specifikovaným identifikátorem na levé straně. Zpoždění uložení se dosáhne naplánováním zmíněné nové hodnoty pro určitou hodnotu modelového času. Toto zpoždění může být explicitně uvedeno. Pokud tomu tak není, aplikuje se implicitně tzv. **delta zpoždění**. Toto zpoždění má z hlediska modelového času nulovou hodnotu, ale vyjadřuje skutečnost, že k vlastnímu uložení hodnoty výrazu do daného signálu dojde až po vyhodnocení výrazů ve všech ostatních příkazech, které mají být provedeny v daném *modelovém čase*. Zmíněná strategie vytváří iluzi paralelního efektu vyhodnocování výrazů, které jsou ve skutečnosti prováděny sériově.

Pokud bychom uvedli do uvedeného zdrojového kódu do části **architecture** ještě třetí řádek, např.

```
S <= A xor B xor CI;
COUT <= (A and B) or (A and CI) or (B and CI);
S <= A or B;
```

potom nastanou problémy. Při simulaci bude hodnota S nedefinována. Není totiž fyzikálně možné, aby jeden výstupní vodič současně reprezentoval dva různé logické výrazy. Nezapomeňme, že všechny konstrukce uvedené v části architecture musí fungovat souběžně. Pokud bychom takový řádek dopsali např. do zdrojového kódu v jazyce C, potom by zřejmě nenastala chyba (pomineme-li jinou syntax) a nejdříve by se do proměnné S přiřadil výsledek prvního výrazu a potom výsledek z přidaného třetího řádku. Tedy na konci programu by S obsahoval výsledek odpovídající připsanému řádku. Můžeme říci, že každý paralelní signálový přiřazovací příkaz vytváří tzv. budič daného signálu a vytvořme si užitečnou představu, že budič, pokud není odpojen, budí daný signál neustále.

Uvedený styl popisu obvodu se nazývá data-flow popis. Je asi nejjednodušším způsobem, jak popisovat hardware, protože přímo vychází z logických rovnic. Jeho možnosti jsou však omezené.

2.1.2 Strukturní popis sčítačky

Druhou možností, jak popsat sčítačku, je využít čistě strukturní popis. Budeme se snažit „přepsat“ schéma zapojení sčítačky uvedené na Obr. 6 do textové podoby. Sčítačka sestává z komponent, logických členů, AND, OR a XOR. Nejdříve je třeba mít tyto logické členy k dispozici jako komponenty s jasně definovaným rozhraním. Ukažme si např. popis komponenty třívstupový OR, kterou nazveme OR3.

```
library IEEE;
use IEEE.std_logic_1164.all;
```

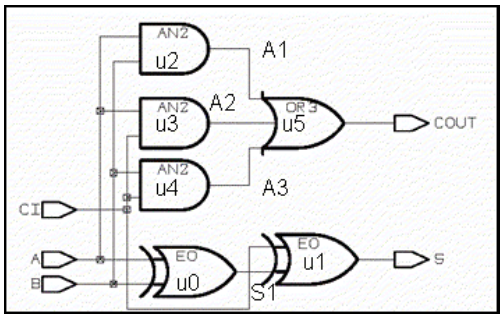
```
entity OR3 is
  port (
    A, B, C  : in  std_logic;
    Z  : out std_logic);
end OR3;

architecture DF of OR3 is
begin
  Z <= A or B or C;
end DF;
```

Je zřejmé, že způsob popisu OR vychází z popisu představeného v předchozí kapitole. Stejným způsobem popíšeme i ostatní potřebné komponenty, tj. dvouvstupový AND (AN2) a dvouvstupový XOR (EO). Předpokládejme, že jsou všechny tři soubory popisující AND, OR a XOR ve stejném adresáři jako je celkový popis sčítačky, který teď vytvoříme.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity FA is
  port(
    A, B, CI : in  std_logic;
    S, COUT  : out std_logic);
end FA;

architecture LIB of FA is
  component EO
    port( A, B : in std_logic;  Z : out std_logic);
  end component;

  component AN2
    port( A, B : in std_logic;  Z : out std_logic);
  end component;

  component OR3
    port( A, B, C : in std_logic;  Z : out std_logic);
  end component;

  signal S1, A1, A3, A2 : std_logic;

begin
  u0 : EO port map( A => A, B => B, Z => S1);
  u1 : EO port map( A => CI, B => S1, Z => S);
  u2 : AN2 port map( A => A, B => B, Z => A1);
  u3 : AN2 port map( A => A, B => CI, Z => A2);
  u4 : AN2 port map( A => B, B => CI, Z => A3);
  u5 : OR3 port map( A => A1, B => A2, C => A3, Z => COUT);
end LIB;
```

Nejdříve si všimněme, že popis rozhraní sčítačky je úplně stejný jako v předchozí kapitole. Protože se snažíme pouze vytvořit jiný způsob implementace sčítačky, liší se náš kód pouze v části **architecture**. Oproti předchozímu příkladu využijeme prostor před klíčovým slovem **begin** k tomu**,** abychom deklarovali komponenty, ze kterých složíme sčítačku, a signály (vodiče), kterými tyto komponenty propojíme.

Obr. 7. Úplná sčítačka s popisem komponent a signálů



Pomocí klíčového slova **component** tedy uvedeme rozhraní komponent EO, AN2 a OR3 (podobně jako uvádíme v běžných programovacích jazycích deklaraci funkce, jejíž implementace je provedena v jiném souboru). Dále deklarujeme čtyři pomocné signály A1, A2, A3 a S1.

Mezi **begin** a **end** bude následovat vytvoření instancí požadovaných komponent a jejich propojení, což jsme pro úplnost uvedli na Obr. 7. Na pořadí zápisu komponent nezáleží. Každá instance komponenty má název (např. u2), dále následuje dvojtečka, typ komponenty (např. AN2) a pomocí konstrukce **port map** je vytvořeno propojení dané komponenty se zbývajícími komponentami, vstupy, výstupy, popř. pomocnými signály. Např. instance u2 komponenty AN2 je připojena k primárním vstupům obvodu A a B a její výstup je připojen k pomocnému signálu A1, který bude následně přiveden do třívstupového členu OR (u5).

Ve srovnání s předchozím data flow popisem se zdá, že je tento způsob popisu číslicových obvodů výrazně složitější. Ano, v tomto případě je to pravda, protože uvedený obvod je velmi jednoduchý. Hlavní výhodou tohoto způsobu popisu je, že celý systém můžeme sestavit ze znovupoužitelných komponent, které si sami připravíme nebo nakoupíme od různých výrobců. Procesor se např. popisuje tak, že si připravíme hlavní komponenty, tj. ALU, řadič, sadu registrů, čítače, paměť atd. a tyto komponenty propojíme uvedeným způsobem. Další velkou výhodou strukturního popisu je, že tak, jak obvod popíšeme, tak bude také vysyntetizován. Vhodným návrhem můžeme tedy ovlivňovat strukturu a zpoždění obvodu.

2.1.3 Behaviorální popis sčítačky

Na stejném příkladu jednobitové úplné sčítačky si rovněž ukážeme behaviorální popis. Behaviorální popis je založen na použití tzv. procesů. Proces je prostředek pro popis chování obvodu nebo jeho části. Může obsahovat proměnné, řídicí struktury, ale i přiřazení signálů (<=). Pokud je proces aktivován, jsou jeho příkazy provedeny a to sekvenčně – tudíž záleží na jejich pořadí ve zdrojovém kódu.

```
architecture BEH of FA is
begin
  p1 : PROCESS(A, B, CIN)
   BEGIN
    IF (A='1' AND B='1') OR (A='1' AND CIN='1') OR
    (B ='1' AND CIN='1') THEN
       COUT <= '1';
    ELSE
       COUT <= '0';
    END IF;
    S <= A XOR B XOR CIN;
```

```
   END PROCESS;
end BEH;
```

Zápis procesu začíná nepovinným návěštím (zde p1), následuje dvojtečka, klíčové slovo **process** a v závorce může následovat seznam tzv. citlivých signálů. Proces je aktivován, pokud dojde ke změně na některém z uvedených signálů. V našem příkladě je zápis opět poněkud složitější než již diskutovaný dataflow popis, nicméně umožňuje demonstrovat několik důležitých jevů. Mezi **begin** a **end** procesu se nacházejí příkazy, které se vykonávají sekvenčně. Chování obvodu se popíše algoritmem. Jako jedna z možností je uvedeno využití konstrukce if-then-else pro výpočet výstupního přenosu. Poznamenejme, že do apostrofů ‘’ se ve VHDL zapisují literály, kterých může nabývat signál typu std_logic.

V rámci procesu se mohou vyskytovat proměnné, ve VHDL však mají oproti signálům význam spíše pomocný. Pro přiřazení výrazu do proměnné se využívá operátor „:=“.

Součet je zapsán pro změnu pomocí data flow popisu. V prostředí procesu však existence několika signálových přiřazovacích signálů, které přiřazují hodnoty témuž signálu, neznamená existenci několika různých budičů. V takovém případě budeme provedení příkazů chápat jako aktivitu jednoho a téhož budiče, který pro daný signál vytváří posloupnost časově oddělených hodnot. Tudíž připíšeme-li na konec procesu ještě přiřazení S <= A or B;, nebude se jednat o chybu (ani v rámci simulace), nicméně sčítačka nebude produkovat správný součet. Připomeňme znovu, že přiřazení hodnoty do signálu se děje s určitým zpožděním.

Procesů se může nacházet ve zdrojovém kódu více. Vzájemně se aktivují a pozastavují – tím je simulována činnost paralelního systému (obvodu) na sekvenčním počítači. Procesům se budeme více věnovat v dalších kapitolách.

2.1.4 Test Bench

Pro testování navržených komponent se nejčastěji používá tzv. testbench. Představuje testovací prostředí pro navrženou komponentu – generuje pro ni testovací signály a popř. zpracovává odezvy. Spuštěním vhodně navrženého testbenche získáme časové průběhy ze simulované komponenty.

Jedná se o kód ve VHDL, který nemá v části **entity** žádné signály. V části **architecture** se deklaruje komponenta, kterou chceme testovat, dále signály, jejichž prostřednictvím se připojíme k této komponentě, a popř. další signály a objekty VHDL. Dále v části **architecture** obvykle najdeme proces(y), který generuje test pro danou komponentu.

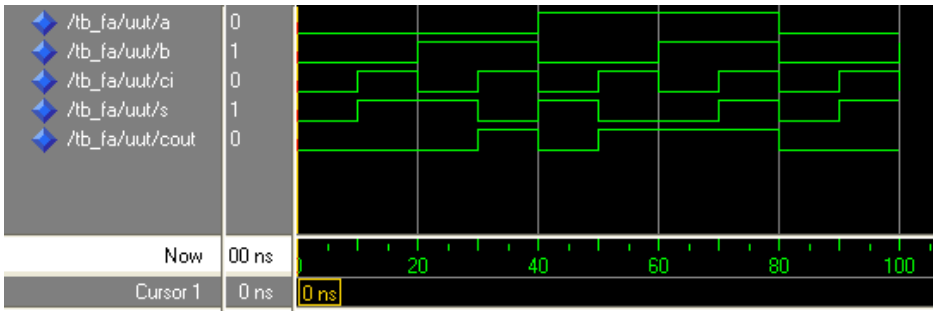
Následující testbench byl vytvořen pro sčítačku FA. Poznamenejme, že tento testbench je možné použít pro libovolnou implementaci FA (behaviorální, strukturní, …), která však splňuje požadované rozhraní. V části **architecture** jsou definovány přesně ty stejné signály, jako má rozhraní FA. Tyto signály jsou dále napojeny na rozhraní FA a pomocí těchto signálů jsou v rámci procesu zasílány testovací vektory pro FA (všechny možné vstupní kombinace v našem případě). Abychom mohli pohodlně sledovat všechny výstupy, vložíme po nastavení každé testovací vstupní kombinace příkaz `wait for 10 ns;`, který zajistí zpoždění 10 ns.

Abychom celou sčítačku odsimulovali v rámci prostředí ModelSim, musíme:

- Zkompilovat soubory fa.vhd (sčítačka) i tb_fa (testbench).
- Nastavit testbench jako soubor, který chceme simulovat (tzv. top level entity).
- Spustit simulační prostředí.
- Otevřít okno Wave se signály sčítačky.
- Příkazem run 100 ns zapsaným na příkazovou řádku spustit simulaci.


```
-- testbench pro FA (tb_fa.vhd)
library ieee;
use ieee.std_logic_1164.all;

entity tb_fa is
end tb_fa;

architecture arch_tb_fa of tb_fa is

-- pomoci techto signalu se pripojime k testovane jednotce
  signal A, B, CI: std_logic;
  signal S, COUT : std_logic;

-- tuto jednotku budeme testovat ("deklarace")

component FA port (
   A, B, CI  : in  std_logic;
   S, COUT  : out std_logic);
end component;


begin

-- "instance" testovane jednotky je pripojena pomoci zavedenych signalu
UUT : FA
port map (
       A => A, B => B, CI => CI,
       S => S,      COUT => COUT);

-- vlastni testovani zkousi vsechny kombinace na vstupech
process
begin
  A <= '0'; B <= '0'; CI <= '0';
  wait for 10 ns;
  A <= '0'; B <= '0'; CI <= '1';
  wait for 10 ns;
  A <= '0'; B <= '1'; CI <= '0';
  wait for 10 ns;
  A <= '0'; B <= '1'; CI <= '1';
  wait for 10 ns;
  A <= '1'; B <= '0'; CI <= '0';
  wait for 10 ns;
  A <= '1'; B <= '0'; CI <= '1';
  wait for 10 ns;
  A <= '1'; B <= '1'; CI <= '0';
  wait for 10 ns;
  A <= '1'; B <= '1'; CI <= '1';
  wait for 10 ns;
end process;
end arch_tb_fa;
```

Obr. 8. Simulace sčítačky



2.2 Vybrané konstrukce jazyka VHDL

V této podkapitole jsou uvedeny základní konstrukce jazyka VHDL, které budeme potřebovat v rámci povinných kurzů na FIT VUT v Brně.

2.2.1 Použití knihoven

Ve všech našich příkladech budeme používat knihovnu `std_logic_1164`, která umožňuje tzv. *vícehodnotovou simulaci*. Při vícehodnotové simulaci nepracujeme pouze s logickými hodnotami 0 a 1, ale pracujeme s více hodnotami, které nám umožní vyjádřit sílu (tvrdost) signálu a tím zefektivnit proces návrhu, kdy snadněji odhalíme potenciální chyby. Signály tedy budou nabývat hodnot nejen 0 a 1, ale i X (neznámá hodnota, konflikt), L (slabší 0), H (slabší 1), U (počáteční neinicializovaná hodnota), Z (vysoká impedance), W (neznámá hodnota), - (don't care) apod.

Pro signály budeme používat typ `std_logic` (jeden vodič) a `std_logic_vector` (skupina vodičů, např. sběrnice), který je svázán s tzv. rezoluční funkcí umožňující určit výslednou hodnotu i v takových případech, kdy je signál buzen log. 0 i log. 1 současně (což v reálném obvodu vede ke zkratu). Simulátor ohlásí výsledek jako neznámou hodnotu (X). Následující tabulka definuje výsledné hodnoty pro všechny kombinace hodnot signálů.

	U	X	0	1	Z	W	L	H	-
U	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
X	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Knihovna `ieee.std_logic_unsigned.all`; definuje aritmetické operátory nad binárními vektory pro operace sčítání (+), odčítání (-) a násobení (*).

Knihovna `ieee.std_logic_arith.all` umožňuje pracovat s čísly se znaménkem i bez znaménka tak, že zavádí dva nové typy `signed` a `unsigned`. Binární hodnotu *h* datového typu std_logic_vector můžeme zkonvertovat na typ unsigned zápisem `unsigned (h)` a na typ signed zápisem `signed(h)`. Pomocí

funkce `conv_integer` je možné provádět konverze mezi hodnotami typu integer a std_logic_vector, např.

```
signal c : std_logic_vector (3 downto 0);
signal ci : integer;
c <= "1000";
ci <= conv_integer(signed(c)); -- v ci bude -8
```

2.2.2 Konstanty

Syntax deklarace a příklady deklarace konstanty je následující:

```
CONSTANT constant_name : type_name [:= value];

CONSTANT PI : REAL := 3.14;
CONSTANT SPEED : INTEGER;
```

2.2.3 Proměnné

Proměnné jsou určeny pro lokální uložení dat. Pro přiřazení do proměnné se používá symbol :=. Hodnota proměnné se mění **ihned** po přiřazení. Syntax deklarace proměnné:

```
VARIABLE variable_name : type_name [:= value];
```

Příklad deklarace:

```
VARIABLE opcode : STD_LOGIC_VECTOR (3 DOWNTO 0) := "0000";
VARIABLE freq : INTEGER;
```

2.2.4 Signály

Jak již bylo řečeno, signály slouží pro komunikaci mezi VHDL moduly a typicky přímo odpovídají existenci fyzického vodiče. Syntax deklarace signálu je následující:

```
SIGNAL signal_name : type_name [:= value];
```

Příklady deklarace:

```
signal din: STD_LOGIC_VECTOR (7 downto 0);
signal   a: STD_LOGIC_VECTOR (0 to 2);
signal   y: STD_LOGIC;
```

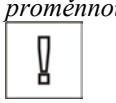
U `std_logic_vector` je uvedena bitová šířka a významnost bitů. Zápis (7 downto 0) znamená, že bit 7 je MSB a bit 0 je LSB. Zápis (0 to 2) znamená, že bit 0 je LSB a bit 2 je MSB. K jednotlivým bitům signálu o typu `std_logic_vector` je možné přistupovat pomocí kulatých závorek, např. din(0) je hodnota nejnižšího bitu din a din(7) je hodnota nejvyššího bitu din.

Jak již bylo uvedeno, oproti proměnným nedochází u signálů k bezprostřední změně hodnoty. Hodnota je přiřazena (pomocí operátoru <=) až s určitým zpožděním (tzv. zpožděním *delta*), které umožňuje realizovat simulaci paralelismu. Hodnota signálu je přiřazena až v okamžiku provedení všech (částí) procesů, které byly naplánovány k provedení právě pro danou hodnotu modelového času. Jinak řečeno, všechny procesy nejprve provedou vlastní

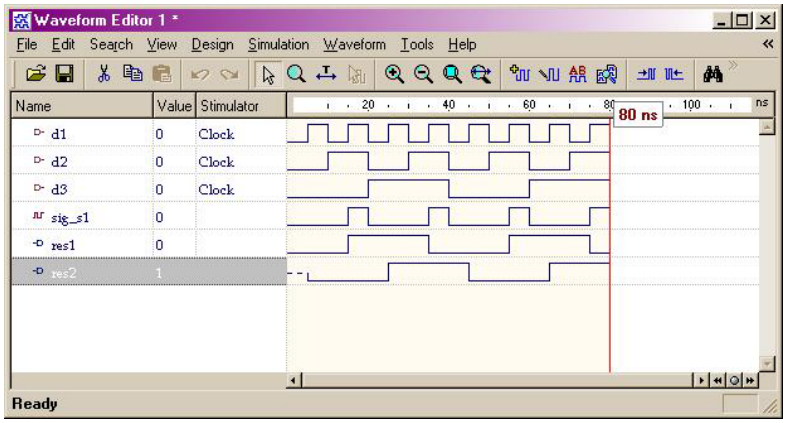
výpočty (bez uložení hodnot do signálů) a teprve pak se modifikují všechny změněné hodnoty. Tento způsob výpočtu eliminuje vliv pořadí, ve kterém jsou procesy prováděny.

Následující kód, který popisuje jednoduchý kombinační obvod se dvěma výstupy, a následující obrázek ilustrují efekt delta zpoždění. Chování každého výstupu je popsáno jedním procesem. Sledujte výsledné hodnoty signálů res1 a res2.

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

entity sig_var is
port(d1, d2, d3:    in std_logic;
     res1, res2:    out std_logic);
end sig_var;

architecture behv of sig_var is

 signal sig_s1: std_logic;
 begin

  proc1: process(d1,d2,d3)
    variable var_s1: std_logic;
  begin
       var_s1 := d1 and d2;
       res1 <= var_s1 xor d3;
  end process;

  proc2: process(d1,d2,d3)
  begin
       sig_s1 <= d1 and d2;
       res2 <= sig_s1 xor d3;
  end process;
end behv;
```



Obr. 9. Ilustrace rozdílu mezi signálem a proměnnou

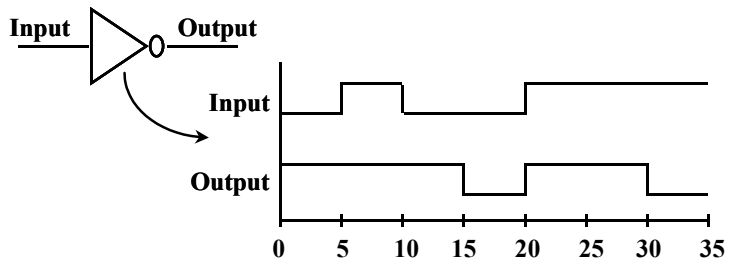


Každý příkaz přiřazení signálu předepisuje čas (zpoždění), který uplyne než signál nabude nové hodnoty. Kromě již zmiňovaného delta zpoždění existuje ještě transportní zpoždění a inertní zpoždění.

Transpotrní zpoždění musí být explicitně předepsáno klíčovým slovem **transport**. Typicky modeluje zpoždění přenosu signálu hradlem. Následující obrázek ukazuje vliv konstrukce:

```
Output <= TRANSPORT NOT Input AFTER 10 ns;
```

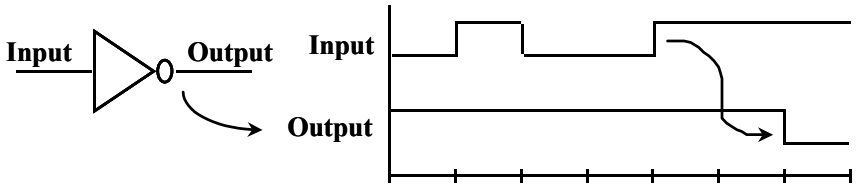
Obr. 10. Transportní zpoždění



Inertní zpoždění specifikuje přenosové zpoždění a navíc i minimální šířku pulsu. Následující obrázek demonstruje vliv konstrukce:

```
output <= REJECT 5ns INERTIAL NOT input AFTER 10ns;
```

Obr. 11. Inertní zpoždění



2.2.5 Porty entity

Porty jsou speciální signály, se kterými lze provádět omezené operace čtení a zápisu. Kromě své funkce rozhraní entity mohou být použity jako lokální signály v rámci příslušné architektury. Port je charakterizován názvem, módem činnosti a datovým typem. Dle módu činnosti rozlišujeme porty:

- `In` – vstupní port
- `Out` - výstupní port
- `Inout` – vstupně výstupní port, který umožňuje obousměrnou komunikaci a používá se např. pro realizaci sběrnice.
- `Buffer` – je v podstatě výstupní port, ale jeho obsah je možné číst.
- `Linkage` – směr toku dat neznámý.

Příklad:

```
port (d: inout STD_LOGIC_VECTOR (7 downto 0);
      a: in STD_LOGIC_VECTOR (0 to 2);
      y: out STD_LOGIC);
```

Uvedené rozhraní obsahuje osmibitový obousměrný port *d*, který může např. reprezentovat sběrnici, tříbitový vstup *a* a jednoduchý výstup *y*.

2.2.6 Datové typy

VHDL podporuje následující datové typy: skalární (integer, real apod.), složené (pole, záznam), přístupový typ (ukazatel) a typ soubor.

2.2.6.1 Real a integer

Datové typy real a integer je obdobné jako u jiných programovacích jazyků. Proměnné těchto typů se používají v procesech. Následující proces ukazuje různé korektní i nekorektní použití proměnné typu real.

```
ARCHITECTURE test_real OF test IS
BEGIN
       PROCESS (X)
          VARIABLE a: REAL;
       BEGIN
          a := 1.3;  -- OK
          a := -7.5;  -- OK
          a := 1;  -- chyba
          a := 1.7E13;  -- OK
          a := 5.3 ns;  -- chyba
       END PROCESS;
END test_real;
```

Příklad deklarací konstant, proměnných a signálů (i s inicializací, inicializace je implicitně na hodnotu 0 pro proměnné typu integer, ‘0’ pro znaky):

```
constant pi: real := 3.14;
constant rychlost: integer; -- implicitně 0
variable pom: integer;
variable suma : std_logic_vector (0 to 3) := "0001";
signal s1 : std_logic;
```

Všimněme si, jakým způsobem se zapisují literály. Pro `std_logic_vector` je to pomocí uvozovek, pro `std_logic` je to pomocí apostrofů (s1 <= ‘0’).
Zápis hodnot v různých číselných soustavách je možný dle následujícího systému, který je vysvětlen na příkladech:
Pro typy integer, real apod:

- Dekadická soustava: 0, 123, 6.76, 10E4
- Oktalová soustava: 8#730#
- Hexadecimální soustava: 16#FFFF#
- Binární soustava: 2#1001#

Pro `std_logic_vector`:

- Binární soustava: “000110101”
- Hexadecimální soustava: X“F6“
- Oktalová soustava: O“77“

2.2.6.2 Typ - fyzikální veličina

Ve VHDL je možné definovat datový typ, který odpovídá fyzikální veličině. Standardně je předdefinovaný *fyzický typ* pro čas (time). Jako příklad je uvedena definice typu pro vyjádření odporu. Vždy musí být uvedeny jednotky.

```
TYPE resistance IS RANGE 0 TO 10000000

UNITS
ohm;  -- ohm
Kohm = 1000 ohm;  -- i.e. 1 KΩ
Mohm = 1000 kohm;  -- i.e. 1 MΩ
END UNITS;
```

2.2.6.3 Výčtový typ

Příklad deklarace nového výčtového typu, který můžeme využít např. při implementaci automatu:

```
-- Vytvoření nového výčtového typu muj_stav
TYPE muj_stav IS (reset, idle, rw, io);
```

```
-- Vytvoření proměnné typu muj_stav
signal stav: muj_stav;

-- Do proměnné lze přiřadit pouze hodnotu daného typu
stav <= reset; -- nelze psát, např. stav <="00";
```

2.2.6.4 Podtypy

Je možné definovat podtyp nějakého typu. Např. definujme podtyp FIRST_TEN jako podtyp integeru:

```
SUBTYPE first_ten IS INTEGER RANGE 0 TO 9;
```

Obecně je syntaxe definice podtypu následující:

```
SUBTYPE name IS base_type RANGE <user range>;
```

2.2.6.5 Atributy

Skalární datové typy disponují tzv. atributy. Pomocí notace s apostrofem lze tak získat informaci o předchůdci (pred), následníkovi (succ), maximální (high) a minimální hodnotě (low) rozsahu a další důležité údaje. Předpokládejme, že *pom* je proměnná typu *integer*. Potom zápisem `pom'pred(4)` lze získat hodnotu předchůdce (což je hodnota 3).

Signály disponují řadou specifických atributů (active, last_active, last_event, last_value atd.). Pokud např. provádíme test na to, zda došlo k náběžné hraně hodinového signálu *clk*, potom využijeme atribut *event* (který vrací true, pokud došlo ke změně hodnoty signálu) a píšeme

```
if (clk'event) and (clk = 1) then …
```

2.2.6.6 Pole

Datový typ pole použijeme nejčastěji při implementaci paměťových struktur, např. deklarace

```
type ram_typ is array(0 to 63) of STD_LOGIC_VECTOR(15 downto 0);
```

vytváří datový typ popisující paměť o 64 šestnáctibitových slov. Vlastní paměť vytvoříme jako proměnnou typu ram_typ

```
signal ram: ram_typ;
```

do paměti se zapisuje např, takto:

```
ram(0)  <= x"1a0f";
```

2.2.7 Operátory

2.2.7.1 Logické operátory

and, or, nand, nor, xor, nxor, not

2.2.7.2 Relační operátory

=, /= (test na nerovnost), >, <, <=, >=

2.2.7.3 Aritmetické operátory

Binární: +, -, *, /, rem, mod, ** (mocnina)
Operátory rem i mod produkují zbytek po celočíselném dělení (výsledek `a mod b` má znaménko shodné s `b`, `a rem b` má znaménko shodné s `a`)
Unární: +, -, abs

2.2.7.4 Posuvy a rotace

Posuvy: SLL, SRL (logický), SLA, SRA (aritmetický – zachovává nejnižší bit)
Rotace: ROL (vlevo), ROR (vpravo)

Příklad: `c <= c ror 3; -- rotace c o tři bity vpravo`

2.2.7.5 Konkatenace

Operátor konkatenace & slouží pro sdružování vodičů, rozšiřování operandů a realizaci posuvů. Mějme deklarace signálů:

```
signal a, b: std_logic_vector (3 downto 0);
signal c: std_logic_vector (7 downto 0);
```

Předpokládejme, že v *a* je hodnota "0011" a v *b* je hodnota "1111". Potom můžeme provést sloučení *a* s *b* například takto:

```
c <= a & b; -- konkatenace, v c bude „00111111"
```

Pomocí konkatenace je možné realizovat i rotace, např. o jeden bit vlevo.

```
a <= a(2 downto 0) & a(3); -- „0011" -> "0110"
```

Pomocí závorek je možné pohodlně pracovat s částmi vektoru:

```
b(3 downto 2) <= "01"; -- práce s horními dvěma bity b
```

Pomocí závorek můžeme agregovat několik signálů do jednoho:

```
c <= ('0', '1', others=>'0'); -- agregace
```

Výsledek bude „01000000", kde klíčové slovo **others** nahrazuje zbylé bity vektoru.

2.2.7.6 Priorita operátorů

1. **, ABS, NOT
2. *, /, MOD, REM
3. Unární +, -
4. +, -, &
5. SLL, SRL, SLA, SRA, ROL, ROR
6. =, /=, <, <=, >, >=
7. AND, OR, NAND, NOR, XOR, XNOR

2.2.8 Proces a příkazy v něm

Proces je dílčí sekvenční část simulačního programu. Jeho syntax je uvedena zde:

```
process [(sensitivity_list)]
  [subprogram_decl|subprogram_body]
  [type_decl]
  [subtype_decl]
  [constant_decl]
  [variable_decl]
  [file_decl]
  [alias_decl]
  [attribute_decl]
  [attribute_spec]
  [use_clause]
begin
  [sequential_statements]
end process [proc_label];
```

V rámci procesů je kromě přiřazení do proměnné a signálu možné používat příkazy:

2.2.8.1 If

Syntax:

```
if condition then
  sequential_statements
{elsif condition then
  sequential_statements}
[else
  sequential_statements]
end if;
```

2.2.8.2 Case

Syntax:

```
case expression is
  {when choices => sequential_statements}
end case;
```

Volba možnosti musí vypadat takto:

```
value => sequential_statements   -- pro jednu hodnotu
value1 | value2 ... => sequential_statements  -- pro více hodnot
value1 to value2 => sequential_statements   -- pro určitý rozsah
others => sequential_statements   -- pro ostatní
```

Všechny možné hodnoty musí být podchyceny, jinak nastanou problémy při syntéze. Nejčastěji se používá konstrukce s klíčovým slovem **others**.

Příklad: dekodér pro sedmisegmentovku (zapojení jednotlivých segmentů jistě vytvoříte sami).

```
case BCD is
  when "0000" => LED := "1111110";
  when "0001" => LED := "1100000";
  when "0010" => LED := "1011011";
  when "0011" => LED := "1110011";
  when "0100" => LED := "1100101";
  when "0101" => LED := "0110111";
  when "0110" => LED := "0111111";
  when "0111" => LED := "1100010";
  when "1000" => LED := "1111111";
  when "1001" => LED := "1110111";
```

```
  when others => LED := "-------";   -- don't care
end case;
```

2.2.8.3 Loop

Umožňuje opakovat sekvenci příkazů

Syntax:

```
[label:] while condition loop
    ... sequence of statements ...
    end loop label;

[label:]  for loop_variable in range loop
  ... sequence of statements...
   end loop label;
```

Ukončení současné iterace.

```
next [loop_label][when condition];
```

Opuštění cyklu:

```
exit [loop_label][when condition];
```

2.2.8.4 Assert

Příkaz `assert` dovoluje v průběhu činnosti programu otestovat určitou kritickou podmínku a následně eventuelně ukončit činnost simulace.

Syntax:

```
assert condition
  [report string_expr]
  [severity failure|error|warning|note];
```

Příklad: Ukončení simulace, pokud je X > 3.

```
process (CLK, DIN)                behavior of a D-FF
  variable X: integer;
  ...
begin
  ...
  assert (X > 3)
    report "setup violation"
    severity warning;
  ...
end process;
```

2.2.8.5 Wait

Dynamicky řídí spouštění a pozastavování procesů. Pokud je tento příkaz použit v rámci procesu, pak proces nesmí obsahovat citlivostní seznam.

Syntax:

```
wait
  [on signal_name {, signal_name}]
  [until condition]
  [for time_expr];
```

2.2.9 Paralelní příkazy

V části `architecture` můžeme kromě procesů, přiřazení signálu a instance

komponenty použít i další příkazy. Nejčastější jsou výběrové přiřazení signálu a podmíněné přiřazení signálu.

2.2.9.1 Výběrové přiřazení signálu

Syntax:

```
with expression select
  signal_name <=expression when value{,
               expression when value};
```

Příklad: Na základě hodnoty MYSEL přiřadí do Z buď hodnotu z A, B nebo C.

```
with MYSEL select
  Z <=  A when 15,
        B when 22,
        C when 28;
```

2.2.9.2 Podmíněné přiřazení signálu

Syntax:

```
signal_name <=expression when condition else
                   {expression when condition else}
                   expression;
```

Příklad:

```
Z <=  A when (X > 3) else
      B when (X < 3) else
      C;
```

2.2.10 Generická komponenta a příkaz generate

Pomocí příkazu `generate` lze jednoduše vygenerovat obvody s pravidelnou strukturou. Tento příkaz se nachází mimo proces a počet opakování nebo podmínka musí být konstantní!! Pro generování pravidelných struktur používáme příkaz:

```
for int_var in min to max generate
```

Pro podmíněné generování prvků použijeme:

```
if bool_expr generate
```

Příklad: Předpokládejme, že máme k dispozici komponentu invertor INV. Pomocí příkazu generate může vygenerovat 4-bitový invertor takto.

```
entity inv_n is
generic (len_c: integer := 4));   -- počet invertorů
port (I : in std_logic_vector (len_c-1 downto 0);
      O : out std_logic_vector (len_c-1 downto 0));
end inv_n;

architecture structural of inv_n is
component INV port (I : in std_logic; O : out std_logic);
end component;

begin
  g1: for i in 0 to len_c-1 generate
      inv: INV port map
             (I=>I(i),O=>O(i));
      end generate;
end structural;
```

Pro n = 4 je uvedená konstrukce ekvivalentní zápisu:

```
inv_0: INV port map (I=>I(0),O=>O(0));
inv_1: INV port map (I=>I(1),O=>O(1));
inv_2: INV port map (I=>I(2),O=>O(2));
inv_3: INV port map (I=>I(3),O=>O(3));
```

2.2.11 Poznámky k syntéze

Pokud jsou nepokryté případy v příkazech **if** nebo **case**, snaží se syntezátor zachovat v nepokrytém případě předchozí stav. Proto syntezátor vygeneruje záchytné registry registry (latch). Ty jsou většinou nežádoucí, protože způsobují zpoždění a tím mohou potenciálně snížit maximální frekvenci obvodu. Proto se vždy snažíme pokrýt všechny možnosti v uvedených příkazech.

Při syntéze jsou signály uvedené v seznamu signálů, na které je proces citlivý, ignorovány, což ovšem neplatí pro behaviorální simulaci. Pokud se stane, že v tomto seznamu chybí nějaký signál přímo ovlivňující chování procesu, pak simulace po syntéze a před syntézou budou dávat jiné výsledky. Proto je třeba vždy zkontrolovat, zda jsme do seznamu zapsali všechny relevantní signály.

Příkazy **loop** a **generate** mohou mít pro syntézu pouze statický rozsah, který je nejlépe definován konstantou typu integer. Obecně lze říct, že příkaz **loop** není pro učely syntézy vhodný.

Příkazy uvnitř procesu jsou při behaviorální simulaci zpracovávány sekvenčně. V případě syntézy jsou všechny příkazy v procesu vykonány paralelně. To se projeví zejména při přiřazování hodnot nebo výrazů do proměnných. Pozor, může dojít k nežádoucímu chování.

Při psaní kódu je potřeba mít na paměti omezení, která jsou dána cílovou technologií (FPGA, CPLD, atd). Nemá smysl ve VHDL předepsat použití třístavového budiče, pokud ho cílová součástka nemá.

2.3 Příklady popisu základních číslicových obvodů

V této kapitole popíšeme elementární číslicové obvody pomocí VHDL.

2.3.1 Multiplexor

Behaviorální popis čtyřvstupového multiplexoru. V uvedených příkladech označují signály I0, I1, I2 a I3 datové vstupy multiplexoru a S je řídicí vstup multiplexoru.

```
process (S, I0, I1, I2, I3)
begin
       case S is
              when "00" => O <= I0;
              when "01" => O <= I1;
              when "10" => O <= I2;
              when "11" => O <= I3;
              when others => O <= 'X';
       end case;
end process;
```

Ukázka dvou různých popisů multiplexoru, který pracuje s tříbitovými signály.

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

entity Mux is
port(  I3:    in std_logic_vector(2 downto 0);
       I2:    in std_logic_vector(2 downto 0);
       I1:    in std_logic_vector(2 downto 0);
       I0:    in std_logic_vector(2 downto 0);
       S:     in std_logic_vector(1 downto 0);
       O:     out std_logic_vector(2 downto 0)
);
end Mux;

-- varianta 1
architecture behv1 of Mux is
begin
    process(I3,I2,I1,I0,S)
    begin

        -- příkaz case
        case S is
           when "00" =>    O <= I0;
           when "01" =>    O <= I1;
           when "10" =>    O <= I2;
           when "11" =>    O <= I3;
           when others =>  O <= "XXX";
       end case;
    end process;
end behv1;

-- varianta 2
architecture behv2 of Mux is
begin
    -- příkaz when.. else
    O <=     I0 when S="00" else
             I1 when S="01" else
             I2 when S="10" else
             I3 when S="11" else
             "XXX";
end behv2;
```

2.3.2 Dekodér

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dec3to8 is
    port (
        addr: in STD_LOGIC_VECTOR (2 downto 0);
        y: out STD_LOGIC_VECTOR (7 downto 0)
    );
end dec3to8;

architecture dec3to8 of dec3to8 is
begin
    with addr select
     y   <=  "10000000" when "111",
             "01000000" when "110",
             "00100000" when "101",
             "00010000" when "100",
             "00001000" when "011",
             "00000100" when "010",
             "00000010" when "001",
             "00000001" when others;
end dec3to8;
```

Alternativní zápis pomocí konstrukce process

```
process(addr)
        begin
           y <= "00000000";
           case addr is
              when "000" => y <= "00000001";
              when "001" => y <= "00000010";
              when "010" => y <= "00000100";
              when "011" => y <= "00001000";
              when "100" => y <= "00010000";
              when "101" => y <= "00100000";
              when "110" => y <= "01000000";
              when "111" => y <= "10000000";
              when others => null;
           end case;
        end process;
```

2.3.3 ALU

Je uveden příklad behaviorálního popisu dvoubitové ALU, která realizuje sčítání, odčítání, logický součet a součin. ALU pracuje v doplňkovém kódu.

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


entity ALU is

port(  A:     in std_logic_vector(1 downto 0);
       B:     in std_logic_vector(1 downto 0);
       Sel:   in std_logic_vector(1 downto 0);
       Res:   out std_logic_vector(1 downto 0)
);

end ALU;

architecture behv of ALU is
begin

    process(A,B,Sel)
    begin
       case Sel is
           when "00" =>
              Res <= A + B;
           when "01" =>
               Res <= A + (not B) + 1;
            when "10" =>
              Res <= A and B;
           when "11" =>
              Res <= A or B;
           when others =>
              Res <= "XX";
         end case;

    end process;

end behv;
```

Nakreslete obvodou realizaci uvedené ALU.

2.3.4 Sčítačky

Uvedené příklady slouží spíše pro demontraci možností VHDL, než aby popisovaly reálnou implementaci sčítaček pro FPGA. Obvykle stačí použít operátor + a nástroj pro syntézu již vygeneruje odpovídající obvod automaticky.

2.3.4.1 Generický dataflow popis sčítačky

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

entity ADDER is
generic(n: natural := 2); -- počet bitů
port(  A:     in std_logic_vector(n-1 downto 0);
       B:     in std_logic_vector(n-1 downto 0);
       carry: out std_logic;
       sum:   out std_logic_vector(n-1 downto 0)
);
end ADDER;

architecture behv of ADDER is
-- pomocný signál pro uchování výsledku
signal result: std_logic_vector(n downto 0);
begin
     result <= ('0' & A)+('0' & B);  -- sečti, zajisti nepřetečení
    sum <= result(n-1 downto 0); -- ulož výsledek
    carry <= result(n); -- ulož přenos

end behv;
```

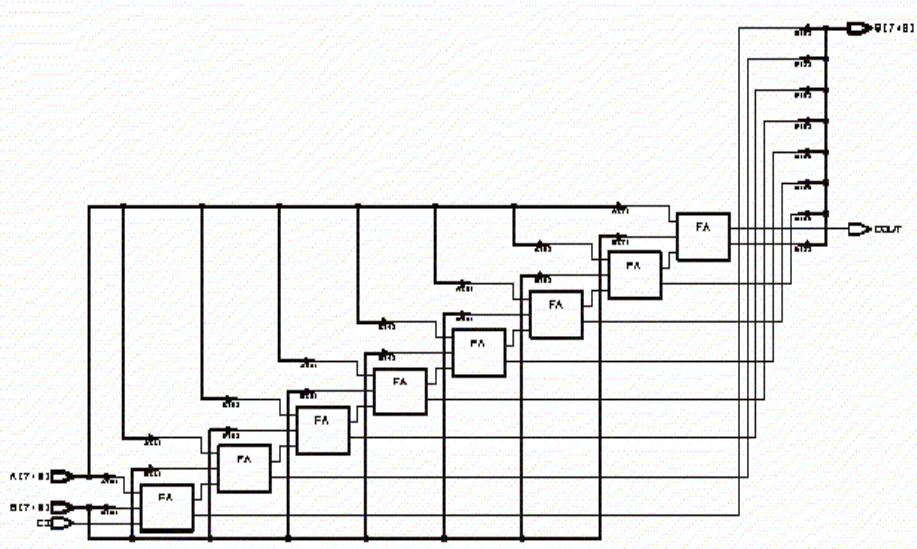
2.3.4.2 Sčítačka s postupným přenosem

Následuje generický popis sčítačky s postupným přenosem. Jako základní komponenta se využívá úplná sčítačka FA definovaná dříve. Proces generování zapojení je rozdělen do tří etap: generování FA pro nejnižší bit, generování FA pro prostřední bity a generování FA pro nejvyšší bit. Označení signálů je stejné jako u úplné sčítačky.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity RPADDER is
  generic (N : in integer := 8); -- počet bitů
  port (
    A, B  : in  std_logic_vector(N-1 downto 0);
    CI    : in  std_logic;
    S     : out std_logic_vector(N-1 downto 0);
    COUT  : out std_logic);
end RPADDER;

architecture RTL1 of RPADDER is
  component FA
  port (
    A, B, CI : in  std_logic;
    S, COUT  : out std_logic);
  end component;
  signal C : std_logic_vector(A'length-1 downto 1);

begin
  gen : for j in A'range generate
```

```
    genlsb : if j = 0 generate
      fa0 : FA port map (A => A(0), B => B(0),
        CI => CI, S => S(0), COUT => C(1));
    end generate;

    genmid : if (j > 0) and (j < A'length-1) generate
      fa0 : FA port map (A => A(j), B => B(j),
        CI => C(j), S => S(j), COUT => C(j+1));
    end generate;

    genmsb : if j = A'length-1 generate
      fa0 : FA port map (A => A(j), B => B(j),
        CI => C(j), S => S(j), COUT => COUT);
    end generate;
  end generate;
end RTL1;
```

Obr. 12. Sčítačka s postupným přenosem – výsledek syntézy



Odvoďte zpoždění a počet hradel n-bitové sčítačky s postupným přenosem.

2.3.5 Násobičky

2.3.5.1 Násobička – behaviorální popis

Uvedený příklad představuje behaviorální popis dvoubitové násobičky, který je založen na algorimu posuň a přičti.

```
entity multiplier is
port(  num1, num2: in std_logic_vector(1 downto 0);
       product:     out std_logic_vector(3 downto 0)
);
end multiplier;

architecture behv of multiplier is
begin
process(num1, num2)
  -- pomocné proměnné s rozšířeným počtem bitů
  variable num1_reg: std_logic_vector(2 downto 0);
  variable product_reg: std_logic_vector(5 downto 0);
```

```
begin
  num1_reg := '0' & num1;
  product_reg := "0000" & num2;

  for i in 1 to 3 loop
    if product_reg(0)='1' then
        product_reg(5 downto 3) := product_reg(5 downto 3)
        + num1_reg(2 downto 0);
      end if;
      product_reg(5 downto 0) := '0' & product_reg(5 downto 1);
  end loop;

  -- vložení výsledku do výstupního signálu
  product <= product_reg(3 downto 0);

end process;

end behv;
```

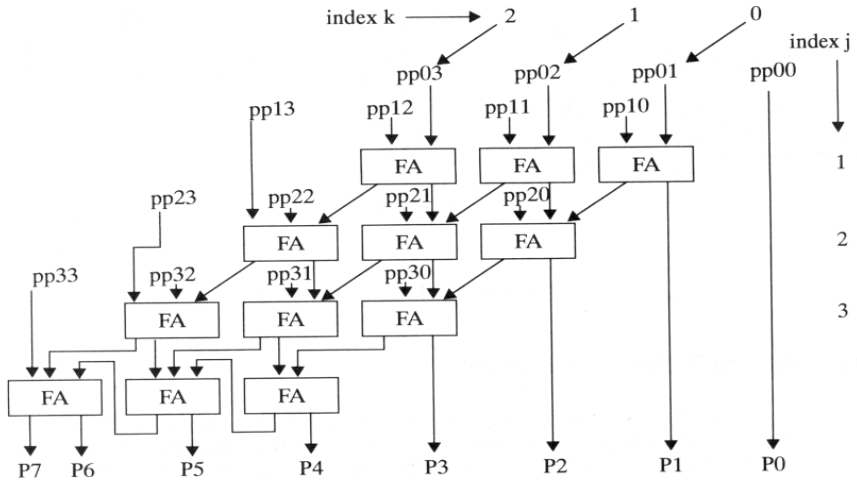
2.3.5.2 Násobička s uchováním přenosu

Jedná se generický popis implementace uvedené na Obr. 13. Význam symbolů je následující: PP - partial product, PC - partial carry, PS-partial sum, j-číslo řádku, k – číslo sloupce.

```
entity MULT is
  port (
    A, B : in  std_logic_vector(3 downto 0);
    PROD : out std_logic_vector(7 downto 0));
end MULT;

architecture RTL1 of MULT is
  constant N : integer := 4;
  subtype p1ary is std_logic_vector(N-1 downto 0);
  type pary is array(0 to N) of p1ary;
  signal PP, PC, PS : pary;

  component FA -- full adder
  port (
    A, B, CI : in  std_logic;
    S, COUT  : out std_logic);
  end component;

component ANDG -- and gate
  port (
    A, B : in  std_logic;
    C    : out std_logic);
  end component;

begin
  pgen : for j in 0 to N-1 generate
    pgen1 : for k in 0 to N-1 generate
      and0 : ANDG port map (
        A => A(k), B => B(j), C => PP(j)(k));
    end generate;
    PC(0)(j) <= '0';
  end generate;
  PS(0) <= PP(0);
  PROD(0) <= PP(0)(0);  -- nultý bit výsledku

-- Sčítačky řádků 1..N-1
  addr : for j in 1 to N-1 generate
    addc : for k in 0 to N-2 generate
      fa0 : FA port map (
        A => PP(j)(k), B => PS(j-1)(k+1), CI => PC(j-1)(k),
```

```
        S => PS(j)(k), COUT => PC(j)(k));
    end generate;
    PROD(j)    <= PS(j)(0);
    PS(j)(N-1) <= PP(j)(N-1);
  end generate;
  PC(N)(0) <= '0';

--Sčítačky v posledním řádku
addlast : for k in 1 to N-1 generate
    fa1 : FA port map (
      A => PS(N-1)(k), B => PC(N-1)(k-1), CI => PC(N)(k-1),
      S => PS(N)(k), COUT => PC(N)(k));
  end generate;
  PROD(2*N-1) <= PC(N)(N-1);
  PROD(2*N-2 downto N) <= PS(N)(N-1 downto 1);
end RTL1;
```

Obr. 13. Násobička s uchováním přenosu



Odvoďte zpoždění násobičky s postupným přenosem.

2.3.6 Klopné obvody (KO)

Klopné obvody, čítače a registry se ve většině případů popisují behaviorálně. Ukážeme si několik typických příkladů.

2.3.6.1 KO D

```
library IEEE;
use IEEE.std_logic_1164.all;

entity dffx is
      port (
            CLK   : in  std_logic;
            RSTn  : in  std_logic;
            DATA  : in  std_logic;
            QOUT  : out std_logic );
end dffx;

architecture behavr1 of dffx is
begin
process (CLK,RSTn)
begin
      if (RSTn='0') then  -- asynchronni reset
```

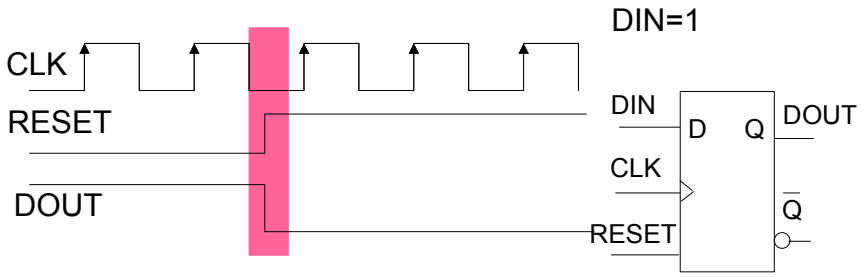
```
                QOUT <= '0';
        elsif (CLK'event and CLK = '1') then
                QOUT <= DATA;
        end if;
end process;
end  behavr1;
```

x+y

2.3.6.2 KO D s asynchronním resetem

```
process (CLK, RESET)
begin
        if RESET='1' then   --asynchronní RESET aktivní při log. 1
                DOUT <= '0';
        elsif (CLK'event and CLK='1') then  --CLK rising edge
                DOUT <= DIN;
        end if;
end process;
```

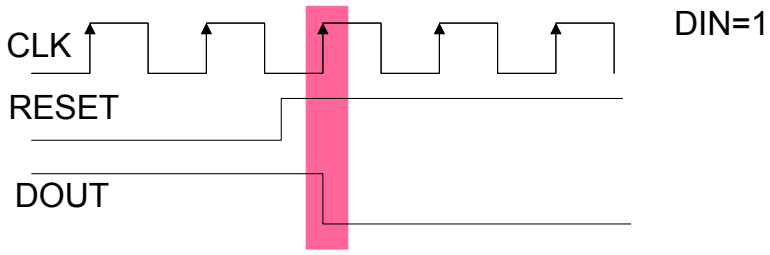
Obr. 14. KO D s asynchronním resetem



2.3.6.3 KO D se synchronním resetem

```
process (CLK)
begin
if CLK'event and CLK='1' then  --CLK rising edge
        if RESET='1' then   --synchronní RESET aktivní při log. 1
                DOUT <= '0';
        else
                DOUT <= DIN;
        end if;
end if;
end process;
```

Obr. 15. KO D se synchronním resetem



2.3.6.4 Klopný obvod JK

```
entity JKFF is
   port (
      CLK, RSTn, J, K : in  bit;
      Q               : out bit);
end JKFF;

architecture RTL of JKFF is
   signal FF : bit;
begin
   process (CLK, RSTn)
```

```
      variable JK : bit_vector(1 downto 0);
   begin
      if (RSTn = '0') then
         FF <= '0';
      elsif (CLK'event and CLK = '1') then
         JK := J & K;
         case JK is
            when "01" => FF <= '0';
            when "10" => FF <= '1';
            when "11" => FF <= not FF;
            when "00" => FF <= FF;
         end case;
      end if;
   end process;
   Q <= FF;
end RTL;
```

2.3.7 Registr s asynchronním nulováním

```
-- 4-bitový registr s asynchronním nulováním
-- signály v části entity
--     CLK: in STD_LOGIC;
--     ASYNC: in STD_LOGIC;
--     LOAD: in STD_LOGIC; aktivací dojde k zápisu s náběžnou hranou clk
--     DIN: in STD_LOGIC_VECTOR(3 downto 0);
--     DOUT: out STD_LOGIC_VECTOR(3 downto 0);

process (CLK, ASYNC)
begin
       if ASYNC='1' then
              DOUT <= "0000";
       elsif CLK='1' and CLK'event then
              if LOAD='1' then
                     DOUT <= DIN;
              end if;
       end if;
end process;
```

2.3.8 Posuvný registr

```
-- 4-bitový posuvný registr (vpravo) se sériovým vstupem a výstupem
-- signály v části entity
--     CLK: in STD_LOGIC;
--     DIN: in STD_LOGIC;
--     DOUT: out STD_LOGIC;

process (CLK)
       variable REG: STD_LOGIC_VECTOR(3 downto 0);
begin
       if CLK'event and CLK='1' then
              REG := DIN & REG(3 downto 1);
       end if;
       DOUT <= REG(0);
end process;
```

2.3.9 Čítače

2.3.9.1 Tříbitový čítač s asynchronním resetem

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;

entity cnt07 is
    port (
       CLK: in STD_LOGIC;
       RESET: in STD_LOGIC;
       COUNT: out STD_LOGIC_VECTOR (2 downto 0)
    );
end cnt07;


architecture beh of cnt07 is
begin

process (CLK, RESET)
variable COUNT_INT: STD_LOGIC_VECTOR(2 downto 0);
begin
       if RESET = '0' then
              COUNT_INT := "000";
       elsif CLK='1' and CLK'event then
              if COUNT_INT = "111" then
                     COUNT_INT := "000";
              else
                     COUNT_INT := COUNT_INT + 1;
              end if;
       end if;
       COUNT <= COUNT_INT;
end process;
end beh;
```

2.3.9.2 Složitější čítače

Obousměrný (DIR určuje směr čítání) synchronní čítač s předvolbou počátečního stavu (LOAD) a s povolením činnosti (CE).

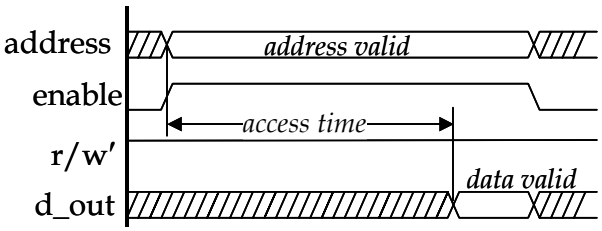
```
-- signály v části entity
--     LOAD: in STD_LOGIC; DIR: in STD_LOGIC;
--       DIN: in STD_LOGIC_VECTOR (3 downto 0);
--     COUNT: out STD_LOGIC_VECTOR (3 downto 0)

process (CLK, RESET)
       variable COUNT_INT: STD_LOGIC_VECTOR(3 downto 0);
begin
       if RESET = '1' then
              COUNT_INT := (others => '0');
       elsif CLK='1' and CLK'event then
              if LOAD = '1' then
                     COUNT_INT := DIN;
              else
                     if CE = '1' then
                            if DIR = '1' then -- čítej nahoru
                                   if COUNT_INT = "1111" then
                                          COUNT_INT := (others => '0');
                                   else
                                          COUNT_INT := COUNT_INT + 1;
                                   end if;
                            else - čítej dolů
                                   if COUNT_INT = "0000" then
                                          COUNT_INT := (others => '1');
                                   else
                                          COUNT_INT := COUNT_INT - 1;
                                   end if;
                            end if;
                     end if;
              end if;
       end if;
       COUNT <= COUNT_INT;
end process;
```

x+y

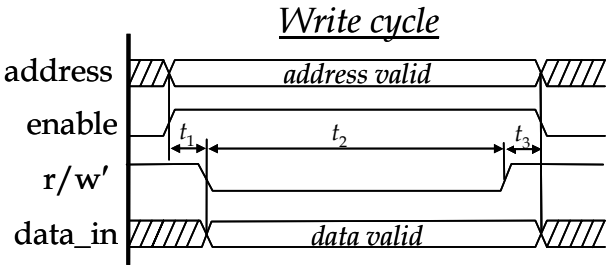
2.3.10 RAM

Paměť RAM se obvykle implementuje pomocí pole vektorů. V uvedeném příkladu obsahuje paměť 64 šestnáctibitových slov. Význam signálů je patrný z časových diagramů na následujících obrázcích. Zápis do paměti je synchronizován hodinovým signálem *clk* a dochází k němu, pokud je *en* = 1, *r_w* = 0 a *reset* = 0. Pokud je signál *reset* aktivován (reset = 1) je možné inicializovat obsah paměti (což se hodí při simulacích). Pokud je *en* = 1 a *r_w* = 1, dochází ke čtení z paměti. Pokud není signál en aktivní je paměť ve stavu vysoké impedance. Všimněme si, že dBus je port obousměrný.

```
entity ram is
    port (
        clk, reset, en, r_w: in STD_LOGIC;
        aBus: in STD_LOGIC_VECTOR(15 downto 0);
        dBus: inout STD_LOGIC_VECTOR(15 downto 0)
    );
end ram;

architecture ramArch of ram is
type ram_typ is array(0 to 63) of STD_LOGIC_VECTOR(15 downto 0);
signal ram: ram_typ;
begin
      process(clk) begin
            if clk'event and clk = '1' then
                  if reset = '1' then
                        -- inicializace paměti
                        ram(0)  <= x"1a0f";
                        ram(1)  <= x"2010";
                        -- atd.
                  elsif en = '1' and r_w = '0' then -- zápis
                        ram(conv_integer(unsigned(aBus))) <= dBus;
                  end if;
            end if;
      end process;
      dBus <= ram(conv_integer(unsigned(aBus)))  -- čtení
                  when reset = '0' and en = '1' and r_w = '1' else
                  "ZZZZZZZZZZZZZZZZ";
end ramArch;
```



Obr. 16. Čtecí cyklus RAM



Obr. 17. Zapisovací cyklus RAM

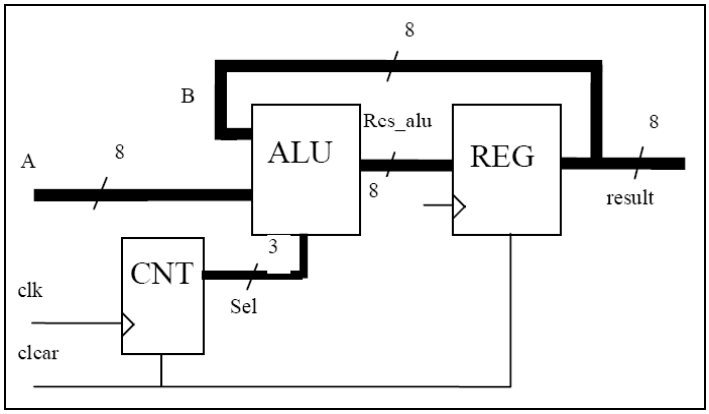
2.3.11 Komunikace komponent

Na Obr. 18 je uveden příklad složitějšího číslicového obvodu, který sestává z ALU, registru a čítače. Obvody tohoto typu je nejjednodušší popsat behaviorálně pomocí komunikujících procesů. Uvedeme si postup, jak popsat takový obvod ve VHDL.

Specifikace obvodu je následující: Cyklický synchronní čítač cnt generuje řídicí signály Sel pro ALU. Signál clear asynchronně nastavuje čítač do stavu 000. Čítač prochází stavy 0-7. Ke změně stavu čítače dochází s náběžnou hranou hodinového signálu. ALU realizuje jednu z osmi funkcí dle hodnoty řídicího signálu Sel. Pracuje s čísly v doplňkovém kódu na 8 bitech (7 bitů, 1 bit na znaménko). Registr reg zaznamenává s náběžnou hranou hodinového signálu výsledek výpočtu ALU. Na začátku výpočtu je jeho obsah vynulován.

Realizace: Každé komponentě bude odpovídat jeden proces. Procesy budou komunikovat pomocí signálů deklarovaných v části atrchitecture. Komponenty jsou synchronizovány hodinovým signálem *clk*.

Obr. 18. ALU s registrem



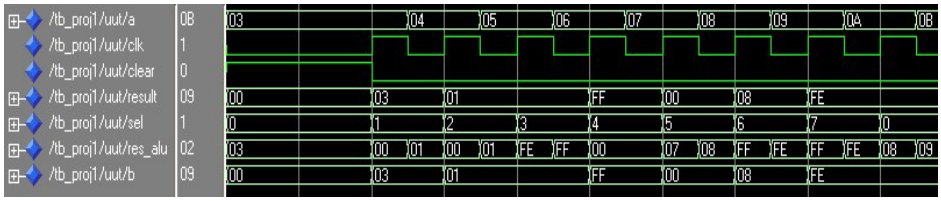

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity proj1 is

port(
      A : in std_logic_vector(7 downto 0);  -- vstupni data
      clk : in std_logic;          -- hodinovy signal
      clear : in std_logic;                     -- nulovani
      result: out std_logic_vector(7 downto 0) -- vysledek
);

end proj1;

architecture behv of proj1 is

signal Sel: std_logic_vector(2 downto 0);        -- rizeni ALU
signal Res_alu: std_logic_vector(7 downto 0);  -- vystup ALU
signal B: std_logic_vector(7 downto 0);          -- vystup registru

begin
-- alu
alu: process(A, B, Sel)
      begin
      case Sel is
```

```
                when "000" =>
                       Res_alu <= A + B;
                when "001" =>
                       Res_alu <= A + (not B) + 1;
                when "010" =>
                       Res_alu <= A and B;
                when "011" =>
                       Res_alu <= A nand B;
                when "100" =>
                       Res_alu <= A nor B;
                when "101" =>
                       Res_alu <= A xor B;
                when "110" =>
                       Res_alu <= A xnor B;
                when "111" =>
                       Res_alu <= A or B;
                when others =>
                       Res_alu <= (Res_alu'range => 'X');
          end case;
      end process;

-- registr
reg: process(Res_alu, clk, clear)
       begin
       if clear = '1' then
          B <= (B'range => '0');
       elsif (clk = '1' and clk'event) then
          B <= Res_alu;
       end if;
end process;

-- citac
cnt: process(clk, clear)
begin
       if clear = '1' then
           Sel <= (Sel'range => '0');
       elsif (clk = '1' and clk'event) then
              Sel <= Sel + 1;
       end if;
       end process;
       result <= B;
end behv;
```

Následující testbench umožňuje ověřit funkci navrženého obvodu:

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

-- komponenta nemá vstupy ani výstupy
entity tb_proj1 is
end tb_proj1;

architecture beh of tb_proj1 is

-- pomocí těchto signálů se připojíme k testované jednotce
signal A:     std_logic_vector(7 downto 0);
signal clk : std_logic;
signal clear : std_logic;
signal  result:     std_logic_vector(7 downto 0);

-- deklarace testované jednotky
component proj1
port ( A:     in std_logic_vector(7 downto 0);
             clk : in std_logic;
             clear : in std_logic;
```

```
            result:  out std_logic_vector(7 downto 0)
);
end component;


begin

-- "instance" testované jednotky signalu
UUT : proj1
port map (
       A => A,
       clk => clk,
       clear => clear,
       result => result
);

process
variable i : std_logic;
begin
       -- nastav čítač a registr do počátečního stavu
       clear <= '1';
       clk <= '0';
       A <= "00000011";

       wait for 20 ns;

       -- vyzkoušej 10 taktů clk
       clear <= '0';
       for i in 1 to 10 loop
              clk <= '1';
              wait for 5 ns;
              clk <= '0';
              A <= A + 1;
              wait for 5 ns;
       end loop;
end process;
end beh;
```



Uvedenou úlohu řešte tak, že si připravíte implementace jednotlivých komponent v samostatných souborech a potom celý obvod popíšete strukturně – propojíte tyto komponenty v jiném souboru.

2.3.12 Popis automatů

Stavový automat obvykle definujeme pomocí 2-3 procesů. Definice musí obsahovat:

- stavovou proměnnou (současný a následující stav),
- jediný hodinový signál,
- specifikaci změny stavu,
- specifikace výstupů a
- reset (není nutné).

Stavové proměnné musí být signály nebo proměnné, nesmí to být signál rozhraní komponenty. Operace nad stavovou proměnnou je limitována na "=" a "/=". Kódování vnitřních stavů vznikne při syntéze. Nejdříve deklarujme nový

typ my_state. Vytvoříme dvě proměnné tohoto typu: jedna bude nést hodnotu současného stavu a druhá hodnotu příštího stavu.

```
TYPE my_state IS (frst_state, s_something, s_something_other);
signal cur_state, next_state : my_state;
```

První proces bude realizovat přepnutí mezi stavy, ke kterému dojde s příchodem náběžné hrany hodinového signálu. Dále při resetu nastaví počáteční stav.

```
process (clk, reset)
begin
   if reset='1' then
      cur_state <= frst_state;
   elsif (clk'event and clk='1') then
      cur_state <= next_state;
   end if;
end process;
```

Druhý proces bude na základě hodnot vstupních signálů a hodnoty současného stavu počítat hodnotu následujícího stavu.

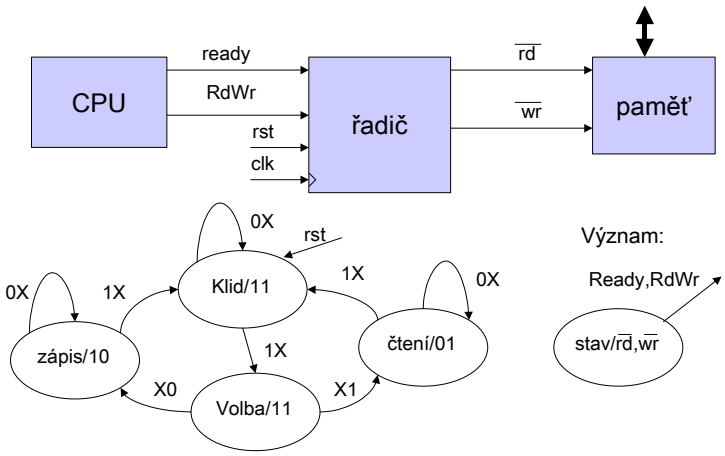
```
ns_logic : process (cur_state, inputs)
begin
   next_state <= frst_state;

   case cur_state is
      when frst_state   =>
         next_state <= s_something;

      when s_something  =>
         next_state <= s_something_other;

      when others       =>
         null;
   end case;
end process ns_logic;
```

Třetí proces bude na základě hodnot vstupních signálů a hodnoty současného stavu definovat hodnoty výstupních signálů. Pokud budeme realizovat Moorův automat, potom budou tyto výstupní hodnoty záviset pouze na stavech.

```
o_logic : process (cur_state, inputs)
begin
   out_sig  <= '1';

   case cur_state is
      when frst_state   =>
         out_sig  <= '0';

      when s_something  =>
         out_sig  <= '1';

      when others       =>
         null;
   end case;
end process o_logic;
```

Příklad: Ve VHDL popište a simulujte řadič paměti, který pracuje podle zadaného Moorova automatu. Napište testbench, kterým ověříte jeho funkci.

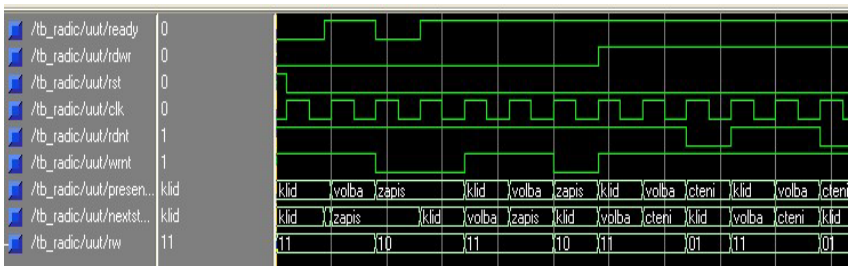
Obr. 19. Řadič paměti




```
entity RdWrMoo is port
   (ready, RdWr, Rst,      Clk    : in std_logic;
    Rdnt, Wrnt  : out std_logic );
end RdWrMoo;

architecture beh of RdWrMoo      is

type StateType is (Klid, Volba, Zapis, Cteni);
signal PresentState, NextState : StateType;
signal Rw: std_logic_vector(1 downto 0); -- pomocný signál (výstupy)

begin
Rdnt <= Rw(1);      Wrnt <= Rw(0);

-- stavový regist
Clock: process (Clk)
begin
       if Rst = '1' then
              NextState <= Klid;
       elseif if (clk'event and clk = '1') then
              PresentState <= NextState;
       end if;
end process Clock;

-- výpočet nového stavu a hodnot výstupů
Comb: process (PresentState, Ready, RdWr, Rst)
 begin
  case PresentState is
       when Klid => Rw <="11";
       if Ready = '1' then NextState <= Volba;
       else                    NextState <= Klid;
       end if;

       when Volba => Rw <= "11";
       if RdWr = '1' then  NextState <= Cteni;
       else                     NextState <= Zapis;
       end if;

       when Zapis => Rw <= "10";
       if Ready = '1' then NextState <= Klid;
       else                    NextState <= Zapis;
       end if;

       when Cteni => Rw <="01";
       if Ready = '1' then NextState <= Klid;
       else                NextState <= Cteni;
       end if;
   end case;
  end process Comb;
end beh;
```

Test bench pro řadič paměti:

```
entity tb_radic is
end tb_radic;

architecture beh of tb_radic is
signal ready, RdWr, Rst, Clk : std_logic;
signal Rdnt, Wrnt : std_logic;

component RdWrMoo
port ( ready,RdWr,Rst,Clk   : in std_logic;
       Rdnt,Wrnt  : out std_logic
);end component;

begin
UUT : RdWrMoo -- "instance" testované jednotky
port map (ready => ready, RdWr => RdWr,
Rst => Rst,Clk => Clk,Rdnt => rdnt,Wrnt => Wrnt);

process
begin
  Rst <= '1';  Ready <= '0';      RdWr <= '0'; clk <= '0';
  wait for 5 ns;   Rst <= '0';
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 7 ns;
  Ready <= '1';     wait for 3 ns;

  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
  Ready <= '0';
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
  Ready <= '1';
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
  RdWr <= '1';
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
  clk <= '1';  wait for 10 ns;  clk <= '0'; wait for 10 ns;
end process; end beh;
```



Obr. 20. Ověření činnosti řadiče paměti

2.4 Shrnutí

V této kapitole byla vysvětlena filozofie jazyka VHDL a uvedeny základní konstrukce jazyka VHDL. Uvedené příklady ilustrují popis základních komponent číslicových obvodů. Podrobnější informace o VHDL je možné získat v literatuře, viz seznam.

3 ModelSim

ModelSim je program pro simulaci číslicových obvodů, které mohou být popsány např. ve VHDL. Tento text poskytuje základní informace o tom, jak si

ModelSim nainstalovat doma a jak ho používat. Na školních počítačích je ModelSim rovněž nainstalován.

Varování: Nové verze programu ModelSim jsou vydávány poměrně často. Následující popis nemusí být vždy aktuální!

Podrobnější informace o práci s ModelSimem nabízí stránky FitKitu:
http://merlin.fit.vutbr.cz/FITkit/?pg=navody

3.1 Jak provozovat ModelSim doma

Počítač, na kterém chcete provozovat ModelSim, musí být připojitelný k Internetu (např. přes modem). Musíte totiž z něho zaregistrovat nainstalovaný program u společnosti Xilinx – během registrace se podle některých čísel ukrytých ve vašem počítači vytváří licenční soubor, který je unikátní pro každý počítač a bez kterého software nefunguje. Nejdříve se musíte zaregistrovat (vytvořit si účet, pokud ho ještě nemáte) u společnosti Xilinx na:
http://www.xilinx.com/ise/logic_design_prod/webpack.htm
To můžete udělat z jakéhokoliv počítače (třeba ve škole). Klikněte na Download Free. Dále klikněte na Create an Account a vyplňte a potvrďte, co je požadováno. Emailem vám přijde potvrzení o vytvoření účtu. Dále je potřeba stáhnout ModelSim (nebo si ho zkopírovat od někoho, kdo už ho stáhl). Ze stránce http://www.xilinx.com/webpack/index.htm stáhněte ModelSim Xilinx Edition-III (MXE-III) cca 103MB.

Spusťte instalační soubor/program a nainstaluje ModelSim. Během instalace vyberte verzi, která je označena jako Starter, popř. FREE. V dalším okně vyberte Full VHDL.

Po instalaci je nutné ModelSim zaregistrovat. Prakticky až v tomto okamžiku potřebujete přístup z vašeho počítače (na kterém jste právě nainstalovali ModelSim) na Internet. Buď si nechte poslat licenční soubor v posledním kroku instalace nebo po instalaci v menu Start … ModelSim XE III vyberte Submit Licence Request. Vyberte Continue a nechte si poslat licenční soubor. Licenční soubor vám vzápětí přijde emailem. V menu Start … ModelSim XE III a vyberte Licence Wizard. Pomocí Browse vyberte váš licenční soubor a je to.

3.2 Základy práce v ModelSimu

ModelSim je program, který umožní vytvořit, editovat, kompilovat a simulovat obvody popsané v jazycích VHDL nebo Verilog. Nás bude zajímat jen VHDL. Zdrojový kód ve VHDL můžete editovat v libovolném textovém editoru podobně jako zdrojový kód v Pascalu nebo C++.

Vytvořte si někde adresář, který se bude jmenovat např. AA. Do adresáře AA si zkopírujte soubor fa.vhd (sčítačka) a tb_fa.vhd (test bench). Jedná se o úplnou sčítačku popsanou ve VHDL, na které si ukážeme si základní funkce ModelSimu. Spusťte ModelSim.

V menu File->New vyberte Project. Zvolte Project name např. fa a v Project Location nastavte adresář AA. OK. Objeví se okno Add File To Project, ve kterém klikněte na Add Existing File a přidejte fa.vhd a tb_fa.vhd. Zavřete okno Add File To Project.

Okno ModelSimu je rozděleno na dvě podokna. Levé obsahuje záložky Projekt a Library. Druhé okno obsahuje příkazovou řádku, ze které ke možné zadávat příkazy.

Pokud 2x klikneme na fa.vhd, otevře se editor a soubor fa.vhd lze editovat. Pokud klikneme pravým tlačítkem myši na fa.vhd, můžeme soubor zkompilovat. Alternativně pomocí menu Compile. V případě nějaké chyby je třeba zdrojový kód opravit. Stejně tak zkompilujeme test bech.

Po úspěšném zkompilování souboru v projektu se přepneme do záložky Library. Ve složce work dvakrát klikneme na tb_fa – tím inicializujeme simulátor.

Otevře se záložka Simulate, ve které vybereme UUT – jednotku, jejíž signály chceme sledovat. Nyní nemusíme nastavit vstupní hodnoty, protože je generuje testbench. Pomocí lokálního menu a Add to Wave->Signals in Region se vytvoří se okno, ve kterém budeme sledovat průběh simulace.

Simulaci spustíme pomocí příkazu run 100 ns. Pomocí lupy upravíme časové měřítko tak, abychom viděli všechny kombinace na adresovém vstupu sčítačky. Pokud chceme simulaci zopakovat, vybereme Simulate->Run->Restart a potom pokračujeme jako předtím: run 100 ns.

Celou simulaci ukončíme pomocí menu Simulate->End Simulation. Všimněte si, že zmizela záložka sim.

Podrobný popis práce s programem ModelSim je k dispozici jako .pdf soubor v helpu programu.

3.3 Shrnutí

V této kapitole byl vysvětlen způsob práce s programem ModelSim. Podrobnější informace o programu je možné získat z nápovědy k tomuto programu.

4 Použitá literatura

- Douša, J.: Jazyk VHDL. Skriptum ČVUT, 2003
- Kolouch, J.: Programovatelné logické obvody a modelování číslicových systémů v jazycích ABEL a VHDL. Skriptum VUT v Brně, 2000.
- Chang, K. C.: Digital Design and Modeling with VHDL and Synthesis. IEEE Computer Society, Los Alamitos 1997
- Chang, K. C.: Digital Systems Design with VHDL and Synthesis: An Integrated Approach. IEEE Computer Society, Los Alamitos and John Wiley 1999
- Geißler, R., Bulach, S.: VDHL manual (University of Ulm) http://mikro.e-technik.uni-ulm.de/vhdl/anl-engl.vhd/html/vhdl-all-e.html
- Klenke, R.: Basic VHDL RASSP Education & Facilitation, Module 10, Version 3.00, 1999 http://www.people.vcu.edu/~rhklenke/tutorials/vhdl/modules/m10_23/index.htm
- The Hamburg VHDL Archive. http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html
- The Designer's Guide to VHDL. http://www.ashenden.com.au/designers-guide/DG.html

Obsah