Teoretický úvod – laboratorní úloha číslo 3

Podmínkové konstrukce v jazyce VHDL, konverze typů, realizace sčítačky

1 Behaviorální popis v jazyce VHDL

V předchozí laboratorní úloze č. 2 jsme převodník kódu z BCD do kódu 7segmentového displeje realizovali pomocí tzv. dataflow (RTL) popisu a stručně jsme zmínili, že kromě něj je v jazyce VHDL možné použít i tzv. behaviorální a strukturální popis. Úkolem v této laboratorní úloze je využít právě behaviorální (`Behavioral`) typ pro realizaci sčítačky. Jak z jeho názvu vyplývá, tento způsob je založen na popisu chování navrhovaného obvodu v jazyce VHDL. Navrhovaný obvod si tak lze představit jako tzv. black box (černou skříňku), kdy známe pouze jeho vstupy a výstupy (porty entity) a víme, jaké hodnoty se mají objevit na jeho výstupech, pokud použijeme k buzení jeho vstupů konkrétní hodnotu. Neznáme tedy ani vnitřní stavbu (zapojení) obvodu, seznam použitých hradel (součástek), nedokážeme ani vyjádřit funkci obvodu pomocí Booleových rovnic (jako v případě dataflow (RTL)). Pro behaviorální popis jsou v jazyce VHDL typické právě podmínkové konstrukce, kdy vyjdeme např. z pravdivostní tabulky obvodu nebo jeho popisu chování a vyjmenováním a otestováním jednotlivých vstupních kombinací a stavů specifikujeme výstupy obvodu. Tento způsob popisu obvodů v jazyce VHDL má své výhody i nevýhody, v rámci jedné entity lze samozřejmě využít kombinaci různých způsobů popisu, podrobněji viz přednášky předmětu.

2 Paralelní a sekvenční prostředí, procesy v jazyce VHDL

V jazyce VHDL můžeme využít dvě odlišná prostředí (`domain`) pro zápis kódu v architektuře entity, a to paralelní (`concurrent`) a sekvenční (`sequential`). Vzhledem k tomu, že VHDL jako HDL jazyk pro popis hardware byl primárně navržen pro popis digitálních obvodů a logických struktur, které se z principu chovají paralelně a skrz každý blok, hradlo či vodič nezávisle na ostatních se šíří různé signály ve stejný okamžik, **je jazyku VHDL z podstaty vlastní paralelní prostředí** (`concurrent domain`). Paralelní prostředí v jazyce VHDL si také můžeme jednoduše představit jako obecný kombinační obvod složený ze základních logických hradel, kdy výstup obvodu je pevně zapojen pomocí hradel a vodičů a jakákoliv změna na vstupu obvodu se okamžitě promítne do jeho výstupu. Pro paralelní prostředí tedy platí, že veškeré příkazy jsou vykonávány souběžně a okamžitě a nezávisle na pořadí v jakém jsou zapsány. Mezi typické paralelní příkazy a konstrukce v jazyce VHDL patří zejména přiřazování hodnot do signálů a portů, podmíněné přiřazování (`select`/`when`, `when`/`else`), využití základních logických operací (pomocí dataflow – RTL popisu), propojování komponent pomocí tzv. `port map`, souběh procesů apod.

Kromě paralelního existuje v jazyce VHDL pro popis obvodu v rámci architektury i **sekvenční prostředí** (`sequential domain`). Jak z názvu vyplývá, jde v tomto případě o postupné zpracování příkazů jdoucích po sobě v pořadí, v jakém jsou zapsány. Aby bylo možné zapisovat sekvenční příkazy do principiálně paralelního prostředí jazyka VHDL, je nutné je

zapsat vždy do tzv. procesů (`process`). Architektura entity v jazyce VHDL může pak obsahovat libovolné množství procesů, které navzájem vedle sebe běží paralelně, ale příkazy uvnitř samotných procesů se vykonávají sekvenčně. Mezi typické sekvenční příkazy jazyka VHDL patří podmínkové konstrukce (`if/else`, `case`), podmíněné čekání `wait`, smyčky `loop`, `for`, `while` apod. Sekvenční prostředí se v jazyce VHDL typicky využívá pro realizaci synchronních obvodů synchronizovaných pomocí periodického (hodinového) signálu, sekvenčních logických obvodů, stavových automatů a dalších.

Proces je základním blokem pro zápis sekvenčního kódu ve VHDL, uveďme jeho syntaxi:

```
[Navesti:] process [(citlivostni seznam)]
    -- deklarace promennych a konstant
begin
    -- telo procesu
end process [Navesti];
```

Návěstí (pojmenování) procesu je nepovinné a lze ho využít pro potřeby skoku či odkazu v rámci VHDL kódu. Citlivostní seznam procesu představuje seznam vstupů procesu, signálů či portů, a daný proces je spuštěn vždy, když je detekována změna jakéhokoliv vstupu v tomto seznamu. Není-li citlivostní seznam definován, je proces aktivován ihned a je neustále opakovaně vykonáván, případně syntezátor jazyka VHDL vytvoří citlivostní seznam procesu automaticky při jeho syntéze. Výjimkou je použití příkazu `wait` v těle procesu, který rovněž určuje čekání na změnu stavu definovaného vstupu, pokud proces tento příkaz obsahuje, nesmí mít zároveň citlivostní seznam. V procesu můžeme deklarovat lokální proměnné a konstanty, které jsou platné a dostupné jen v rámci daného procesu. Tyto proměnné a konstanty mohou například sloužit jako paměť dočasného výpočtu či pro uložení stavu některého vstupu/výstupu, či konstantní parametr apod. Mezi klíčová slova `begin` a `end process` pak zapisujeme vlastní sekvenční příkazy.

3 Podmínkové konstrukce v paralelním prostředí

V paralelním prostředí lze pro podmíněné přiřazování využít dvojici podmínkových konstrukcí, `select/when` a `when/else`. V syntaxi `select/when` nejprve deklarujeme název signálu/vstupu, který testujeme, a dále dle výsledku jeho stavu (hodnoty) přiřazujeme do výstupu/signálu požadovanou hodnotu:

```
with {testovany vstup} select
{vystup} <= {hodnota ci prirazeni} when {moznost 1},
            {hodnota ci prirazeni} when {moznost 2},
             ...
            {hodnota ci prirazeni} when others;
```

Jednotlivé možnosti oddělujeme čárkou a důležitá poslední podmínka specifikuje, jaká hodnota se má na výstup uložit, pokud neplatí žádná z předchozích. Implementací podmínky typu `select/when` je multiplexor. Druhou podmínkovou konstrukci v paralelním prostředí jazyka VHDL `when/else` můžeme definovat:

```
{vystup} <= {hodnota ci prirazeni} when {podminka 1} else
            {hodnota ci prirazeni} when {podminka 2} else
             ...
            {hodnota ci prirazeni};
```

Z porovnání obou podmínkových konstrukcí je patrné, že typ `when/else` je obecnější, neboť v každém kroku můžeme klidně testovat jiný (libovolný) vstup či podmínku, zatímco v případě `select/when` je hned na začátku určeno, jaký jeden vstup testujeme skrze všechny možnosti. I v případě druhé podmínkové konstrukce typu `when/else` je na posledním řádku zvykem uvést, jaká má být vložena hodnota do výstupu, pokud ani jedna z předchozích podmínek neplatí. Implementací podmínky typu `when/else` je prioritní kodér.

4 Podmínkové konstrukce v sekvenčním prostředí

Analogií podmínek v paralelním prostředí jsou podmínky typu `case` a `if/else` v sekvenčním prostředí, ty musí být však vždy zapsány v těle procesu. Podmínka typu `case` odpovídá předchozí `select/when` a její implementací je multiplexor. Opět tedy na počátku definujeme, který vstup budeme testovat, a postupně vyjmenováváme jednotlivé možnosti, na jejichž základě pak přiřazujeme hodnotu či přiřazení do výstupu:

```
case {testovany vstup} is
when {moznost 1} => {sekvence prikazu 1};
when {moznost 2} => {sekvence prikazu 2};
...
when others => {sekvence prikazu 3};
end case;
```

Oproti typu `select/when` v případě `case` oddělujeme jednotlivé možnosti pomocí středníku za přiřazením a celou podmínkovou konstrukci musíme zakončit klíčovým slovem `end case`. V případě podmínky `case` můžeme také pomocí symbolu `|` sloučit více možností dohromady. Stejně jako v případě typu `select/when`, obvyklý poslední řádek s klíčovým slovem `when others` specifikuje hodnotu či přiřazení do výstupu, pokud ani jedna z předchozích možností není splněna. Druhou častou podmínkovou konstrukcí v sekvenčním prostředí jazyka VHDL je `if/else`, jež je analogií k `when/else`:

```
if {podminka 1} then
    {sekvence prikazu 1};
elsif {podminka 2} then
     {sekvence prikazu 2};
...
else
     {sekvence prikazu 3};
end if;
```

Opět, jako v případě typu `when/else`, lze v každém kroku otestovat libovolný vstup či podmínku, jedná se proto o obecnější typ než v případě `case`. Dobrým zvykem bývá opět

definovat posledním klíčovým slovem `else` možnost, pokud ani jedna z předchozích podmínek neplatí. Podmínku `if` je pak nutné zakončit pomocí klíčového výrazu `end if`. Výhodou podmínek typu `if` je, že je lze vnořovat do sebe navzájem, v takovém případě však každá podmínka `if` musí být řádně ukončena svým výrazem `end if`. Kromě pokračování původní podmínky `if` pomocí klíčového slova `elsif` existuje v jazyce VHDL i výraz `else if`. Ten však vznikne spojením samostatných výrazů `else` a `if` a jedná se tedy o vnoření zcela nové podmínky `if` do původní varianty `else` nadřazené podmínky `if` (je tedy nutné vnořenou podmínku zakončit vlastním `end if`) a nikoliv o pokračování původní nadřazené podmínky `if`. Podmínka typu `if` je syntezátorem implementována jako prioritní kodér.

Jak již bylo zmíněno, podmínky typu `if/else` a `when/else` umožňují testovat v každé podmínce libovolné jiné vstupy či stavy, navíc lze testovat složené podmínky, tedy např. současný stav dvou či víc vstupů. V takovém případě se složené podmínky pro lepší přehlednost kódu zapisují často do kulatých závorek a pro testování současné platnosti se mezi jednotlivé části složené podmínky (vstupy) vkládají odpovídající logické operace (součin či součet), pokud požadujeme současnou platnost všech výrazů (částí), či platnost alespoň jedné. Obě podmínky lze jako prioritní kodéry rovněž zapsat tak, že může zároveň platit i více možností. Zatímco podmínky `select/when` a `case` jsou implementovány jako multiplexory, a tedy nesmí nastat současně platnost více možností.

5 Logický vektor std_logic_vector v jazyce VHDL

Tuto kapitolu věnujeme problematice logických vektorů `std_logic_vector`. Již v laboratorní úloze č. 2 jsme definovali datový typ `std_logic`, který v jazyce VHDL představuje standardní sadu logických hodnot. Pokud sdružíme několik portů, signálů či proměnných typu `std_logic`, získáme obdobu sběrnice, ať již na vstupu obvodu, výstupu, nebo interní sběrnici uvnitř obvodu (entity). Této sběrnici odpovídá právě tzv. vektor[1] v jazyce VHDL, tedy sdružení *n* 1bitových logických portů, signálů či proměnných do výsledné n-bitové sběrnice neboli n-bitového logického vektoru. Díky tomu lze s celým vektorem provádět naráz různé operace, samozřejmě lze však stále přistupovat ke každému takto sdruženému portu/signálu/proměnné i nadále samostatně. Při deklaraci vektoru musíme vždy uvést jeho velikost (kolik 1bitových portů, signálů, proměnných jsme sdružili) a také jejich číslování, které pak slouží pro identifikaci jednotlivých sdružených vstupů. V jazyce VHDL je obvyklé číslovat od 0 a obvykle v sestupném směru (LSB pozici odpovídá 0, MSB pozici pak nejvyšší číslo vektoru), není však problém očíslovat vektor i vzestupně a i od libovolného nezáporného čísla. V laboratorní úloze č. 2 jsme uvedli, že 1bitové logické hodnoty se v jazyce VHDL zapisují pomocí apostrofů, např. logická 1 jako `'1'`, pokud sdružíme *n* 1bitových logických hodnot do logického vektoru, zapisují se pak pomocí uvozovek, např. dvojice logických 1 jako `"11"`.

```
port (a : in std_logic_vector(1 downto 0));
a <= "01";
```

[1] Kromě `std_logic_vector`, tedy logických vektorů, v jazyce VHDL můžeme vytvořit i vektory (sběrnice) z některých dalších datových typů, např. `bit_vector`. Také datové typy `signed`, `unsigned` lze chápat více méně jako vektory a rovněž datový typ `string` je v podstatě vektorem.

Vstup `a` jsme definovali jakou 2bitovou sběrnici očíslovanou sestupně, což vyplývá z použití klíčového slova `downto` pro sestupné číslování od 1 do 0. Pokud bychom chtěli přistupovat samostatně k jednotlivým bitům vektoru `a`, využijeme k tomu kulaté závorky s číslem požadované pozice v rámci vektoru. Na vyšší pozici vektoru `a` se tedy nachází bit `a(1)` a na nižší pak `a(0)`. Pokud bychom chtěli vložit samostatně na tyto jednotlivé bity stejné hodnoty jako výše, můžeme to udělat následovně, výsledek bude samozřejmě stejný:

```
a(1) <= '0';
a(0) <= '1';
```

Pokud bychom chtěli naopak očíslovat vektor vzestupným směrem, použijeme k tomu klíčové slovo `to` a pořadí čísel v závorce při deklaraci musí být samozřejmě vzestupné, např.:

```
port (b : in std_logic_vector(1 to 3));
```

Vytvoří vektor `b` o velikosti 3 bitů očíslovaný 1, 2 a 3 směrem od LSB k MSB. Při testování hodnoty vektoru pomocí podmínek, které jsme si představili v kapitole 3 a 4, můžeme testovat samozřejmě celý vektor, nebo i jen jeho jednotlivé n-bitové části, vytvářet složené podmínky, vnořené podmínky apod. To může být v různých situacích výhodné, vytvořme třeba ukázku, ve které využijeme vektor `a` definovaný na začátku kapitoly a typ `if`:

```
if a="01" then
 ...
end if;
```

```
if (a(1)='0' and a(0)='1') then
  ...
end if;
```

Pomocí složené podmínky typu `if`, kdy testujeme současnou hodnotu bitů `a(1)` a `a(0)` získáme v tomto případě samozřejmě stejný výsledek, jako při použití kódu vlevo.

6 Operátor zřetězení (concatenation) & a atributy

V teoretickém úvodu se ještě zmíníme o tzv. operátoru zřetězení v jazyce VHDL se symbolem `&` (ampersand). Vzhledem k tomu, že často ve VHDL v rámci jedné entity pracujeme jak s jednobitovými datovými typy (např. `std_logic`), tak i vícebitovými datovými vektory (např. `std_logic_vector`), narazíme na potřebu spojit několik jednobitových portů, signálů či proměnných do vícebitového vektoru, nebo i spojit dva vektory do výsledného většího vektoru a pracovat pak s výsledným vektorem jako celkem apod. K tomu slouží v jazyce VHDL tzv. operátor zřetězení (spojení) se symbolem `&` (nejedná se o logický součin, pro nějž je ve VHDL vyhrazen operátor `AND`). Pomocí tohoto operátoru lze zřetězit nejen logické datové typy, ale i např. znakové `char` a `string` (`string` můžeme definovat jako zřetězení jednotlivých symbolů typu `char`), při ladění VHDL kódu lze využít i pro výpis hodnot a informací o stavu proměnných či signálů apod. Při zřetězení se kontroluje celkový počet zřetězovaných bitů na vstupu a velikost výstupního vektoru (která musí samozřejmě souhlasit), záleží rovněž na pořadí, v jakém zřetězujeme dané vstupy do výstupu, jak ukazuje příklad:

```
signal a : std_logic_vector(1 downto 0) :="11";
signal b : std_logic_vector(3 downto 0) :="0010";
signal c : std_logic_vector(5 downto 0);
c<=a&b; -- vysledna hodnota v c bude "110010"
c<=b&a; -- vysledna hodnota v c bude "001011"
c<=a(1)&b(1)&a(0)&b(3 downto 2)&b(0); -- v c bude "111000"
```

V krátkosti se ještě zmíníme i o tzv. atributech v jazyce VHDL. Kromě vlastní hodnoty datového objektu (portu, signálu, proměnné atd.) můžeme také sledovat různé dodatečné informace a vlastnosti tohoto objektu, k čemuž v jazyce VHDL slouží právě tzv. atributy. Atributů je v jazyce VHDL definována celá řada, většina je omezena na použití jen s vybranými datovými typy (např. délka vektoru, pozice ve vektoru apod.), některé jsou univerzálně použitelné (např. atribut pro sledování změny hodnoty apod.), řadu z nich lze využít i přímo pro syntézu obvodů, ale některé jsou určeny jen pro simulace. Obecnou syntaxi atributu v jazyce VHDL spolu s několika ukázkami můžeme uvést např.:

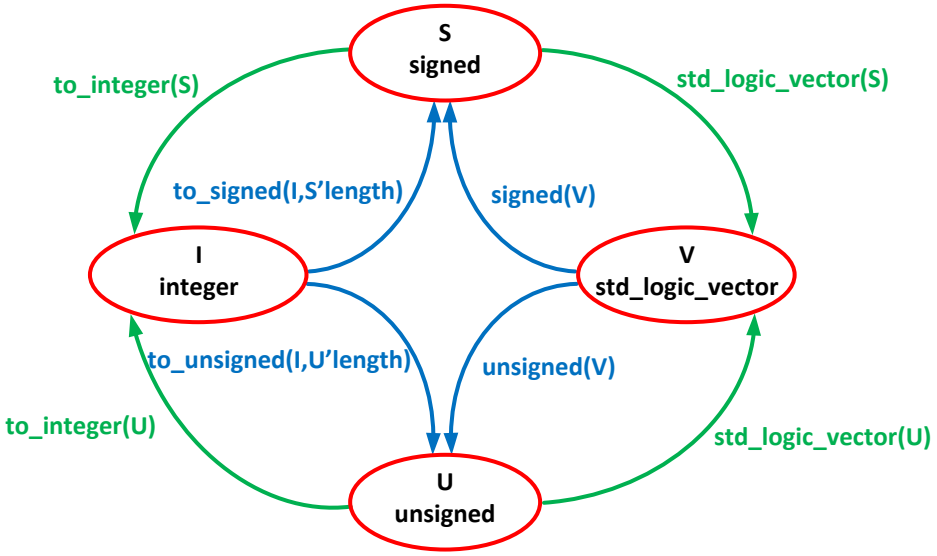
```
datovy_objekt'atribut; -- k zapisu atributu slouzi apostrof '
'event -- casty atribut v jazyce VHDL slouzici k detekci hrany
signalu (okamzik zmeny signalu z log. 1 na 0 a naopak)
'length -- atribut pro zjisteni delky (poctu pozic) vektoru
'high – nejvyssi hodnota ve vektoru (poli)
-- a cela rada dalsich...[2]
```

7 Konverze datových typů, knihovna numeric_std, datové typy signed/unsigned

Datovým typem v jazyce VHDL rozumíme typ (formát) dat, která daný port, signál, proměnná či konstanta může přenášet, obsahovat, zapisovat apod. Při deklaraci každého nového objektu (portu, signálu, proměnné apod.) v jazyce VHDL musíme právě specifikovat její datový typ. Datové typy v jazyce VHDL můžeme rozdělit do 4 základních kategorií: logické datové typy (např. std_logic, bit, boolean a další), číselné datové typy (integer, real, natural, signd/unsigned a jiné), znakové datové typy (např. char, string), vlastní datový typ (výčet). Specifickým datovým typem je `time`, který lze však využít pouze v simulacích a nikoliv pro syntézu obvodů. V předchozí laboratorní úloze č. 2 jsme si představili jeden z nejpoužívanějších logických datový typů `std_logic`, v rámci této úlohy jsme si v předchozí kapitole 5 doplnili i jeho rozšíření na datový typ `std_logic_vector`. Jazyk VHDL je obecně silně typově orientovaný, je tedy nutné dbát striktně pravidel, které operace lze se kterými datovými typy provádět, pomocí tzv. konverzních funkcí lze mezi sebou vybrané datové typy převádět, což umožňuje využít některé dodatečné funkce jazyka. V této laboratorní úloze s výhodou využijeme datový typ unsigned (signed), který si blíže popíšeme.

[2] S využitím některých jiných atributů se setkáme v dalších laboratorních úlohách a několik ukázek si uvedeme také na přednáškách.

Pro realizaci některých obvodů[3] je někdy výhodné chápat hodnotu logického vektoru `std_logic_vector` nikoliv jako sběrnici logických hodnot logická 1 a 0, ale jako binární číslo vyjádřené pomocí 1 a 0 a to buď bez uvažování znaménka (`unsigned`), nebo s uvažováním znaménkového bitu (`signed`). S takto vyjádřenou hodnotou lze například provádět běžné aritmetické operace jako je např. sčítání (+) či odečítání (-) s logickými vektory `std_logic_vector`, porovnávat hodnotu vektoru s číslem typu `integer` apod. Tomuto procesu, kdy lze za určitých podmínek použít konkrétní operátor i pro jinou funkci nebo jiné vstupní datové typy, než pro jaké byl původně definován, se obecně říká tzv. přetěžování operátorů. Jazyk VHDL jako striktně typový jazyk dbá na to, aby dané operace byly prováděny pouze s definovanými datovými typy. Nicméně, byly do něho přidány funkce a knihovny, z nichž postupně byla standardizována knihovna IEEE `numeric_std`, která umožňuje převádět datové typy `integer` a `std_logic_vector` navzájem pomocí datových typů `signed` a `unsigned`. Dále lze pak s těmito datovými typy `signed`/`unsigned` též provádět běžné aritmetické operace jako s běžnými čísly, a jejich zpětný převod na `std_logic_vector`, případně číslo typu `integer` převést na logický vektor (viz přednášky). Obr. č. 1 ukazuje takové převodní funkce pro různé směry převodů.



Obr. č. 1: Převody mezi datovými typy integer, signed, unsigned a std_logic_vector.

Obr. č. 1 uvádí konverzní funkce mezi jednotlivými datovými typy. V případě převodu z typu `integer` na typ `signed` či `unsigned` je potřeba ještě uvést, na kolik bitů je celočíselný typ `integer` převáděn, jak je naznačeno pomocí atributu `'length` (délka vektoru). Pro všechny konverze a použití aritmetických operací s typy `signed` a `unsigned` je potřeba v záhlaví VHDL modulu deklarovat použití knihovny:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;
```

[3] Zejména se jedná o čítače, děličky kmitočtu, obvody vykonávající aritmetické operace apod.

8 Sčítačky, úplná a neúplná 1bitová sčítačka

Sčítačkou rozumíme obecně logický obvod, který provádí sčítání dvojice vstupních hodnot, výstupem je pak vlastní součet a případný přenos (příznak přenosu) do vyššího řádu v případě překročení rozsahu původně sčítaných hodnot. V logických obvodech se zaměřujeme na tzv. binární sčítačky, které provádějí sčítání čísel v binárním tvaru (soustavě)[4]. Základem je tzv. jednobitová sčítačka, která dovoluje provádět součet 1bitových vstupních hodnot, výstupem je opět 1bitová hodnota. 1bitovou sčítačku můžeme dále rozlišit na tzv. úplnou a neúplnou (někdy též poloviční). Neúplná 1bitová sčítačka má na vstupu pouze dvojici 1bitových hodnot (`a`, `b`), výstupem je 1bitový součet (`s`) a 1bitový přenos do vyššího řádu (`cout`). Oproti ní, obsahuje úplná sčítačka navíc i 1bitový vstup pro přenos z nižšího řádu (`cin`). Pro realizaci vícebitové sčítačky můžeme např. využít tzv. paralelní zapojení sčítačky s postupným přenosem, kdy pomocí vstupů a výstupů přenosů jednotlivých řádů za sebe kaskádně zapojíme požadovaný počet 1bitových sčítaček. Více se realizací vícebitové sčítačky budeme zabývat v další laboratorní úloze a obecně sčítačkám na přednáškách předmětu.

Úkolem této laboratorní úlohy je v jazyce VHDL realizovat několika odlišnými způsoby úplnou 1bitovou sčítačku. Tab. č. 1 ukazuje její pravdivostní tabulku. Obr. č. 2 pak představuje její zapojení i vyjádření výstupních funkcí pomocí Booleových rovnic.

Tab. č. 1: Pravdivostní tabulka úplné 1bitové sčítačky.

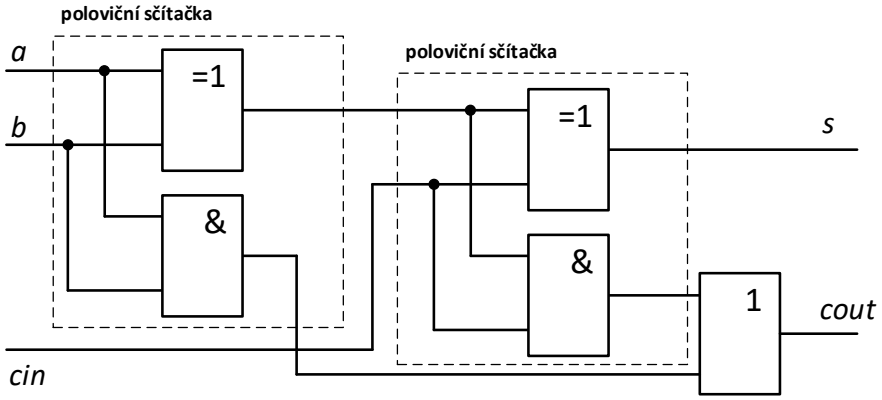
vstupy			**výstupy**	
`cin`	**`b`**	**`a`**	**`cout`**	**`s`**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Výstupní funkce `s` a `cout` úplné 1bitové sčítačky pak můžeme na základě pravdivostní tabulky zapsat:

$$\begin{aligned} s &= \bar{a}\bar{b}cin \vee a\bar{b}\overline{cin} \vee abcin \vee \bar{a}b\overline{cin} = a \oplus b \oplus cin \\ cout &= ab \vee acin \vee bcin = ab \vee a \cdot \left(b \vee \bar{b}\right) \cdot cin \vee \left(a \vee \bar{a}\right) \cdot bcin = ab \vee cin \cdot \left(a \oplus b\right) \end{aligned} \quad . \qquad (1)$$

Na základě vztahu (1) a výše uvedené tabulky pak můžeme zakreslit schéma jejího zapojení.

[4] Binární sčítačka však obecně umožňuje provádět i operaci odečítání, pokud využijeme tzv. dvojkového doplňku a pomocí něho převedeme jeden ze sčítanců, provádíme v podstatě operaci jeho odečítání. Princip a využití dvojkového doplňku jsme si představili a vyzkoušeli v úvodních teoretických cvičení předmětu, které jsme věnovali operacím v číselných soustavách.



Obr. č. 2: Zapojení úplné 1bitové sčítačky.

Na tomto místě si popišme ve stručnosti několik možných způsobů realizace úplné 1bitové sčítačky v jazyce VHDL:

- Tab. č. 1 a pomocí behaviorálního popisu s využitím jedné z podmínkových konstrukcí představených v kapitole 3 a 4 tuto tabulku popsat. Pro tento způsob bychom nejprve provedli zřetězení trojice 1bitových vstupů do jednoho 3bitového logického vektoru, na základě jehož hodnoty bychom následně pomocí některé z podmínek mohli provést přiřazení hodnot obou výstupů.
- Další možností je využít Booleových rovnic pro oba výstupy viz vztah (1) a v rámci dataflow (RTL) popisu je přepsat v jazyce VHDL obdobným způsobem, jako v případě realizace převodníku z kódu BCD do kódu 7segmentového displeje v předchozí laboratorní úloze č. 2.
- Sčítačku bychom také mohli realizovat přímo pomocí operace sčítání, nejprve je však potřeba pomocí vhodných konverzních funkcí provést převod z datového typu `std_logic` na typ `unsigned`, následně provést součet a poté zpět výsledek vhodně převést na typ `std_logic`. K tomu můžeme využít konverze popsané v kapitole 7.
- Obr. č. 2 ukazuje schéma zapojení sčítačky; realizaci je možné provést s využitím strukturálního popisu v jazyce VHDL. Tímto způsobem se však budeme zabývat zejména v následující laboratorní úloze č. 4 při realizaci 2bitové sčítačky.

Na závěr uveďme ještě obecně základ VHDL kódu entity úplné sčítačky, tedy deklaraci knihoven a portů, vlastní popis (architekturu) lze pak doplnit jedním ze způsobů uvedených výše. Jak jsme si uvedli na přednáškách předmětu, jedna entita v jazyce VHDL může obsahovat libovolný počet architektur. Při kompilaci entity, její syntéze a implementaci do FPGA pole je buď potřeba specifikovat, která z architektur se má použít, nebo pokud toto nespecifikujeme, vybere se z entity automaticky poslední zapsaná architektura, jak si ukážeme dále.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;

entity adder is
port (a,b,cin : in std_logic;
      s,cout : out std_logic);
end adder;

architecture Behavioral_concurrent of adder is
signal vstup : std_logic_vector(2 downto 0);
begin
vstup<=cin&b&a;
...
end Behavioral_concurrent;
```