Teoretický úvod – laboratorní úloha číslo 4

Realizace 2bitové sčítačky pomocí strukturálního návrhu s výstupem na displej

1 Strukturální popis v jazyce VHDL, komponenta

V případě předchozích způsobů popisu obvodů v jazyce VHDL, behaviorálního a dataflow (RTL), jsme neznali přesnou stavbu a zapojení navrhovaného obvodu, pouze jsme popsali jeho chování či způsob manipulace se vstupními hodnotami na výstup obvodu a vygenerování samotného zapojení a struktury obvodu jsme přenechali na automatickém procesu v rámci syntézy v programu Quartus. Oproti tomu nám umožňuje strukturální popis (`Structural`) v jazyce VHDL přesně definovat vnitřní stavbu a zapojení obvodu, a to klidně až do úrovně základních logických hradel. Jednou z výhodných vlastností jazyka VHDL je totiž možnost hierarchického rozložení celého obvodu na jednotlivé samostatné funkční subbloky, ve VHDL označované jako komponenty (`component`) a popisem jejich vzájemného propojení pak získat celkový výsledný obvod. Postupujeme přitom tak, že nejprve vytvoříme nejjednodušší požadovanou entitu v jazyce VHDL, např. logické hradlo, a tu pak využijeme při vytvoření výsledné entity obvodu, kterou získáme propojením několika těchto hradel, kde hradlo slouží jako komponenta této entity. Nemusíme však samozřejmě vždy obvod rozkládat až do úrovně logických hradel, například v této laboratorní úloze využijeme úplnou 1bitovou sčítačku jako komponentu pro realizaci 2bitové sčítačky. Další komponentou naší výsledné (top-level) entity bude i převodník z kódu BCD do kódu 7segmentového displeje z laboratorní úlohy č. 2.

Pro to, abychom mohli v nadřazené entitě (top-level entitě) použít nižší entitu jako komponentu, musíme jako první vytvořit VHDL modul této nižší entity. Poté musíme ve vyšší (nadřazené) entitě deklarovat v rámci její architektury mezi názvem architektury (`architecture`) a klíčovým slovem `begin`, že využijeme pro její realizaci danou komponentu (`component`). V jazyce VHDL k tomu slouží syntaxe:

```
component {nazev komponenty}
  port ({vycet portu komponenty, jejich smeru a datoveho typu});
end component;
```

Alternativní možností je vynechání uvedené deklarace komponenty a uvedení cesty a názvu entity, kterou využijeme jako komponentu, v rámci její volání v procesu mapování portů (`port map`), jak si popíšeme dále.

2 Signály, proměnné a konstanty v jazyce VHDL

Jazyk VHDL obsahuje 3 výše uvedené datové objekty, které lze použít pro různé účely. U všech z nich je vždy při deklaraci nutné definovat jejich datový typ obdobně jako v případě deklarace portů (lze využít všechny datové typy jako pro porty), na rozdíl od portů však nedefinujeme žádný směr (`in`, `out`, `buffer`, `inout`), neboť signály, proměnné i konstanty

lze využít pro čtení i zápis hodnot (do konstanty lze až na výjimku zapsat jen při její deklaraci, pak lze její hodnotu již jen číst). Výjimkou jsou tzv. procedury (`procedure`) v jazyce VHDL, v jejichž deklaracích signály, proměnné i konstanty mají určené směry; procedury však v této laboratorní úloze nebudeme uvažovat. Stejně jako v případě portů i 1bitové signály, proměnné a konstanty můžeme sdružovat a vytvářet tak vektory, např. `std_logic_vector`.

Signál (`signal`) lze ve VHDL přirovnat k funkci vodiče (spoje) v číslicovém systému a na rozdíl od proměnné a konstanty mu skutečně odpovídá HW realizace formou vodiče. Signály deklarujeme vždy v úvodu architektury mezi klíčovými slovy `architecture` a `begin` a jsou vždy globální, tzn. můžeme k nim přistupovat v libovolné části architektury, procesu apod. Již při deklaraci signálu do něho můžeme vložit jeho počáteční hodnotu pomocí symbolu `:=`[1], pokud zapisujeme hodnotu signálu v průběhu architektury, používáme stejně jako v případě portů symbol <=. Testování hodnoty signálu (např. v případě podmínek) je, stejně jako v případě portů, pomocí symbolu =. Při použití signálů v procesech musíme mít na paměti, že aktualizace (vložení) hodnoty do signálu se neprovede okamžitě, ale vždy až na konci běhu procesu po uplynutí tzv. delta time, více viz přednášky. V simulátorech (včetně Questy) lze obvykle zobrazit (simulovat) hodnoty signálů stejně jako portů entity.

Proměnná (`variable`) slouží v jazyce VHDL jako dočasné místo v paměti, kam lze například uložit výsledek výpočtu, stav čítače, hodnotu vstupního portu apod. Na rozdíl od signálu ji tedy neodpovídá žádná HW realizace v obvodu, jedná se o čistě abstraktní objekt. Lokální proměnnou (`variable`) deklarujeme jen v rámci procesu mezi klíčovými slovy `process` a `begin` a lze ji tak využít jen v daném konkrétním procesu, nikde jinde v architektuře či jiném procesu tato proměnná neexistuje. Globální (sdílenou) proměnnou (`shared variable`) deklarujeme stejně jako signály na počátku architektury mezi klíčovými slovy `architecture` a `begin`. Na rozdíl od lokální proměnné existuje sdílená proměnná v celé architektuře, můžeme proto do ní přistupovat v libovolném procesu i v paralelní části architektury a pracovat s ní stejně jako se signálem. Opět, jako v případě signálu, i do proměnné můžeme již při její deklaraci vložit počáteční hodnotu pomocí symbolu `:=`[2], rozdílem oproti signálu je, že v průběhu procesu (či architektury v případě sdílené proměnné) zapisujeme hodnotu do proměnné opět pomocí symbolu `:=`, pro testování hodnoty slouží opět jednoduché =. Zásadním rozdílem oproti signálům je, že po přiřazení nové hodnoty do proměnné v rámci procesu, se toto provede okamžitě na daném řádku kódu a bez čekání na aktualizaci na konci procesu, viz přednášky. Některé simulátory nedovolují simulovat a zobrazit hodnoty proměnných, simulátor Questa nainstalovaný v rámci programu Quartus však ano.

Konstanta (`constant`) je v pojetí jazyka VHDL proměnná, jejíž hodnotu můžeme definovat pouze při její deklaraci a pak ji již nemůžeme měnit, pouze číst. Výjimkou jsou konstanty v tzv. VHDL balíčcích (`package`), kdy lze hodnotu konstanty dodefinovat později (tzv. `deferred constant`), s těmi ale v rámci této laboratorní úlohy pracovat nebudeme. Konstantu můžeme deklarovat na počátku architektury mezi klíčovými slovy `architecture` a `begin` stejně jako signály a sdílené proměnné, v tom případě lze takovou konstantu používat

[1] Počáteční hodnota vložená do signálu či portu je však syntezátorem při syntéze a kompilaci navrhovaného obvodu a celého projektu obvykle ignorována a není ve skutečnosti provedena.

[2] Ta obvykle při syntéze obvodu při kompilaci projektu syntezátorem ignorována není.

v celé architektuře, nebo ji můžeme definovat pouze lokálně v konkrétním procesu mezi klíčovými slovy `process` a `begin` a obdobně, jako v případě proměnné, existuje tato konstanta jen v rámci daného procesu. Obecně lze konstanty využít pro zjednodušení a zpřehlednění kódu, jejich využití ve vhodných situacích rovněž může někdy vést ke zjednodušení obvodu při jeho syntéze a implementaci.

3 Mapování portů komponent v jazyce VHDL, port map

Jak jsme již uvedli, v případě strukturálního popisu v jazyce VHDL využíváme komponenty jako subbloky pro realizaci výsledné entity obvodu. Kromě deklarace vlastní komponenty, jak jsme uvedli v předchozí kapitole 1, musíme však také definovat propojení jednotlivých portů komponent mezi sebou, propojení portů komponenty na porty entity, případně na interní signály (vodiče). K tomu slouží v jazyce VHDL tzv. mapování portů pomocí klíčového slova `port map`. S ohledem na pevné přiřazení (propojení) portů/signálů jde o příkaz v paralelním prostředí jazyka VHDL, který umisťujeme přímo do těla architektury dané entity. Obecnou syntaxi příkazu `port map` můžeme zapsat například takto:

```
{oznaceni bloku v entite} : {nazev komponenty}
port map ({propojeni portu komponenty na porty/signaly entity});
```

První řádek specifikuje před dvojtečkou označení (jméno) daného subbloku v rámci vyšší entity, pojmenování subbloku musí být jednoslovné bez mezery, s vyloučením diakritiky, speciálních znaků a nesmí začínat číslem. Za dvojtečkou pak pokračuje název komponenty, pokud ta se nachází v souboru přímo ve stejném projektu a byla přímo deklarována pomocí klíčového slova `component` v úvodu architektury (viz kapitola 1), lze uvést jen jednoduše její jméno. Alternativně lze tento řádek upravit:

```
... : entity work.{nazev komponenty}({nazev architektury})
```

Pomocí klíčového slova `entity` označíme, že pro realizaci vyšší entity, tzv. top-level entity, budeme používat nižší entitu jako komponentu a pomocí klíčového slova `work.` vybereme, že daná nižší entita (komponenta vyšší entity) se nachází v pracovním adresáři tohoto projektu[3]. Dále v kulatých závorkách můžeme (ale nemusíme) specifikovat, která architektura z nižší entity se pro realizaci tohoto sub-bloku má použít, což může být užitečné, pokud daná nižší entita obsahuje více architektur. Výhodou tohoto druhého zápisu pro výběr komponenty je, že vůbec nemusíme v úvodní části architektury deklarovat komponentu, tak jak bylo popsáno v kapitole 1.

Vraťme se však ještě k druhému řádku příkazu `port map`. V jazyce VHDL rozeznáváme tzv. poziční mapování a jmenné mapování portů. Poziční mapování můžeme definovat:

[3] Klíčové slovo `work` v jazyce VHDL označuje dynamický prostor aktuálního projektu. Pokud se tedy komponenta (soubor s VHDL kódem komponenty) nachází přímo v adresáři tohoto projektu, můžeme se na něj jednoduše odkázat pomocí slova `work`. Jinak bychom museli specifikovat cestu na disku, kde se daný soubor nachází. Podobně můžeme využít klíčové slovo `work` např. i pro načítání knihoven a balíčků.

```
port map ({port/signal entity c. 1},{port/signal entity c.
2},...{port/signal entity c. n});
```

Do kulatých závorek příkazu `port map` vyjmenováváme porty nadřazené entity či interní signály, na které mají být připojeny porty dané komponenty, a to přesně v tom pořadí a pozici, v jakém jsou porty komponenty deklarovány. Tento způsob mapování má jednodušší a rychlejší zápis, na druhou stranu při velkém množství portů a signálů lze lehce udělat chybu, případně při pozdějších úpravách a modifikacích komponent a jejich portů je opět nutné upravit celý proces mapování a pečlivě doplnit/odstranit pořadí portů v mapování tak, aby souhlasilo dle aktuální deklarace portů. Z tohoto důvodu je někdy výhodnější jmenné mapování portů:

```
port map ({port komponenty}=>{port/signal entity},
          {port komponenty}=>{port/signal entity},
          ...
          {port komponenty}=>{port/signal entity});
```

V něm vždy vlevo nejprve uvedeme název portu komponenty a pomocí symbolu přiřazení => pak název portu či signálu entity, na který daný port připojujeme, jednotlivá přiřazování oddělujeme opět čárkou. Přiřazovat porty můžeme v libovolném pořadí, nemusíme tedy dodržovat pořadí portů při deklaraci komponenty.

4 Výběr architektury komponenty, configuration

Pokud je výsledný obvod (entita) složen z komponent a v dané komponentě je definováno více architektur, můžeme zvolit, která architektura bude vybrána individuálně při každém použití komponenty pro daný subblok. V jazyce VHDL se nabízí dvě možnosti. První způsob je definovat použitou architekturu při deklaraci subbloku (komponenty) v procesu mapování portů. Jak již bylo uvedeno výše, v takovém případě využijeme zápis:

```
... : entity work.{nazev komponenty}({nazev architektury})
```

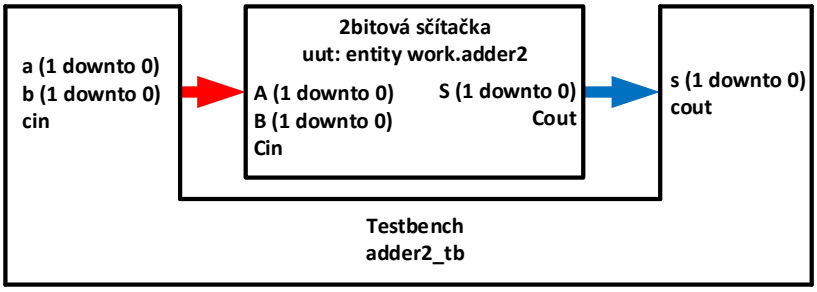
V kulatých závorkách za označením jména komponenty z ní tedy vybereme rovnou i požadovanou architekturu. Druhý způsob představuje použití klíčového slova `configuration`, který si ale představíme na přednášce předmětu.

5 Simulace v jazyce VHDL

Jednou z výhod jazyka VHDL je, že kromě vlastní syntézy digitálních obvodů jej lze použít i pro jejich simulaci, a to navíc v různých fázích syntézy a implementace výsledného digitálního obvodu. V prostředí Quartus je pro tyto účely možné využít vestavěný simulátor Questa Intel FPGA, volitelně lze samozřejmě doinstalovat libovolný jiný simulátor. Pro účely simulace je nutné nejprve vytvořit vlastní entitu obvodu (v jazyce VHDL, použitím schematického editoru aj.), kterou chceme simulovat. Tato entita je v simulátoru označena jako UUT (Unit Under Test). Dále je nutné vytvořit tzv. testbench, což je VHDL kód určený pro simulaci a testování dané entity – slouží pro definované buzení vstupů testované entity (UUT) a pro shromáždění výstupů testované entity a jejich zobrazení v čase. Testování UUT entity probíhá tak, že v testbench souboru jsou zapsány tzv. stimuly – v podstatě časová posloupnost

hodnot budících signálů, které jsou připojeny (namapovány) k portům UUT entity a na opačné straně pak simulátor ukládá výstupy dané UUT entity, jako její odezvu na dané buzení. Tyto výstupy lze pak zobrazit v časovém průběhu, případně společně i s budícími vstupy. Kromě vstupů/výstupů testované entity lze simulovat i chování (hodnoty) interních signálů entity a v některých simulátorech i interních proměnných[4]. Pro VHDL testbench lze využít kromě syntetizovatelných příkazů jazyka VHDL i příkazy (konstrukce) aplikovatelné pouze pro simulace. Mezi ně patří například příkaz `wait for {časový údaj}`, přiřazení hodnoty po určitém čase `after {časový údaj}` a další. Simulátory dále obvykle umožňují simulovat danou entitu v různých fázích její syntézy a implementace, provést simulaci na úrovni komponent, procesů, hradel apod.

Uveďme si pro lepší názornost krátký příklad zápisu VHDL testbenche, který následně využijeme i v rámci této laboratorní úlohy pro simulaci navržené 2bitové sčítačky. Entitu tohoto VHDL testbenche pojmenujeme *adder2_tb* (kde zkratku „tb“ v názvech souborů a entit používáme obvykle právě pro testbenche).



Obr. č. 1: Blokové schéma pro testbench 2bitové sčítačky.

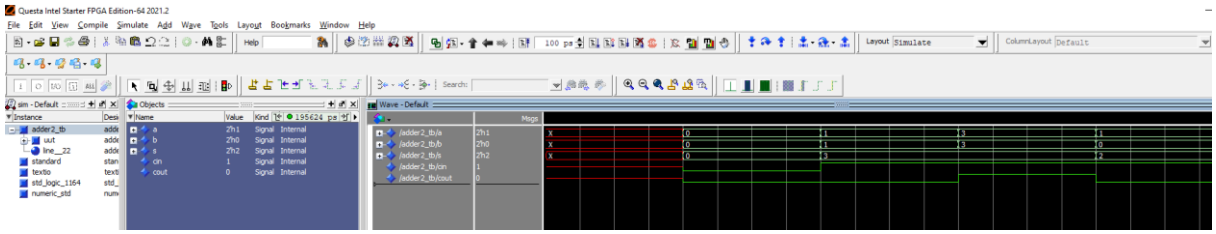
Obr. č. 1 ukazuje, jak budeme simulovat entitu sčítačky s názvem `adder2`: budeme konkrétně budit 2bitové vstupy sčítačky `A`, `B` typu `std_logic_vector (1 downto 0)` a 1bitový vstup přenosu z nižšího řádu `Cin` typu `std_logic`, výstupem 2bitové sčítačky je 2bitový výstup součtu `S` typu `std_logic_vector (1 downto 0)` a 1bitový přenos do vyššího řádu `Cout std_logic`. V rámci testbenche s názvem `adder2_tb` vytvoříme signály se stejnými názvy, jako jsou porty sčítačky, jen pro lepší odlišení použijeme pro jejich názvy malá písmena, tzn. `a`, `b`, `cin`, `s`, `cout`. VHDL kód testbenche tak bude vypadat následovně:

[4] Questa v prostředí programu Quartus toto umožňuje.

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

entity adder2_tb is
end adder2_tb;

architecture Behavioral of adder2_tb is

signal a,b,s : std_logic_vector(1 downto 0);
signal cin,cout : std_logic;

begin

uut: entity work.adder2
port map (A=>a,
          B=>b,
          Cin=>cin,
          S=>s,
          Cout=>cout);

process
begin
wait for 40 ns;
a<="00";
b<="00";
cin<='0';
wait for 40 ns;
...
wait;
end process;

end Behavioral;
```

Pomocí kódu na řádku `uut: entity work.adder2` provedeme deklaraci komponenty, kterou je simulovaná 2bitová sčítačka `adder2`, a dále pomocí `port map` provedeme namapování jejích portů na signály v rámci testbenchové entity. Dále pak v procesu zapíšeme časovou souslednost buzení těchto stimulů (signálů), nejprve na začátku počkáme 40 ns[5], po něm vložíme do signálů sloužících pro buzení vstupů 2bitové sčítačky hodnoty `a<="00"; b<="00"; cin<='0';` a opět počkáme 40 ns, během kterých analyzujeme výstupy sčítačky. Poté můžeme nastavit jiné stimuly atd. Poslední příkaz procesu, `wait;` bez specifikace doby čekání, představuje nekonečně dlouhé čekání a další změny na vstupu sčítačky tedy již nebudou probíhat. Obr. č. 2 ukazuje časový průběh na vstupech a výstupech simulované sčítačky na základě buzení pomocí uvedeného testbenche v okně Questy.

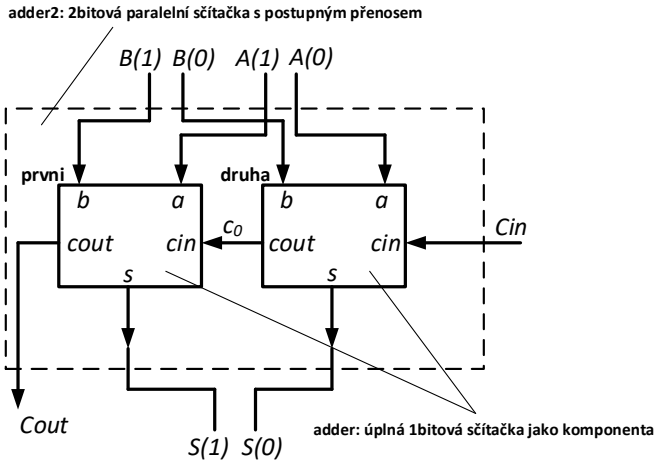
[5] Čekání na začátku simulace, např. 40 či 100 ns, je dobrým zvykem při simulacích v jazyce VHDL. Toto čekání slouží pro inicializaci všech obvodů a FPGA pole před samotným buzením.



Obr. č. 2: Výstup simulace v programu Questa Intel FPGA.

6 2bitová sčítačka s výstupem na displej v jazyce VHDL

V rámci předchozí laboratorní úlohy č. 3 jsme realizovali úplnou 1bitovou sčítačku pomocí několika různých způsobů (architektur), s využitím behaviorálního popisu a podmínek v paralelním i sekvenčním prostředí, pomocí RTL popisu formou Booleových rovnic pro výstupní funkce a případně též operací sčítání s konverzí datových typů. Rovněž pro realizaci 2bitové sčítačky se nabízí několik odlišných možností; v rámci této úlohy se zaměříme na použití strukturálního popisu, využijeme připravené realizace 1bitové sčítačky z předchozí úlohy jako komponenty a 2bitovou sčítačku vytvoříme jako paralelní sčítačku s postupným přenosem. Tuto realizaci si můžeme jednoduše představit jako operaci sčítání dvojice binárních čísel, kdy od konce postupně sčítáme vždy jednotlivá řádová místa navzájem a případný přenos z tohoto řadového místa přičítáme do nejbližšího vyššího. Tímto způsobem postupně provedeme sečtení na všech řádových místech a získáme výsledek. Obr. č. 3 obecně znázorňuje strukturu zapojení takové 2bitové paralelní sčítačky.



Obr. č. 3: Strukturální návrh paralelní 2bitové sčítačky s postupným přenosem.

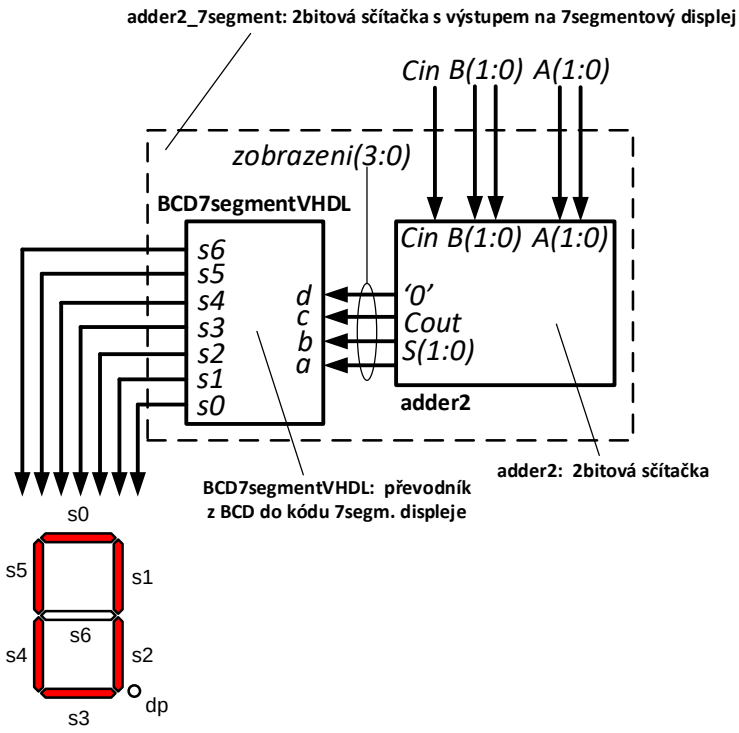
Vstupem jsou tedy 2bitové logické vektory `A`, `B : std_logic_vector(1 downto 0)` a 1bitový vstup přenosu z nižšího řádu `Cin : std_logic`. Výstupem je 2bitový logický vektor jako součet `S : std_logic_vector(1 downto 0)` a 1bitový výstup přenosu do vyššího řádu `Cout : std_logic`. Dále budeme potřebovat jeden 1bitový logický signál (vodič) pro propojení přenosů obou komponent 1bitových sčítaček, např. `C0 : std_logic`. Založíme tedy novou VHDL entitu s názvem např. *adder2* s výše uvedenými porty. V rámci její architektury deklarujeme použití komponenty *adder* 1bitové sčítačky z předchozí laboratorní úlohy č. 3 a dále signálu pro propojení přenosu. Obr. č. 3 ukazuje, jak provedeme dvojí mapování portů na dvojici komponent 1bitových sčítaček s využitím zapojení.

Kód č. 1

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

entity adder2 is
port (A,B : in std_logic_vector(1 downto 0);
      Cin : in std_logic;
      S : out std_logic_vector(1 downto 0);
      Cout : out std_logic);
end adder2;

architecture Structural of adder2 is

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

signal C0 : std_logic:='0';
begin

prvni : entity work.adder(Behavioral_Concurrent)
port map (A(0),...); -- misto tecek doplnte spravne mapovani

druha : entity work.adder(RTL)
port map (A(1),...); -- misto tecek doplnte spravne mapovani

end Structural;
```

Nakonec ještě zbývá připojit k výstupu 2bitové sčítačky 7segmentový displej pro zobrazení výsledku. Můžeme opět s výhodou použít strukturální popis. Vytvoříme novou VHDL entitu, pojmenujeme ji např. *adder2_7segment* a uvedeme ji jako top-level entitu celého projektu. Jejími vstupy budou vstupy 2bitové sčítačky *adder2*, tedy 2bitové logické vektory `A`, `B : std_logic_vector(1 downto 0)` a 1bitový vstup přenosu z nižšího řádu `Cin : std_logic`. Výstupem bude 7 segmentů 7segmentového displeje, tedy `s0`, `s1`, `s2`, `s3`, `s4`, `s5`, `s6 : std_logic`. V architektuře deklarujeme použití dvojice komponent – 2bitové sčítačky *adder2* vytvořené v této laboratorní úloze č. 4 výše a dále převodníku z kódu BCD na kód 7segmentovéeho displeje *BCD7segmentVHDL*, který jsme vytvořili v laboratorní úloze č. 2. Pro propojení obou komponent a pro vyvedení výstupu 2bitové sčítačky na vstup komponenty 7segmentového displeje pro zobrazení výsledku na displeji budeme dále potřebovat signál sběrnici (vektor) typu std_logic_vector o velikosti 3 bitů, např. tedy `zobrazeni : std_logic_vector(3 downto 0)`. Po založení entity *adder2_7segment* s porty a signálem viz výše pak ve vlastní části popisu architektury využijeme opět dvojici příkazů *port map*, kdy v prvním namapujeme zapojení 2bitové sčítačky a jejích portů, zatímco druhým příkazem *port map* pak připojíme 7segmentový displej pro zobrazení výsledku. Vzhledem k tomu, že výstupem 2bitové sčítačky na displej mohou být číslice

v rozsahu 0-7 v dekadické soustavě, bude nejvyšší bit sběrnice zobrazeni trvale na hodnotě logická 0 a nebude využit. První dva bity na nižších pozicích sběrnice využijeme pro připojení součtu `S` 2bitové sčítačky, zatímco v pořadí třetí bit připadne na výstup přenosu `Cout`. Poslední čtvrtý bit zůstane nevyužit. Obr. č. 4 představuje schematické zapojení celé entity a obou komponent.



Obr. č. 4: Výsledný strukturální návrh 2bitové sčítačky s výstupem na 7segmentový displej.

Obr. č. 4 ukazuje, jak můžeme zapsat základ a část VHDL kódu pro výsledné zapojení komponent:

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

entity adder2_7segment is
port (...); -- sem doplnte deklaraci portu entity
end adder2_7segment;

architecture Structural of adder2_7segment is

component adder2 is
port (A,B : in std_logic_vector(1 downto 0);
      ...); -- zde doplnte deklaraci portu komponenty adder2
end component;

component BCD7segmentVHDL is
port (...); -- zde doplnte porty komponenty BCD7segmentVHDL
end component;

signal zobrazeni : std_logic_vector(3 downto 0) := "0000";

begin

scitacka : entity work.adder2
port map (...); -- zde namapujte porty komponenty adder2

displej : entity work.BCD7segmentVHDL
port map (...); -- zde namapujte porty komponenty
BCD7segmentVHDL

end Structural;
```