

# **Parametrizace VHDL kódu, procedury, funkce, knihovny**

Parametrizace VHDL kódu, vytvoření a použití procedur, funkcí,  
balíčků a knihoven

Ing. Pavel Lafata, Ph.D.  
lafatpav@fel.cvut.cz

## VHDL – Parametrizace VHDL kódu, generic

- **Opakování – parametrizace VHDL kódu – generic**

- generic – slouží pro parametrizaci návrhu entity, komponenty v jazyce VHDL
- parametrizace nám umožňuje jednoduše z jednoho místa měnit hodnoty v různých částech VHDL kódu – parametrizovat navržený obvod
- např. můžeme měnit rozsah integeru, délku vektoru, počet cyklů smyčky, počet pozic rotace/posuvu registru, aritmetické operace, porovnání hodnoty v podmínce....
- hodnoty tohoto parametru (generic) jsou dostupné v celé architektuře (globální), nebo můžeme parametrizovat jen určitou komponentu (port map)
- generic deklarujeme v první části entity (jako porty) a definujeme jejich datový typ a hodnotu, na rozdíl od portů však ne směr (nejsou směrové)
- nejedná se o port, ačkoliv jej deklarujeme na počátku entity (komponenty) a můžeme rovněž provádět jeho mapování – ale jedná se o parametr, kterým můžeme ovládat (specifikovat) různé části VHDL kódu

- **generic syntaxe:**

**entity** *název entity* **is**

**generic** (*seznam generic*);

**port** (*seznam portů*);

**end** *název entity*;

- ***seznam generic*** – *název* : *datový typ* := *počáteční hodnota*;

## VHDL – Parametrizace VHDL kódu, generic

- **Parametrizace VHDL kódu – generic**

- příklad – posuvný registr s volitelnou délkou, jednou rotací vlevo a 2bitovým výstupem

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

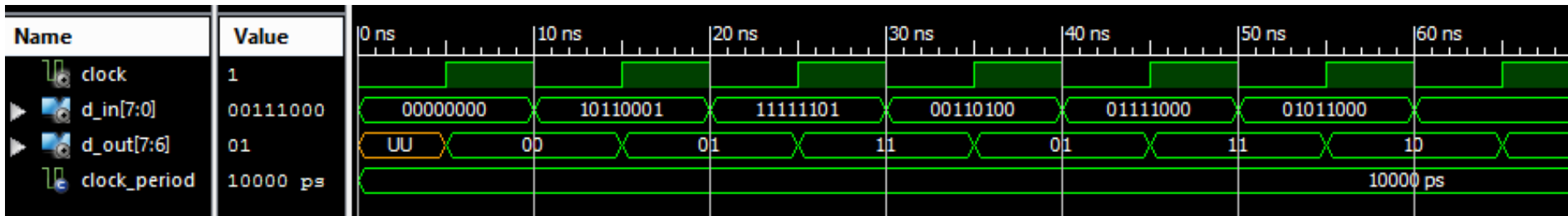
entity reg_generic is
generic (width : integer := 7);
port (Clock : in std_logic;
      D_in : in std_logic_vector(width downto 0);
      D_out : out std_logic_vector(width downto width-1));
end reg_generic;
architecture Behavioral of shift_generic is
begin
process(Clock)
variable reg : std_logic_vector (width downto 0);
begin
if Clock='1' and Clock'event then
    reg := D_in;
    reg := reg(width-1 downto 0) & reg(width);
    D_out<=reg(width downto width-1);
end if;
end process;
end Behavioral;
```

- **klíčové slovo generic** – v části entity před deklarací portů
- parametr **width** – délka registru – vložíme do něj např. 7 (8bitový registr)
- **generic** – bez směru
- nyní můžeme parametr **width** použít jednoduše v deklaracích a práci s ostatními porty, signály a proměnnými
- parametr **width** můžeme použít i v rámci různých operací – např. zřetězení (&)

## VHDL – Parametrizace VHDL kódu, generic

- **Parametrizace VHDL kódu – generic**

- simulace předchozího registru pro parametr width = 7
- 8bitový registr, 1bitová rotace vlevo a 2bitový výstup (7 downto 6)



- **výhoda generic** – v příkladu změníme jen hodnotu parametru *width* a hned máme jiný *n*-bitový registr
- použití parametrizace generic pro komponenty
- pokud chceme parametrizovat komponentu, deklarujeme generic i mezi jejími porty
- následně použijeme tzv. generic map – obdoba mapování portů pro parametry generic
- stejné dvě metody – **poziční mapování** a **jmenné mapování generic**:  
**generic map (mapování pro port1, mapování pro port2,...)**  
**generic map (port1 komponenty =>mapování, port2 komponenty=>mapování,...)**
- po namapování generic portů provedeme standardní **mapování portů**
- **za příkaz mapování generic portů nepíšeme středník „;“** – mezi **generic map ()** a **port map ()**

## VHDL – Parametrizace VHDL kódu, generic

- **Parametrizace VHDL kódu – generic**

- příklad – použijme předchozí návrh parametrizovaného registru jako komponentu a vytvoříme registr složený ze 2 těchto komponent – první část vstupního vektoru je přivedena na první komponentu s danou délkou, druhá část na druhou komponentu registru s jinou délkou
- použijeme parametry generic pro určení délek obou částí registru – width\_reg1, width\_reg2

```
entity reg_top is
generic (width_reg1 : integer :=5;
        width_reg2 : integer :=10);
port (Clock : in std_logic;
      Input : in std_logic_vector(width_reg1+width_reg2 downto 0);
      Output : out std_logic_vector(width_reg1+width_reg2 downto
width_reg1+width_reg2-3));
end reg_top;
```

- width\_reg1 – délka prvního registru, width\_reg2 – délka druhého registru
- Input – celková délka celého registru musí být **width\_reg1+width\_reg2 downto 0**
- Output – celková délka výstupního vektoru musí být 4, protože délka každého z obou výstupů je 2

## VHDL – Parametrizace VHDL kódu, generic

- **Parametrizace VHDL kódu – generic**

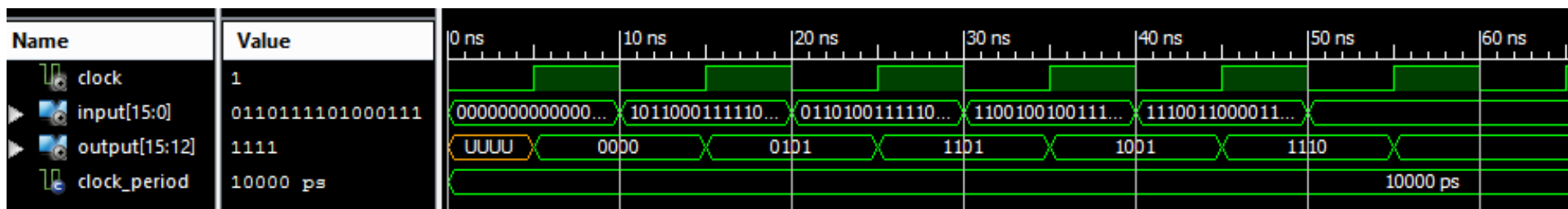
```
architecture Behavioral of reg_top is
component reg_generic is
generic (width : integer);
port (Clock : in std_logic;
      D_in : in std_logic_vector(width downto 0);
      D_out : out std_logic_vector(width downto width-1));
end component;
begin
reg1 : reg_generic
generic map (width_reg1-1)
port map (Clock, Input(width_reg1+width_reg2 downto width_reg2+1),
Output(width_reg1+width_reg2 downto width_reg1+width_reg2-1));
reg2 : reg_generic
generic map (width=>width_reg2)
port map (Clock, Input(width_reg2 downto 0), Output(width_reg1+width_reg2-2 downto
width_reg1+width_reg2-3));
end Behavioral;
```

- deklarujeme komponentu, nesmíme zapomenout **deklarovat** i její **generic parametr** - *width*
- následně definujeme použití 2x této komponenty – jako první musíme mapovat generic porty (**generic map**), poté mapujeme zbylé standardní porty (**port map**)

## VHDL – Parametrizace VHDL kódu, generic

### • Parametrizace VHDL kódu – generic

- pro **první generic mapování** použijeme např. **poziční způsob** – protože komponenta obsahuje jen jeden generic port, je situace jednoduchá
- pro ukázkou – **pro generic mapování druhé komponenty** použijeme **jmenné mapování** s použitím symbolu =>
- následně provedeme mapování zbylých standardních portů obou komponent, mezi generic mapování a mapování portů se nepíše středník!
- uvnitř mapování portů jednotlivých komponent **nahradíme původní parametr *width* pomocí generic parametrů deklarovaných v entitě** (*width\_reg1*, *width\_reg2*)
- protože počáteční hodnota parametru *width* deklarovaná v komponentě (*width* : integer := 7) je nahrazena hodnotami parametrů entity (*width\_reg1* : integer :=5; *width\_reg2* : integer :=10), můžeme původní počáteční hodnotu vypustit
- simulace navrženého obvodu



- vstupní vektor je rozdělen na 2 části – 5bitovou část a 11bitovou část, každá část je samostatně rotována o 1 pozici vlevo a 2 nejvyšší bity z každé části jsou vysunuty z registru ven, výstup tedy obsahuje 4 bity

## **VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny**

- **Strukturování VHDL kódu – nemá vliv na implementaci, jen zvyšuje přehlednost!**
  - **jazyk VHDL podporuje modularitu** – mapování komponent a bloků, deklarování vlastních funkcí a knihoven... – nemá vliv na syntézu ani implementaci!
  - v jazyce VHDL můžeme využít – **procedury, funkce, balíčky, knihovny...**
  - základní idea – uložit si často používané konstrukce, části kódu nebo i celé bloky obvodů a pak je jednoduše využívat v různých návrzích – rychlost, přehlednost
  - **balíček (package)** = skupina obvykle příbuzných (podobných) funkcí a procedur může být uložena v jednom speciálním VHDL modulu = balíčku (package)
  - **knihovna (library)** = soubor několika příbuzných balíčků (nejvyšší modulární jednotka)
  - do knihoven a balíčků můžeme také deklarovat své datové typy, konstanty, funkce, procedury a celé obvody a pak je používat v nejrůznějších návrzích
- **funkce a procedury**
  - obvykle sdružujeme do balíčků a knihoven, ale můžeme zapisovat přímo i v architektuře
  - **funkce**
    - může mít 0 a více vstupních parametrů, výstupem je vždy jen jedna hodnota
    - vstupem může být pouze signál nebo konstanta
  - **procedura**
    - libovolné množství in, out, inout parametrů, mohou to být signály, proměnné, konst.
    - výstupem může být signál, proměnná, konstanta, defaultní je proměnná
  - **funkce je volána jako část nějakého příkazu (výrazu), procedura je příkaz sám o sobě**



## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **funkce (function)**

- vypočítá a vrátí hodnotu zadaného datového typu s použitím vstupních parametrů
- parametry jsou použity uvnitř funkce, ale nesmí být změněny
- parametry jsou pouze vstupy (nikoliv výstupy), nemusíme proto definovat jejich směr
- určíme pouze jejich datový typ – výstupem může být signál nebo konstanta, defaultní je konstanta

- **funkce syntaxe:**

[pure | impure] **function** *název (seznam parametrů)* **return** *datový typ* **is**  
{*deklarace proměnných*}

**begin**

...

**end** [function] *název*;

- **pure funkce** – pokaždé vrátí stejnou hodnotu při zadání stejných vstupních parametrů (defaultní typ funkce – není potřeba specifikovat)
- **impure funkce** – může vrátit různou hodnotu na výstupu ačkoliv zadáme stejné vstupní parametry, tento typ funkce může přistupovat ke všem signálům a sdíleným proměnným deklarovaným v architektuře, není tak potřeba deklarovat seznam parametrů (signálů, proměnných, portů) – příklad impure funkce: generátor náhodných čísel

- **Funkce – příklad**

**entity** func is

```
port(Data_in : in std_logic_vector(7 downto 0);  
      Count : out integer range 0 to 8);  
end func;
```

**architecture** Behavioral **of** func is

```
function ones (signal data : std_logic_vector) return integer is  
variable temp : integer range 0 to data'length;
```

```
begin
```

```
for i in 0 to data'length-1 loop
```

```
    if data(i) = '1' then
```

```
        temp := temp+1;
```

```
    end if;
```

```
end loop;
```

```
return(temp);
```

```
end ones;
```

```
begin
```

```
process(Data_in)
```

```
begin
```

```
Count<=ones(Data_in);
```

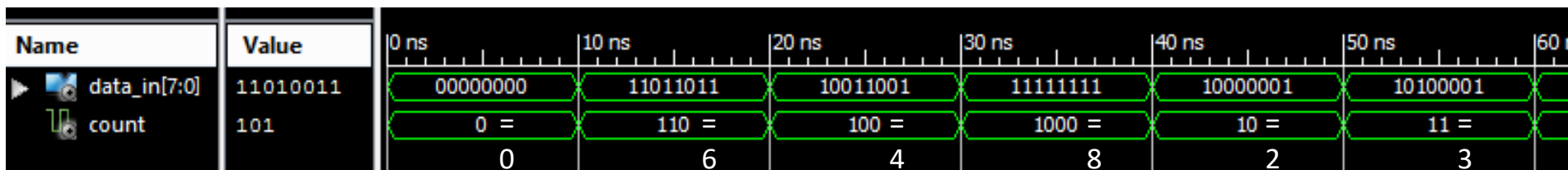
```
end process;
```

```
end Behavioral;
```

# VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

## • Funkce

- příklad funkce – funkce s názvem **ones**, počítá výskyt log. jedniček v std\_logic\_vector
- **signal data** – vstupní parametr – zadaný vektor, std\_logic\_vector, výstup je integer (počet log. jedniček ve vektoru), proměnná *temp* je použita pro počítání log. jedniček
- **for loop** – smyčka for – projde postupně všechny pozice vektoru a spočítá log. jedničky
- vlastní kód použití funkce – jeden proces, spouštěný změnou vstupu Data\_in
- zavoláme funkci *ones* a její výsledek vložíme na výstup Count: **Count<=ones(Data\_in);**
- obecně, funkce může obsahovat libovolné příkazy v jazyce VHDL, kromě podmíněného (paralelního) přiřazování a příkazu wait
- funkce konverze datových typů – mohou být použity přímo při mapování portů pro konverzi datových typů mapovaného portu a jeho přiřazení
- funkce mohou být volány jak v paralelním (concurrent) tak sekvenčním (sequential) prostředí jazyka VHDL



## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Funkce – příklad**
- příklad **impure funkce** – počítání s náhodným číslem
- deklarujeme **sdílenou proměnnou** v architektuře – je přístupná ve všech funkcích a procesech
- výsledek ukázkové funkce je pokaždé jiný – protože vnitřní proměnná *counter* se při každém vyvolání funkce změní

```
entity func2 is
```

```
port(data_in : in integer;  
      data_out : out integer);  
end func2;
```

```
architecture Behavioral of func2 is  
shared variable number : integer;
```



```
impure function calc (A : integer) return integer is  
variable counter : integer;  
begin  
  counter := A*number;  
  number := number + 1;  
  return counter;  
end calc;
```

```
begin  
process(data_in)  
begin  
  data_out<=calc(data_in);  
end process;  
end Behavioral;
```

# VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Funkce**

- simulace příkladu impure funkce

Name	Value	0 ns	10 ns	20 ns	30 ns	40 ns	50 ns	60 ns
 data_in	14	2147483648	8	14	8	14	8	
 data_out	84	0	8	28	24	56	40	

- předchozí příklady představovaly deklaraci funkcí přímo v architektuře dané entity – častěji ale funkce zapisujeme v balíčcích a knihovnách a vyvoláváme odtud, viz dále

# VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Procedura**

- podobná jako funkce, ale můžeme použít **libovolný počet** in, out, inout proměnných, signálů, konstant...
- procedura nevrací jednu výstupní hodnotu – mění zadané výstupy
- procedura může obsahovat jak paralelní tak sekvenční příkazy, může být vykonávána jak sekvenčně tak i paralelně
- procedura může být umístěna přímo v architektuře – pak je přístupná v celé entitě, nebo v procesu – pak je přístupná jen v tomto procesu
- seznam parametrů – obsahuje in/out/inouts – procedura je jediná konstrukce ve VHDL, kdy **při deklaraci proměnných a signálů musíme určit jejich směr (in, out, inout)!**
- **procedura syntaxe:**  
**procedure** *název (seznam parametrů)* **is**  
*{deklarace proměnných}*  
**begin**  
...  
**end** [procedure] *název*;
- **volání procedury – podobně jako při mapování portů:**
  1. **poziční** – musíme uvést parametry (vstupy/výst.) přesně v tomtéž pořadí, jaké odpovídá vstupům a výstupům dané procedury, žádné dodatečné symboly nejsou potřeba
  2. **jmenné** – můžeme uvést parametry (vst./výst.) v libovolném pořadí vůči vstupům a výstupům procedury pomocí symbolu: *název parametru procedury => název objektu v entitě*

# VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Procedura**

- příklad použití procedury – 2bitová sčítačka použitím 1bitové sčítačky jako procedury

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

architecture Behavioral of proc is
  procedure add (a, b, carry_in : in std_logic; signal s, carry_out : out std_logic) is
  begin
    s<=a xor b xor carry_in;
    carry_out<= (a and b) or (carry_in and (a xor b));
  end add;
  signal s1 : std_logic;
  begin
    process(A, B, Cin)
    begin
      add(A(0), B(0), Cin, S(0), s1);
      add(A(1), B(1), s1, S(1), Cout);
    end process;
  end Behavioral;
```

## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Procedura**

- příklad – procedura 1bitová sčítačka a její použití pro vytvoření 2bitové sčítačky
- 2bitové vstupy A, B, 2bitový výstup S, 1bitové přenosy Cin, Cout
- **procedura add – vytvořili jsme proceduru 1bitové sčítačky**
  - nejprve **definujeme parametry** – pokud in/out/inout v deklaraci chybí – **in je defaultní**
  - pokud chybí typ, **defaultní je proměnná (variable)**
  - vytvoříme samotnou sčítačku (proceduru) **add** – RTL model sčítačky
  - **žádný příkaz return jako v případě funkce** – procedura nic „nevrací“!
- pak použijeme jen proces a zavoláme 2x proceduru – pro 2bitovou sčítačku
- pro propojení přenosu mezi oběma 1bitovými sčítačkami – potřebujeme **signál s1** („drát“)
- proceduru můžeme zavolat i v paralelním prostředí bez použití procesu:

```
architecture Behavioral of proc is
```

```
signal s1 : std_logic;
```

```
begin
```

```
add(a=>A(0), b=>B(0), carry_in=>Cin, s=>S(0), Carry_out=>s1);
```

```
add(a=>A(1), b=>B(1), carry_in=>s1, s=>S(1), Carry_out=>Cout);
```

```
end Behavioral;
```

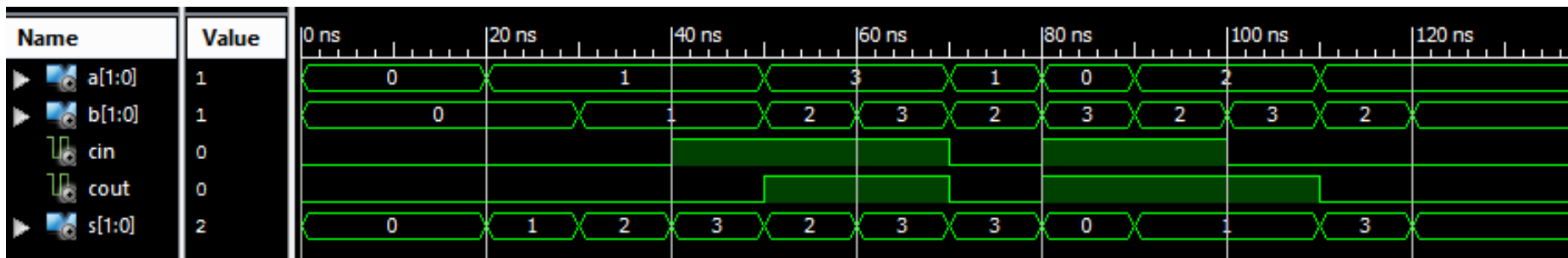
- pro ukázkou **v sekvenčním prostředí bylo použito poziční mapování a v paralelním jmenné mapování vstupů a výstupů procedury** – můžeme samozřejmě libovolně kombinovat



# VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

## • Procedura

- příklad – procedura 1bitová sčítačka a její použití pro vytvoření 2bitové sčítačky
- behaviorální simulace:



- **volání procedury v paralelním vs. sekvenčním prostředí**
  - **paralelní volání** – spouštění a vykonání procedury je paralelní s ostatními příkazy v paralelním prostředí, procedura je spuštěna při detekci změny jakéhokoliv signálu či sdílené proměnné v jejím seznamu parametrů, seznam parametrů může obsahovat jen signály, sdílené proměnné a konstanty (nikoliv proměnné)
  - **sekvenční volání** – procedura je volána v rámci procesu, je zavolána v sekvenčním pořadí příkazů vykonávaných procesem, seznam jejich parametrů může obsahovat i proměnné deklarované v tomto procesu
- **v proceduře můžeme volat jinou proceduru!**
- nápad – **vytvoříme proceduru 2bitové sčítačky ve které budeme volat procedury 1bitových sčítaček**

## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Procedura**

- vytvoříme proceduru 2bitové sčítačky zavoláním 2 procedur 1bitových sčítaček

```
architecture Behavioral of proc is
```

```
signal s1 : std_logic;
```

```
procedure add (a, b, carry_in : std_logic; signal s : out std_logic; signal carry_out : out std_logic) is  
begin
```

```
s<=a xor b xor carry_in;
```

```
carry_out<= (a and b) or (carry_in and (a xor b));
```

```
end add;
```

```
procedure add2 (A, B : std_logic_vector(1 downto 0); carry_in : std_logic; signal S : out  
std_logic_vector(1 downto 0); signal carry_out : out std_logic; signal s1 : inout std_logic) is  
begin
```

```
add(a=>A(0), b=>B(0), carry_in=>carry_in, s=>S(0), carry_out=>s1);
```

```
add(a=>A(1), b=>B(1), carry_in=>s1, s=>S(1), carry_out=>carry_out);
```

```
end add2;
```

```
begin
```

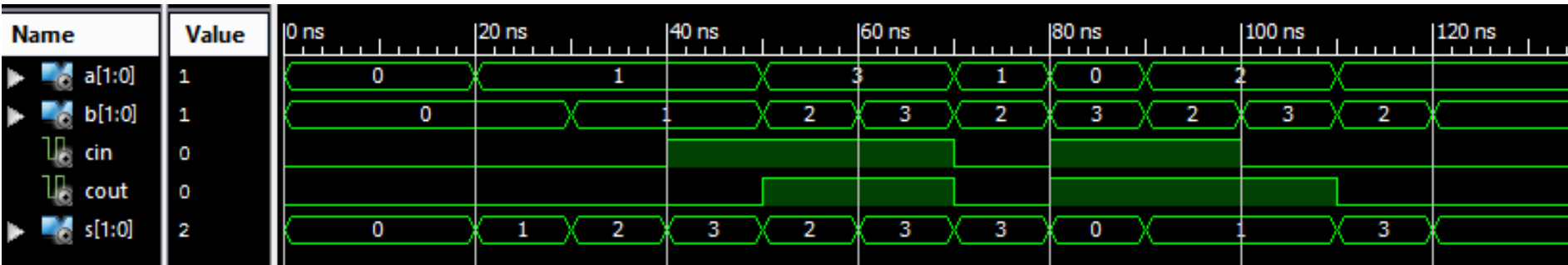
```
add2(A, B, Cin, S, Cout, s1);
```

```
end Behavioral;
```

## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Procedura**

- procedura 1bitová sčítačka – *add* beze změny
- procedura 2bitová sčítačka *add2* – deklarujeme seznam parametrů – vstupní porty A, B 2bitové sčítačky, 1bitový vstup přenosu, 2bitový výstup S, 1bitový výstup přenosu
- musíme deklarovat signál s1 – propojit přenos mezi oběma 1bitovými sčítačkami – musíme jej deklarovat v seznamu parametrů procedury *add2* jako inout (se signálem s1 provádíme oboje – první větev sčítačky do něj zapisuje přenos, druhá jej čte)
- realizace 2bitové sčítačky – procedura *add2* - v ní dvakrát voláme proceduru *add* 1bitové sčítačky, použili jsme jmenné přiřazování parametrů
- behaviorální simulace – stejný testbench jako v předchozí ukázce – stejný výsledek



- pokud potřebujeme signál propojující procedury – typ inout
- v rámci procedury můžeme deklarovat pomocné proměnné, signály ne

## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Balíčky (package)**

- jednoduchý způsob jak sdílet a přenášet již vytvořené bloky VHDL kódu, navržené obvody, komponenty, procedury a funkce... mezi jednotlivými VHDL moduly a návrhy
- do jednoho balíčku obvykle sdružíme funkce, procedury atd. týkající se jedné problematiky – např. balíčky s komunikačními protokoly, balíčky pro konverze...
- balíček obsahuje – **2 části** – deklarace balíčku, tělo balíčku
- **deklarace** – seznam (obsah) funkcí, procedur, komponent, datových typů...  
**tělo** – vlastní funkční části (VHDL kódy) jednotlivých funkcí, procedur, komponent...

- **syntaxe balíčku:**

**package** *název* **is**

*{deklarace balíčku}*

**begin**

*{tělo balíčku}*

**end** [package] [*název*];

- zavolání balíčku příkazem:  
**use cesta.název balíčku.název funkce, procedury... nebo all**
- příklad: **use ieee.numeric\_std.all** – použij všechny funkce, procedury, deklarace (vše) v balíčku s názvem numeric\_std umístěném v adresáři ieee (automaticky načtený adresář)
- cesta – klíčové slovo **work** – pracovní adresář (prostor) daného projektu, jinak lze specifikovat standardně cestu k adresáři projektu

## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Balíčky – příklad, předchozí procedury add, add2 vytvoříme balíček: my\_package**

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package my_package is
  procedure add (a, b, carry_in : std_logic; signal s : out std_logic; signal carry_out : out std_logic);
  procedure add2 (A, B : std_logic_vector(1 downto 0); carry_in : std_logic; signal S : out
std_logic_vector(1 downto 0); signal carry_out : out std_logic; signal s1 : inout std_logic);
end my_package;

package body my_package is
  procedure add (a, b, carry_in : std_logic; signal s : out std_logic; signal carry_out : out std_logic) is
  begin
    s<=a xor b xor carry_in;
    carry_out<= (a and b) or (carry_in and (a xor b));
  end add;
  procedure add2 (A, B : std_logic_vector(1 downto 0); carry_in : std_logic; signal S : out
std_logic_vector(1 downto 0); signal carry_out : out std_logic; signal s1 : inout std_logic) is
  begin
    add(a=>A(0), b=>B(0), carry_in=>carry_in, s=>S(0), carry_out=>s1);
    add(a=>A(1), b=>B(1), carry_in=>s1, s=>S(1), carry_out=>carry_out);
  end add2;
end package body my_package;
```

## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Balíčky – příklad, předchozí procedury `add`, `add2` vytvoříme balíček: `my_package`**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.my_package.all;

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

architecture Behavioral of add_proc is
signal s1 : std_logic;
begin
add2(A, B, Cin, S, Cout, s1);
end Behavioral;
```

- použijeme náš balíček – využijeme z něho proceduru *add2* – 2bitovou sčítačku

## VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Knihovny – vytvoření vlastní knihovny my\_library s balíčkem my\_package**

```
library my_library;  
use my_library.my_package.all;
```

- **knihovna** (library) – ve VHDL představuje v podstatě adresář s uloženými balíčky
- jak vytvořit a pracovat s knihovnou? náš příklad – nejprve vytvoříme **VHDL knihovnu** – adresář s názvem my\_library, který obsahuje balíček my\_package.vhd
- při založení nového projektu musíme přidat (načíst) naši knihovnu do návrhu, protože **pouze IEEE a STD knihovny jsou defaultně načítány**
- poté můžeme klíčovým slovem **use** s názvem balíčku **vybrat balíček z knihovny my\_library**
- **Knihovny (libraries)**
  - svazek (adresář) s možností uložit balíčky i celé VHDL moduly, komponenty atd.
  - zvláštní knihovna, tzv. **work** library – dočasná knihovna (vytvořená v rámci projektu), do které se automaticky ukládají všechny syntezované VHDL kódy v daném projektu
  - tato knihovna je ale **dostupná jen v daném projektu – jiný projekt = jiná knihovna work!**
  - pokud vytvoříme stálou knihovnu, musíme ji vždy při založení nového projektu nejprve načíst, teprve pak pomocí výrazu **use** z ní můžeme vybírat a používat jednotlivé části
    - pokud použijeme výraz **.all** pro daný balíček = celý obsah balíčku je dostupný
    - pokud použijeme jen **.název** jedné procedury, funkce atd. – jen tato procedura, funkce s tímto jménem je použitelná (ostatní se nenačtou)

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

```
package my_lib is  
procedure rl (a : std_logic_vector; b : natural; signal c : out std_logic_vector);  
procedure rr (a : std_logic_vector; b : natural; signal c : out std_logic_vector);  
end my_lib;
```

```
package body my_lib is  
procedure rl (a : std_logic_vector; b : natural; signal c : out std_logic_vector) is  
begin  
  if a'length<b then  
    report "Delka vektoru je mensi nez pozadovany pocet rotaci" severity failure;  
  else  
    c<=a(a'length-1-b downto 0)&a(a'length-1 downto a'length-b);  
  end if;  
end rl;
```

```
procedure rr (a : std_logic_vector; b : natural; signal c : out std_logic_vector) is  
begin  
  if a'length<b then  
    report "Delka vektoru je mensi nez pozadovany pocet rotaci" severity failure;  
  else  
    c<=a(b-1 downto 0)&a(a'length-1 downto b);  
  end if;  
end rr;  
end my_lib;
```



### • Knihovny a balíčky v nich

- vytvořili jsme předchozí **balíček my\_lib** obsahující 2 procedury – **rl** (rotate left) a **rr** (rotate right) pro rotování vektorů **std\_logic\_vector**
- obě procedury **rl** a **rr** mají 2 vstupní parametry – vstupní vektor **std\_logic\_vector** a **natural** = počet rotací, a 1 výstupní parametr – výstupní vektor **std\_logic\_vector**
- nejprve **musíme zkontrolovat jestli počet rotací nepřekračuje délku vektoru std\_logic\_vector** – v takovém případě jsme se rozhodli rotaci neprovádět a pomocí funkce **report** vypsat chybu a ukončit běh (šlo by ošetřit jinak, jak?)
- pokud je počet rotací menší **provedeme rotaci pomocí operátoru zřetězení – &**
- uložíme nyní balíček s názvem **my\_lib.vhd** a vytvoříme nový adresář, knihovnu – **my\_library** umístěnou v adresáři pro knihovny a balíček tam vložíme
- založíme nový projekt, v něm nový VHDL modul entity, např.: **rotate**
- označíme naši knihovnu **my\_library**
- klikneme OK a objeví se seznam knihoven dostupných pro daný projekt, můžeme tedy použít **my\_lib.vhd** pro náš projekt
- v menu **Libraries** klikneme pravým tlačítkem na **entity rotate** a vybereme naši knihovnu **my\_library**
- nyní můžeme deklarovat použití naší knihovny v daném modulu pomocí **library** a vybírat z ní balíčky pomocí klíčového slova **use**:

```
library my_library;  
use my_library.my_lib.all;
```

- **Knihovny a balíčky v nich**

- protože **dynamická knihovna work** automaticky obsahuje všechny vložené a aktivní knihovny, můžeme také vyvolat balíček my\_lib použitím knihovny **work**, **nemusíme tedy deklarovat použití knihovny my\_library** a stačí uvést:

```
use work.my_lib.all;
```

- nyní využijeme naší knihovnu k vytvoření jednoduchého příkladu – rotace 16bitového registru

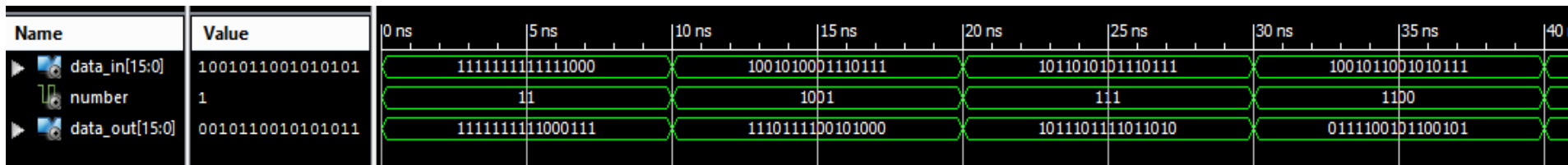
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
library my_library;
use my_library.my_lib.all;

entity rotate is
port(Data_in : in std_logic_vector(15 downto 0);
      number : in integer;
      Data_out : out std_logic_vector(15 downto 0));
end rotate;
architecture Behavioral of rotate is
begin
  rl(Data_in, number, Data_out);
end Behavioral;
```

# VHDL – Strukturování VHDL kódů, funkce, procedury, balíčky a knihovny

- **Knihovny a balíčky v nich**

- použijeme proceduru **rl** (rotate left) z našeho balíčku
- behaviorální simulace:



- nyní změníme deklaraci použití balíčku **my\_lib**:

```
use my_library.my_lib.rr;
```

- to znamená, že z balíčku **my\_lib** jsme načetli (vybrali) jen proceduru **rr**
- proto, když se budeme snažit nyní použít proceduru **rl**, která není načtená, skončí syntéza předchozího kódu chybou:  
**Undefined symbol 'rl'.**