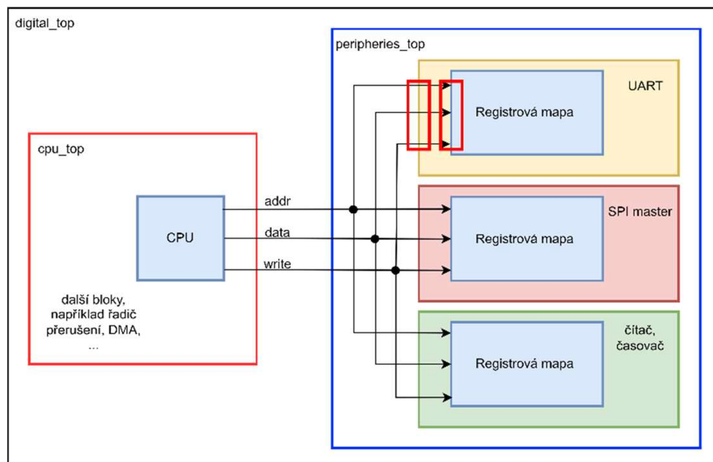
Výhody použití datového typu record v jazyce VHDL

Jakub Šťastný
jakub.stastny@asicentrum.com
ASICentrum spol. s r.o.

1 Úvod

Dobře dekomponovaný číslicový návrh má často hierarchickou strukturu, skládající se z řady menších do sebe různě vnořených bloků. Taková struktura je přirozená a velmi praktická. Proč? Prvotní specifikace každého produktu nahlíží na navrhovaný systém jako na jednu kompaktní entitu, jako na „černou krabičku“ (*black box*) a popisuje jeho funkce a vnější chování. Pohled je to užitečný pro počáteční definici cílů návrhu a snadné pochopení systému uživatelem. Není ale praktický pro realizaci, neboť je nemožné celý systém mentálně zvládnout jako „monolit“. Jak parametry produktu, tak návrhového projektu nelze odhadnout bez jeho rozdělení na menší části, nemluvě o distribuci úkolů mezi členy návrhového týmu. Proto během systémového návrhu dekomponujeme produkt (zde číslicový návrh) do bloků propojených vhodnými rozhraními. Důsledkem dekompozice je právě zmíněná hierarchická struktura navrhovaného systému s bloky propojenými množstvím rozhraní, která prochází napříč úrovněmi hierarchie.



Obrázek 1: Blokové schéma demonstračního systému.

Příklad takové struktury je návrh systému s mikropočítačem rozkreslený na obrázku 1. Ten obsahuje centrální procesorovou jednotku (CPU), instancovanou ve vlastní komponentě *cpu_top*; v ní jsou kromě procesoru všechny jeho podpůrné obvody (DMA, řadič přerušení a další). Dále vidíme řadu periferií procesoru sdružených v komponentě *peripheries_top*. Periferie jsou řízeny pomocí svých vlastních lokálních registrových map, jak je u mikropočítačů běžné. Aby běžící aplikace mohla periferie řídit, je každá z periferních registrových map připojena k procesoru (CPU) přes sběrnici, která umožňuje zapsat a číst do/z registrové mapy data instrukcemi v programu

vykonávaném mikroprocesorem. V obrázku je pro jednoduchost namalovaná jen zápisová část sběrnice, popis jejich signálů viz tabulka 1.

Signál	**Šířka**	**Účel**
addr	32	Adresa, na kterou chceme zapsat data, nebo ze které chceme číst.
data	32	Data k zápisu.
write	1	Zápisový signál, k zápisu do registrové mapy dojde s náběžnou hranou hodin. Když je *write* = ‘1’, zapíše se slovo ze sběrnice *data* na adresu přenášenou na sběrnici *addr*.

Tabulka 1: Popis rozhraní pro zápis dat do registrové mapy.

V typickém hierarchickém systému je velké množství rozhraní mezi bloky a to s sebou nese jednu drobnou nevýhodu. Pro lepší ilustraci jsme ve výpisu 1 zachytili ukázku VHDL kódu bloku *peripheries_top*. Výpis zachycuje jen ty části RTL kódu, které se přímo týkají propojení zápisové části registrové sběrnice mezi procesorem a periferiemi. Všimněte si, že v RTL kódu bloku *peripheries_top* jsou signály zápisové sběrnice zopakovány celkem sedmkrát, jednotlivé výskyty v textu jistě rychle identifikujete (sledujte výpis 1):

- jednou v entitě bloku *peripheries_top*,
- třikrát v komponentách jednotlivých periferiíí,
- a nakonec třikrát v jejich instancích.

Podíváme-li se dále do RTL kódů návrhu periferií (UART, SPI master, čítač a časovač – sledujte obrázek 1), zjistíme, že v každé z nich se definice sběrnice vyskytuje na rozhraní periferie, na rozhraní registrové mapy, v komponentě registrové mapy uvedené v architektuře periferie a v propojení (u bloku UART na obrázku 1 zvýrazněné červeně) – dohromady čtyři výskyty na jednu periferii, dvanáct na všechny tři periferie. Podobně na straně procesorové jednotky (opět viz obrázek 1, nyní levá část) je toto rozhraní přítomné na entitě CPU, entitě *cpu_top* bloku, na komponentě procesoru v architektuře bloku *cpu_top* a v propojení instance. Konečně, čtenář teď už asi dokáže přijmout bez potřeby větších důkazů, že na úrovni bloku *dig_top* najde ve zdrojových kódech dalších pět výskytů tohoto rozhraní (zkuste si je všechny uvědomit a nezapomeňte, že jedna z nich je definice propojení signálů sběrnice).

Co to pro nás znamená? Pokud budeme chtít rozšířit procesorové rozhraní o další signál (například *wait_s*, který indikuje, že sběrnice je právě pozastavená v čekacím stavu), musíme ručně změnit rozhraní na celkem 7+12+4+5 = 28 místech v kódu, konkrétně

- v jednom místě naimplementovat řízení nového signálu – „u zdroje“, což zde bude v procesorové jednotce (blok CPU na obrázku 1). To je kreativní návrhářská práce – je potřeba naimplementovat řídící blok.
- na třech místech naimplementovat správné zpracování nového signálu – v registrových mapách periferií (bloky Registrová mapa na obrázku 1). Zde děláme také kreativní práci, musíme signál správně použít v kontextu všech registrových map periferií.
- na 24 místech zajistit správný průchod nového signálu přes rozhraní bloků. To je více-méně repetitivní a nudná záležitost, kdy opravujeme jedno rozhraní po druhém, dokud nemáme pocit, že máme vše „čisté“.

Uvědomme si nyní, že přidat signál na 24 různých míst v kódu – všude stejný – a neudělat při tom chybu není tak jednoduché, jak to vypadá – to i přes to, že návrháři zde může pomoct moderní editor HDL kódu s refaktorizační funkcí „Add New Port to Module“. Nejsnažší chybou je přitom opomenout aktualizaci některého z rozhraní či celé periferie a takovou chybu není jednoduché odhalit pomocí verifikace. Návrhář navíc může mít pocit, že si takovou práci nezaslouží a možná by to mohl dělat nějak lépe. A tento pocit je opodstatněný.

```
ENTITY peripheries_top IS
  PORT (
       … signály bloku…
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
  );
END ENTITY
ARCHITECTURE rtl OF peripheries_top is

  COMPONENT uart IS
    PORT (
       … signály bloku…
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
    )
  END COMPONENT uart;

  COMPONENT spi_master IS
    PORT (
       … signály bloku…
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
    )
  END COMPONENT spi_master;

  COMPONENT counter_timer IS
    PORT (
       … signály bloku…
       addr : IN std_logic_vector (31 DOWNTO 0);
       data : IN std_logic_vector (31 DOWNTO 0);
       write: IN std_logic;
       …
    )
  END COMPONENT counter_timer;

BEGIN
  i_uart : uart
  PORT MAP (
```

```
    … další signály podle potřeby …
    addr  => addr,
    data  => data,
    write => write,
    …
  );

  i_spi_master : spi_master
  PORT MAP (
    … další signály podle potřeby …
    addr  => addr,
    data  => data,
    write => write,
    …
  );

  i_counter_timer : counter_timer
  PORT MAP (
    … další signály podle potřeby …
    addr  => addr,
    data  => data,
    write => write,
    …
  );
END ARCHITECTURE rtl;
```

Výpis 1: Vybrané části zdrojového kódu struktury systému z obrázku 2.

2 Co je datový typ record

Než se podíváme na řešení problému nešťastného návrháře, řekněme si, co je datový typ record. Datový typ *record* (či jeho ekvivalenty, např. *struct* v jazyce C, a SystemVerilogu) je kompozitní datový typ dostupný prakticky ve všech moderních programovacích jazycích. Jeho základním účelem je sdružovat datové položky, které spolu významově souvisí (více viz např. [1, strana 36], [2]) a přitom každá položka může být jiného datového typu. Ve VHDL lze jednoduše deklarovat signály typu record, proměnné či konstanty typu record, pole s prvky typu record a recordy do sebe vnořovat, příklady viz výpis 2. Signály typu record můžeme také používat na vstupně/výstupním rozhraní bloků, což pro nás bude dále zajímavé.

```
TYPE T_REG_WRITE IS RECORD
  addr  : std_logic_vector (31 DOWNTO 0);
  data  : std_logic_vector (31 DOWNTO 0);
  write : std_logic;
END RECORD T_REG_WRITE;

CONSTANT C_REG_WR_RESET : T_REG_WRITE := (
   addr  => (OTHERS => '0'),
   data  => (OTHERS => '0'),
   write => '0'
 );


TYPE T_WRITE_ARR IS ARRAY (0 TO 3) OF T_REG_WRITE;

SIGNAL t_wr_bus : T_REG_WRITE;
```

Výpis 2: Příklady zápisu deklarace typu record, signálu, konstanty a typu pole.

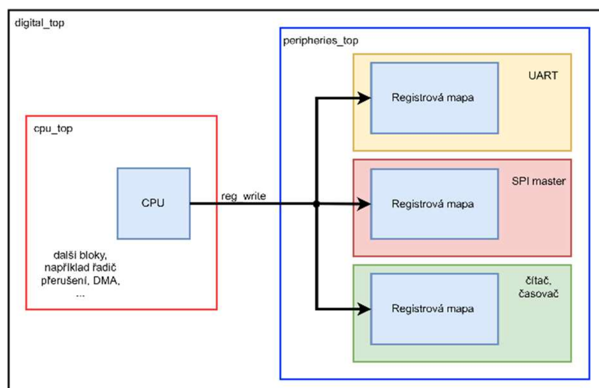
3 Rozhraní bloku a typ record

Vraťme se nyní k tomu, jak návrháři ulehčit psaní jeho RTL. Ke zjednodušení situace a zlepšení nálady návrháře by přitom stačilo málo: mít možnost definovat stejnou a opakující se část rozhraní, zapouzdřit ji a opětovně použít.

Čtenář již jistě tuší, že nám k tomu pomůže signál typu record. Podívejme se rovnou jak: s jeho použitím návrh přepracujeme do podoby znázorněné na obrázku 2. Ekvivalentní RTL kód je potom uveden ve výpisu 3, srovnejte jeho délku s kódem v předchozím výpise 2. Analyzujme kolik práce nyní přinese přidání signálu *wait_s* do zápisového rozhraní. Návrhář místo změny na 28 místech teď

- v jednom místě upraví deklaraci typu *record* tak, že do ní přidá nový signál. Přestože je v kódu stále 7 míst, kde se zápisová sběrnice vyskytuje, deklarace je teď jen jedna – umístěná v našem případě v package v souboru *reg_bus_pkg.vhd* (umístit definici typu record do VHDL package je užitečné proto, aby se dal sdílet mezi entitami). Přidání signálu *wait_s* tedy teď znamená modifikaci jen jediného místa v kódu.
- v jednom místě naimplementuje řízení nového signálu – „u zdroje“, zde v procesorové jednotce (blok CPU v obrázku 2),
- na třech místech naimplementuje správné zpracování nového signálu – v registrových mapách periferií (modré bloky „Registrová mapa“ v obrázku 2). V tomto i předchozích dvou případech – jak už víme – návrhář dělá kreativní činnost.

A to je vše. Před tím bylo potřeba na 24 místech zajistit správný průchod nového signálu přes rozhraní bloků. To je nyní vyřešeno automaticky změnou deklarace datového typu v prvním kroku; změna se všude rozšíří automaticky, neboť kompilátor jednoduše použije novou definici datového typu.



Obrázek 2: Blokové schéma demonstračního systému; místo jednotlivých signálů je použito zapouzdření do datového typu record.

```
reg_bus_pkg.vhd:

TYPE T_REG_WRITE IS RECORD
  addr  : std_logic_vector (31 DOWNTO 0);
  data  : std_logic_vector (31 DOWNTO 0);
  write : std_logic;
  wait_s: std_logic;
END RECORD T_REG_WRITE;

peripheries_top.vhd:

ENTITY peripheries_top IS
```

```
  PORT (
       … signály bloku…
       reg_write : IN T_REG_WRITE;
       …
  );
END ENTITY
ARCHITECTURE rtl OF peripheries_top is

  COMPONENT uart IS
    PORT (
       … signály bloku…
       reg_write : IN T_REG_WRITE;
       …
    )
  END COMPONENT uart;

  COMPONENT spi_master IS
    PORT (
       … signály bloku…
       reg_write : IN T_REG_WRITE;
       …
    )
  END COMPONENT spi_master;

  COMPONENT counter_timer IS
    PORT (
       … signály bloku…
       reg_write : IN T_REG_WRITE;
       …
    )
  END COMPONENT counter_timer;

BEGIN
  i_uart : uart
  PORT MAP (
    … další signály podle potřeby …
    reg_write => reg_write,
    …
  );

  i_spi_master : spi_master
  PORT MAP (
    … další signály podle potřeby …
    reg_write => reg_write,
    …
  );

  i_counter_timer : counter_timer
  PORT MAP (
    … další signály podle potřeby …
    reg_write => reg_write,
    …
  );
END ARCHITECTURE rtl;
```

Výpis 3: Upravený kód entitity peripheries_top s použitím signálů typu record.

4 Implementace registru

Použití signálu typu *record* umožňuje zjednodušit návrháři práci i na řadě dalších míst. Praktické je při implementaci registru: pokud bychom potřebovali původní zápisovou sběrnici (jako ve výpisu 1) zavést do registru aktivního na náběžnou hranu hodin (flip-flop), naivní přístup by vedl na VHDL kód ve výpisu 4.

```
SIGNAL addr_int : std_logic_vector (31 DOWNTO 0);
SIGNAL data_int : std_logic_vector (31 DOWNTO 0);
SIGNAL write_int: std_logic;

output_reg:PROCESS (res_n, clk)
BEGIN
  IF res_n = '0' THEN
```

```
    addr  <= (OTHERS => '0');
    data  <= (OTHERS => '0');
    write <= '0';
  ELSIF rising_edge(clk) THEN
    addr  <= addr_int;
    data  <= data_int;
    write <= write_int;
  END IF;
END PROCESS output_reg;
```

Výpis 4: Registr na zápisové sběrnici, původní implementace.

Po transformaci sběrnice – refaktorování ze tří oddělených signálů na jeden signál typu record – můžeme kód snadno zjednodušit, podívejte se na výpis 5.

```
SIGNAL reg_wr_int : T_REG_WRITE;

output_reg_2: PROCESS (res_n, clk)
BEGIN
  IF res_n = '0' THEN --(#)
    reg_wr.addr  <= (OTHERS => '0');
    reg_wr.data  <= (OTHERS => '0');
    reg_wr.write <= '0';
  ELSIF rising_edge(clk) THEN
    reg_wr <= reg_wr_int; --(*)
  END IF;
END PROCESS output_reg_2;
```

Výpis 5: První verze RTL popisu registru na zápisové sběrnici implementované jako signál typu record.

To už je přínosné, protože při změně deklarace typu record se automaticky na řádku **(*)** naimplementují registry podle aktuální deklarace. Pořád ale budeme proces muset ručně modifikovat. Proč? Prohlédněte si část zodpovědnou za asynchronní reset označenou (#): pokud přidáme například do recordu signál *wait_s*, budeme muset připsat do kódu procesu pod řádek (#) tento kód

```
reg_wr.wait_s <= '0';
```

aby došlo ke správné inicializaci signálu asynchronním resetem do log. 0. Pokud na inicializaci zapomeneme, syntézní nástroj zde bude signál *wait_s* řídit z registru bez asynchronního resetu a to může způsobit komplikace ve finálním návrhu. Pomoct by nám pak zde mohla už jen pečlivá verifikace – testy mohou (ale nemusí!) zachytit důsledek toho, že je signál ve stavu 'U' po startu RTL simulace, nebo že není resetován spolu s ostatními signály. Od problému si elegantně odpomůžeme deklarací konstanty typu record, která bude určovat resetovací hodnoty (neaktivní stav) na výstupu registru, celé řešení je ve výpisu 6.

```
CONSTANT C_REG_WR_RES : T_REG_WRITE := (
  addr => (OTHERS => '0'),
  data => (OTHERS => '0'),
  write => '0'
);

SIGNAL reg_wr : T_REG_WRITE;

output_reg_3:PROCESS (res_n, clk)
BEGIN
  IF res_n = '0' THEN
    reg_wr <= C_REG_WR_RES;
```

```
  ELSIF rising_edge(clk) THEN
    reg_wr <= reg_wr_int;
  END IF;
END PROCESS output_reg_3;
```

Výpis 6: Použití konstanty pro definování resetovací hodnoty výstupu registru.

Nyní jen stačí s každou změnou typu record změnit příslušnou konstantu určující neaktivní stav a registr se „aktualizuje automaticky“ při příští kompilaci či syntéze. Krása tohoto přístupu spočívá také v tom, že nás kompilátor začne hlídat: pokud návrhář změní deklaraci typu a neopraví konstantu, kompilace skončí s chybou a návrh bude muset být opraven. Například VHDL kompilátor Vivado ohlásí

```
ERROR: [VRFC 10-3717] some record elements are missing in this aggregate of 'ṭ_reg_write' [record_logic_demo.vhd:49]
```

Jsme tedy automaticky upozornění na všechna místa kde je potřeba provést změnu, což významně snižuje riziko zanesení chyby při změně návrhu.

5 Kombinační funkce

Jistě teď nebude pro čtenáře překvapením, že podobně „přátelsky“ se chovají i jiné jazykové konstrukce, které jinak používáme s jednoduchými signály. Signály typu record například můžeme jednoduše multiplexovat, viz výpis 7, proces *comb_func*. A analogicky lze implementovat například „hradlování“ signálu do neaktivních hodnot s pomocí předem deklarované konstanty, také viz výpis 7, řádek označený (#). Konečně, jednoduchou konstrukcí naznačenou v procesu *overwrite_write* lze změnit hodnotu jen jediné položky v signálu typu record.

```
SIGNAL reg_wr_0 : T_REG_WRITE;
SIGNAL reg_wr_1 : T_REG_WRITE;
SIGNAL sel : std_logic;

SIGNAL write_2  : std_logic;
SIGNAL reg_wr_2 : T_REG_WRITE


comb_func: PROCESS (sel, reg_rw_0, reg_rw_1)
BEGIN
  IF sel = '1' THEN
    out_wr <= reg_rw_0;
  ELSE
    out_wr <= reg_rw_1;
  END IF;
END PROCESS comb_func;

out_wr <= reg_rw_0 WHEN disable = '0' ELSE C_REG_WR_RES; --(#)

overwrite_write : PROCESS (reg_wr_0, write_2)
BEGIN
  reg_wr_2       <= reg_wr_0;
  reg_wr_2.write <= write_2;
END PROCESS overwrite_write;
```

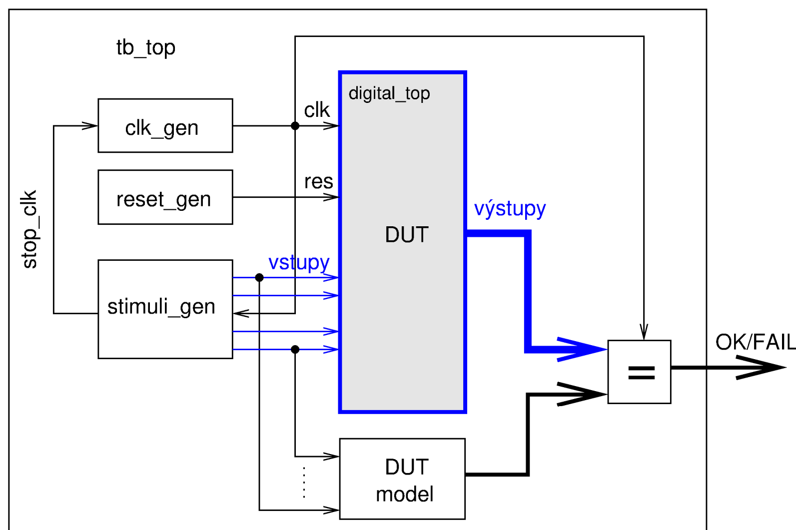
Výpis 7: Další možnosti použití práce se signály typu record.

6 Na co si dát pozor

S výhodami používání datového typu record byl čtenář důkladně obeznámen v předchozím textu. Zdá se, že návrhář zde jen mnoho získává a nic neztrácí. Můžeme se ovšem ptát, zda má používání "recordů" také nějaké stinné stránky?

Za prvé, občas se u některých jazykových rysů jazyka VHDL (zejména z novějších verzí standardu) můžeme setkat s chybějící podporou v simulátorech či nástrojích pro syntézu. Dobrá zpráva je, že datový typ record je bez omezení podporován všemi nám známými a rutinně používanými nástroji. Zde tedy žádnou nevýhodu nevidíme.

Za druhé, signály typu record je lepší nepoužívat na rozhraní nejvyšší úrovně hierarchie návrhu – viz obrázek 3, blok *DUT* (Design Under Test), všechna modře označená rozhraní a signály. Pokud zde signál typu record použijete, bude potom potřeba udržovat dvě verze instance entity *DUT* v entitě *tb_top* (*testbench top*). Problém zde přináší potřeba provádět simulaci a verifikaci návrhu jak na RTL (první verze instance v *tb_top*), tak na hradlové úrovni (druhá verze instance v *tb_top*) kdy na místě *DUT* bloku instancujeme netlist návrhu anotovaný SDF souborem (více viz [3], také odkaz na stránce [4]). Netlist vzniká z RTL návrhu syntézou, rozmístěním a propojením a zachycuje skutečné fyzické schéma návrhu, ve kterém je signál typu record použitý na RTL úrovni nahrazen svým schematickým (fyzickým) ekvivalentem – tzn. každá položka typu record jedním samostatným signálem. Uvnitř bloku to ničemu nevadí, ale pokud je signál typu record použit na rozhraní *DUT* bloku, rozhraní netlistu se bude lišit od rozhraní RTL verze bloku *DUT* a to může způsobit (poznamenejme, že řešitelné) komplikace v místě, kde instancujeme blok *DUT* v bloku *tb_top*. Vše lze ovšem řešit například pomocí VHDL konstrukce ve výpisu 8.



Obrázek 3: Příklad struktury verifikačního prostředí pro číslicový systém. Modře jsou označeny signály a rozhraní, kde není doporučenou používat signály typu record.

Za třetí, malé komplikace přináší také použití signálu typu record spolu s konstrukcí tzv. hierarchických jmen [5, 6] (někdy také tzv. mirrorů, *SignalSpy* v simulátorech Siemens či *nc_mirror* v simulátorech Cadence). Tyto jazykové konstrukce umožňují testům a modelům ve verifikačním prostředí „nahlížet" do vnitřní implementace bloku. V mnoha případech to při verifikaci potřebujeme, například když některý interní signál v bloku nese důležitou informaci o stavu návrhu. I zde je problém v odlišné struktuře sběrnice na RTL a hradlové úrovni – opět bude potřeba udržovat dvě verze kódu, jednu pro RTL a druhou pro simulace na hradlové úrovni. Na druhou stranu je i zde problém snadno řešitelný například pomocí obměny konstrukce IF-GENERATE generující dvě skupiny mapování jako ve výpisu 8.

Za čtvrté, do jednoho signálu typu record sdružujte nejlépe signály, které spolu nějak významově souvisí – v našem příkladu to jsou signály, které dohromady tvoří zápisovou část periferní sběrnice procesoru. Vytvářet „superrekord", který obsahuje množství vzájemně nesouvisejících signálů, kterým je společné jen to, že se vyskytují spolu na rozhraní entity, by kód asi spíše znepřehlednilo.

```
ENTITY dut IS
  GENERIC (
    G_GATE_SIMS : natural := 0; -- 0 - RTL, 1 - hradlová úroveň
  )
  PORT ( … )
END ENTITY dut;

ARCHITECTURE rtl OF dut IS
…
BEGIN

-- propojení pro RTL úroveň
gen_rtl : IF G_GATE_SIMS = 0 GENERATE
  i_dut : dut_rtl
    PORT MAP (
…
    )
END GENERATE gen_rtl;

-- propojení pro hradlovou úroveň
gen_gate : IF G_GATE_SIMS = 1 GENERATE
  i_dut : dut_gate
    PORT MAP (
…
    )
END GENERATE gen_gate;

END ARCHITECTURE rtl;
```

Výpis 8: VHDL konstrukce IF - GENERATE.

7 Závěr

Článek shrnul přínosy i potenciální negativa použití datového typu record v HDL jazyce. Návrhář může použitím signálů typu record ušetřit mnoho nudné a jednotvárné práce, vytvořit kvalitnější a čitelnější kód a zabránit vzniku mnoha nepříjemných chyb. Současně použití signálů typu record nepřináší žádnou dodatečnou režii na úrovni implementace číslicového systému a je bez omezení podporováno ve všech nástrojích známých autorovi. Použití datového typu record v syntetizovatelném RTL kódu lze tedy jen doporučit.

8 Poděkování

Je mou milou povinností poděkovat mé drahé ženě Julince, a kolegům Robertu Kvačkovi, Tomáši Novákovi a Luboši Hradeckému za jazykovou, stylistickou i obsahovou revizi a jejich skvělé komentáře.

Použité zkratky

ASIC	Application Specific Integrated Circuit
DUT	Design Under Test
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IP	Intelectual Property
RTL	Register Transfer Level

tb_top TestBench TOP

Odkazy k dalšímu studiu

[1] PINKER Jiří, POUPA Martin. *Číslicové systémy a jazyk VHD*L. BEN Praha 2006

[2] NAND Land [online]. Records – VHDL Example [cit 25.3.2024]. Dostupné z https://nandland.com/record/

[3] ŠŤASTNÝ, Jakub. Simulace číslicových obvodů na hradlové úrovni. *DPS Elektronika od A do Z,* květen/červen 2015, s 8-11.

[4] ŠŤASTNÝ, Jakub, 202. Minimized Logic. [online; cit 25.3.2024]. Dostupné z www.minimizedlogic.com

[5] DOULOS [online]. VHDL-2008: Easier to use, Hierarchical names [cit 25.3.2024]. Dostupné z

https://www.doulos.com/knowhow/vhdl/vhdl-2008-easier-to-use/#hierarchicalnames

[6] CHIPVERIFY [online]. Verilog Hierarchical Reference Scope [cit 25.3.2024]. Dostupné z

https://www.chipverify.com/verilog/verilog-hierarchical-reference-scope