1 Předmluva

Tento text je určen pro čtenáře, který se chce seznámit s jazykem Verilog a s jeho novější zdokonalenou a rozšířenou verzí – či správnější by snad bylo říci nadstavbou – s jazykem SystemVerilog, a s jejich použitím při návrhu aplikací programovatelných obvodů známých jako **obvody PLD** (*Programmable Logic Devices*) a **FPGA** (*Field Programmable Gate Arrays*), které budeme dále souhrnně (i když ne zcela přesně) označovat vžitým názvem programovatelné obvody. Měl by čtenáře připravit pro praktickou aplikaci těchto obvodů v číslicových konstrukcích. Důraz je zde proto kladen především na syntézu; simulace je probrána v rozsahu, který je potřebný pro ověření funkce takových aplikací. Do značné míry je text použitelný i pro čtenáře, který se zaměřuje na obvody ASIC (*Application Specific Integrated Circuits*), i když tam jsou určité odlišnosti, má-li být výsledek implementace optimální. Aby byl jeho rozsah udržen v přijatelných mezích, není v něm cílem úplnost výkladu syntaxe. Není v něm zahrnuta například většina nesyntetizovatelných částí jazyka nebo úplný výčet různých variant hodnot operandů v popisu funkce operátorů (zejména reálných nebo nedefinovaných hodnot), nebo také podrobnosti o příkazech skupiny `generate`. Nejsou zde také podrobně uvedeny informace o organizaci výpisů na konzolu při simulaci, které odpovídají příslušným výpisům při práci s jazykem C (systémová úloha `$display`, řídící řetězce formátu pro výpisy apod.). V některých případech se pak spoléhá na intuici čtenáře. Zájemce o podrobnější a úplnější zpracování najde informace ve speciální literatuře, především v referenčních manuálech jazyků Verilog [1], popř. [2] a SystemVerilog [6]; někdy může být užitečné vědět i o starších verzích standardu [4], [5]. Standard pro syntézu modelů zapsaných ve Verilogu je předmětem publikace [3]. Referenční manuál jazyka SystemVerilog [6] verze 2012 lze bezplatně stáhnout z www stránky http://standards.ieee.org/getieee/1800/download/1800-2012.pdf. V něm není rozlišeno použití jazyka SystemVerilog k verifikaci a k syntéze, tj. není vyznačena syntetizovatelná část jazyka, což lze pokládat za určitý nedostatek. Použití SystemVerilogu k syntéze je podrobně zpracováno v knize [11], jejíž autor patří k duchovním otcům a předním popularizátorům tohoto jazyka, která však u čtenáře předpokládá znalost jazyka Verilog. Poznatky i příklady v ní uvedené vhodně doplňují referenční manuály a zde se na ni budeme často odkazovat. V některých případech se však vyskytují drobné odlišnosti mezi touto knihou a referenčními manuály, a na ně v našem textu upozorníme, protože je možné, že verze uvedená v knize bude lépe odpovídat současným syntetizérům. O standardech pro syntézu se ještě zmíníme v **odst. 2.1**. Použití jazyka SystemVerilog k verifikaci je podobně zpracováno například v knize [14].

U čtenáře se předpokládá znalost číslicové techniky v rozsahu odpovídajícím základním kurzům technických vysokých nebo i středních škol, shrnutém například v publikaci [8] nebo [9]. Bude také užitečná (i když ne nezbytná) aspoň orientační znalost obvodových struktur a funkčních bloků používaných v obvodech PLD a FPGA. Dále se předpokládá (i když to není bezpodmínečně nutné), že čtenář dokáže pracovat s některým návrhovým systémem, v němž jsou jazyky Verilog a SystemVerilog podporovány, například se systémem Quartus II firmy Altera; další známá firma Xilinx má oba jazyky implementovány v systému Vivado, starší systém ISE podporuje jen jazyk Verilog. Základní verzi systému Quartus – tzv. Web Edition – může zájemce bezplatně stáhnout z www stránek firmy Altera, a podobné je to i u systémů firmy Xilinx. Výhodné, i když nikoli nutné, pro něj bude, když bude mít zkušenosti s některým u nás rozšířeným jazykem HDL (*Hardware Description Language*), například s jazykem VHDL – pro seznámení s tímto jazykem v češtině lze doporučit například knihu [8]. V angličtině jsou jazyky Verilog i VHDL i s ohledem na syntézu zpracovány přístupnou formou například v publikaci [7].

Jazyky HDL jsou dnes nejpoužívanějším a většinou nejefektivnějším prostředkem pro modelování vyvíjených číslicových systémů a pro jejich následnou syntézu a implementaci do programovatelných obvodů i do obvodů ASIC. Nenajdeme dnes prakticky používaný návrhový systém, který by tyto jazyky, především VHDL a Verilog, nepodporoval. U nás se dosud především z historických důvodů pracovalo převážně s jazykem VHDL, který byl nejrozšířenějším jazykem HDL v evropských zemích, zatímco jazyk Verilog dříve dominoval v asijských zemích, a v zemích amerického kontinentu byly zhruba stejně rozšířeny oba tyto jazyky. Tomu také odpovídá výuka v českých školách i literatura psaná v češtině, jako např. [8]. Ve světě se ale stále více prosazuje jazyk Verilog, který je jednodušší na pochopení a stručnější ve vyjadřování, poskytuje však prakticky stejné možnosti jako jazyk VHDL; zhruba od začátku tohoto desetiletí přebírá vedoucí úlohu jazyk SystemVerilog, který je nadstavbou nad Verilogem. V praxi se proto konstruktéři číslicových systémů často setkají se zdrojovými texty napsanými ve Verilogu nebo v SystemVerilogu, které získají například z internetu

KLÍČOVÁ SLOVA jazyka nově zavedená ve verzi *SystemVerilog-2005* s doplněním o klíčová slova verze *SystemVerilog-2009* (kurziva), [4], [11]:

- ***accept_on***
- **alias**
- **always_comb**
- **always_ff**
- **always_latch**
- **assert**
- **assume**
- **before**
- **bind**
- **bins**
- **binsof**
- **bit**
- **break**
- **byte**
- **chandle**
- ***checker***
- **class**
- **clocking**
- **const**
- **constraint**
- **context**
- **continue**
- **cover**
- **covergroup**
- **coverpoint**
- **cross**
- **dist**
- **do**
- ***endchecker***
- **endclass**
- **endclocking**
- **endgroup**
- **endinterface**
- **endpackage**
- **endprogram**
- **endproperty**
- **endsequence**
- **enum**
- ***eventually***
- **expect**
- **export**
- **extends**
- **extern**
- **final**
- **first_match**
- **foreach**
- **forkjoin**
- ***global***
- **iff**
- **ignore_bins**
- **illegal_bins**
- ***implies***
- **import**
- **inside**
- **int**
- **interface**
- **intersect**
- **join_any**
- **join_none**
- ***let***
- **local**
- **logic**
- **longint**
- **matches**
- **modport**
- **new**
- ***nexttime***
- **null**
- **package**
- **packed**
- **priority**
- **program**
- **property**
- **protected**
- **pure**
- **rand**
- **randc**
- **randcase**
- **randsequence**
- **ref**
- ***reject_on***
- ***restrict***
- **return**
- ***s_always***
- ***s_eventually***
- ***s_nexttime***
- ***s_until***
- ***s_until_with***
- **sequence**
- **shortint**
- **shortreal**
- **solve**
- **static**
- **string**
- ***strong***
- **struct**
- **super**
- ***sync_accept_on***
- ***sync_reject_on***
- **tagged**
- **this**
- **throughout**
- **timeprecision**
- **timeunit**
- **type**
- **typedef**
- **union**
- **unique**
- ***unique0***
- ***until***
- ***until_with***
- ***untyped***
- **var**
- **virtual**
- **void**
- **wait_order**
- ***weak***
- **wildcard**
- **with**
- **within**

Poznámka: Klíčová slova všech verzí jazyka Verilog zůstávají klíčovými slovy jazyka SystemVerilog. Direktivami `` `begin_keywords `` a `` `end_keywords `` lze v SystemVerilogu nastavit platnou sadu klíčových slov, viz **odst. 2.1.2**.

KLÍČOVÁ SLOVA jazyka nově zavedená ve verzi *SystemVerilog-2012* [6]:

- **implements**
- **interconnect**
- **nettype**
- **soft**

kterou jsou schopny zpracovat navazující programové nástroje (simulátor, syntetizér) a která zpravidla není uživateli přístupná. Přitom se zpravidla provádí kontrola syntaktické správnosti (*parsing*) textu. **Elaborace** (*elaboration*) navazuje na analýzu: zpracovává se při ní hierarchická struktura projektu – vytvářejí se vazby mezi dílčími bloky konstrukce (komponentami) a jejich vloženími do vyšších jednotek (instancemi), vyčíslují se hodnoty parametrů a podobně. Výsledkem je tzv. netlist, který je pak podroben simulaci nebo syntéze. Některé nástroje obě tyto fáze slučují do jednoho procesu, u jiných jsou fáze více či méně odděleny, někdy se také názvy fází posouvají nebo zaměňují.

V dalších kapitolách budeme často používat pojem **signál**. Budeme tím zpravidla rozumět veličinu, jejíž hodnota se může v čase měnit, na rozdíl od konstant, parametrů a podobně. Signál může být skalární (jednobitový) nebo vektorový (vícebitový), o signálech však budeme mluvit i v souvislosti se složitějšími datovými objekty, jako jsou například pole nebo struktury.

Poznámka k označení `V,SV`: Příklady v dalším textu budou většinou psány v syntaxi SystemVerilogu, tedy v deklaracích budou uváděny proměnné typu `logic` místo typu `reg` z Verilogu a příkazy SystemVerilogu `always_comb`, `always_latch` a `always_ff` budou používány místo příkazů `always` (Verilog). Rovněž budou psány v syntaxi SystemVerilogu příkazy pro smyčky `for` a v definicích funkcí nebudou uváděny nadbytečné příkazové závorky `begin-end`. Tam, kde to bude účelné, budou příkazy `case` doplněny příslušným modifikátorem (`unique`, popř. `priority`). Pokud budou tyto texty po jednoduchém návratu k syntaxi Verilogu, tedy po náhradě typu `logic` typem `reg` (typem `wire` v deklaracích vstupních bran a u signálů, jimž se přiřazuje hodnota v příkazech `assign`, a také u signálů připojených k výstupům vložené jednotky ve strukturálních popisech), po náhradě tří dalších příkazů příkazem `always` (u prvních dvou z nich je třeba doplnit ještě citlivostní seznam), po úpravě syntaxe smyček (viz **odst. 2.6.6**) a funkcí a po vynechání modifikátorů odpovídat syntaxi Verilogu, budou označeny poznámkou `V*,SV`; bude-li stačit pouhá změna typů nebo nebude-li zapotřebí žádná úprava, nebude v označení hvězdička. Bude-li však syntaxe v SystemVerilogu od syntaxe ve Verilogu natolik odlišná, že by převod do syntaxe Verilogu byl příliš komplikovaný, popřípadě má-li text sloužit k ilustraci konstruktů, které nejsou platné ve Verilogu (například při použití uživatelských typů, struktur, rozhraní nebo pokročilých konstruktů pro zpracování polí), bude u nich poznámka `jen SV` (pokud platnost nebude zřejmá z okolního textu). Případné další rozdíly mezi oběma druhy syntaxe budou uvedeny individuálně v okolním textu nebo formou komentáře. Některé texty budou psány v syntaxi Verilogu-2001, a ty budou označeny poznámkou `V` (ty budou samozřejmě vyhovovat i syntaxi SystemVerilogu). Autor věří, že převod textů s poznámkou `V*,SV` do syntaxe Verilogu nebude čtenáři činit velké obtíže.

2.1.1 Způsoby sestavení modelu a styly popisu

Rozeznáváme tři základní **způsoby sestavení modelu:**

Postup **zdola nahoru** (*bottom-up*): nejprve se vytvoří dílčí bloky modelu (moduly, případně další konstrukční prvky, viz **odst. 2.2**) a ty se pak skládají do větších celků. Tento postup byl charakteristický pro sestavování číslicových konstrukcí z pevně zapojených číslicových obvodů, které představovaly bloky nejnižší úrovně, nebo pro vytváření popisu modelů pomocí schémat propojováním schematických značek základních bloků, což bylo typické pro počátky využívání obvodů FPGA.

Postup **shora dolů** (*top-down*): definuje se funkce navrhovaného systému jako celku. V něm se vyčlení bloky, jejichž funkce se specifikuje spolu s vzájemnou návazností jednotlivých bloků. U velkých konstrukcí mohou jednotlivé bloky zpracovávat různí konstruktéři. Tento proces pokračuje tak dlouho, až se na nejnižší úrovni získají dostatečně jednoduché bloky, jejichž funkci je možno popsat prostředky, jako je behaviorální popis v jazycích HDL na úrovni RTL (viz dále). Tak se zpravidla pracuje při syntéze s použitím moderních systémů CAD, kde starost o podrobné zapojení na ještě nižších úrovních přebírají návrhové systémy.

Přístupy zdola nahoru a shora dolů může být účelné v různých fázích zpracování modelu určité konstrukce kombinovat. Tyto způsoby sestavení modelu se označují jako **hierarchické**. Rozumně zvolená hierarchie usnadňuje orientaci v popisu modelu navrhované konstrukce a dovoluje opětné použití odladěných dílčích bloků. Jazyk Verilog podporuje hierarchičnost modelů, v SystemVerilogu je podpora hierarchie ještě značně zdokonalena – je zde zavedeno mnoho nových konstruktů, které usnadňují práci na rozsáhlých hierarchicky uspořádaných projektech.

Pro ilustraci uvedeme několik příkladů deklarací bran v SystemVerilogu:

`module m1 ([3:0] x);`	`module m1 (inout wire logic [3:0] x);`
`module m2 (integer x);`	`module m2 (inout wire integer x);`
`module m3 (input x);`	`module m3 (input wire logic x);`
`module m4 (input var x);`	`module m4 (input var logic x);`
`module m5 (output x);`	`module m5 (output wire logic x);`
`module m6 (output var x);`	`module m6 (output var logic x);`
`module m7 (output integer x);`	`module m7 (output var integer x);`
`module m8 (output signed x);`	`module m8 (output wire logic signed x);`
`module m9 (wire x, y[3:0]);`	`module m9 (inout wire logic x,`
	`inout wire logic y[3:0]);`
`module m10(integer x, signed y);`	`module m10(inout wire integer x,`
	`inout wire logic signed y);`
`module m11(output var x,`	`module m11(output var logic x,`
`input y);`	`input wire logic y);`
`module m12(output logic x, y);`	`module m12(output var logic x,`
	`output var logic y);`
`module m13(output signed [3:0] x,`	`module m13`
`integer y);`	`(output wire logic signed [3:0] x,`
	`output var integer y);`

V levém sloupci jsou zde uvedeny deklarace, kde jsou některé atributy vynechány, v pravém sloupci jsou doplněné deklarace stanovené podle pravidel o doplňování atributů bran v SystemVerilogu. Příručka [6] uvádí další příklady tohoto doplňování. Výslovně se v ní dále uvádí: Není-li v deklaraci výstupní brány uvedeno, jde-li o propojení nebo o proměnnou, je-li však v ní deklarován explicitně datový typ, inferuje se proměnná (jde zřejmě o dodržení kompatibility s Verilogem, kde se často deklaruje výstupní proměnná typu `reg`, čemuž v SystemVerilogu odpovídá proměnná s datovým typem `logic`). Odtud vyplývá doplnění seznamů v modulech `m7`, `m12` a `m13` (proměnná `y`). V modulech `m5` a `m8` není explicitně deklarován datový typ, takže je inferováno propojení; totéž platí pro signál `x` v modulu `m13`.

Z příkladu deklarace modulu `m1` také vyplývá, že k tomu, aby byla deklarace považována za deklaraci ve stylu ANSI, stačí u první brány v seznamu uvést její rozsah, i když to v manuálu [6] není výslovně uvedeno (je tam ale uveden velmi podobný příklad).

SystemVerilog dále dovoluje specifikovat výchozí (*default*) hodnoty signálů vstupních bran – podrobněji o tom bude řeč v **odst. 2.8.7**.

V průběhu vývoje SystemVerilogu došlo ke změně. Ve standardu *IEEE 1800-2005* se signály vstupních bran deklarovaných dvojicí klíčových slov `input logic` (bez bližší specifikace dalšími klíčovými slovy) považovaly za proměnné. Standard *IEEE 1800-2009* zde zavedl změnu: signály těchto vstupních bran mají datový typ `wire` (také čtyřhodnotový), tedy odpovídají úplné deklaraci `input wire logic` (netýká se to bran deklarovaných jako výstupní, tedy `output`). Takové signály vstupních bran mohou být na rozdíl od proměnných buzeny z více budičů, a je také přípustné přiřadit jim hodnotu kontinuálním přiřazením uvnitř jednotky (tzv. *back driving*) – viz [31]. Tedy například:

```
module NetVar (input     logic a,b,  // implikuje propojeni (wire)
               input var logic c,d,  // implikuje promennou
  ...);
```

v deklaracích mohou být celá čísla nebo celočíselné konstanty; tyto hranice mohou být kladné, nulové i záporné. Příklady deklarací vektorů – `Sig1Dim` je jednorozměrný vektor, `Sig2Dim` dvourozměrný:

```
wire [7:0] Sig1Dim;           // jednorozmerny vektor - V,SV
wire [7:0][3:0] Sig2Dim;      // dvourozmerny vektor - jen SV
```

Rozsah může být zapsán jako sestupný (například `[7:0]`) nebo vzestupný (`[0:7]`). Pro stanovení číselné (např. dekadické) hodnoty vektoru se používá číselná interpretace, přičemž se nejlevější bit vektoru považuje za nejvýznamnější (MSB) a nejpravější za nejméně významný (LSB) bez ohledu na vzestupnost či sestupnost rozsahu. Jedním příkazem lze deklarovat i více vektorů stejného typu a stejné bitové šířky, například:

```
wire [7:0] SigA, SigB, SigC;
```

Souvislá část bitů vektoru se označuje za **částečný výběr** (*part-select*). **Odkaz** na ni se zapisuje tak, že se k identifikátoru vektoru připojí rozsah vybraných bitů v hranatých závorkách. Je-li například signálu `Sig1Dim` přiřazena hodnota (způsob zápisu hodnot vektorů bude vysvětlen v **odst. 2.3.9**) příkazem

```
assign Sig1Dim = 8'b00111100;
```

má částečný výběr `Sig1Dim[3:0]` hodnotu `4'b1100`. Bude-li však částečný výběr obsahovat bity, které v původním signálu nejsou, budou tyto bity mít hodnotu `X` – například `Sig1Dim[9:4]` bude mít hodnotu `6'bXX0011`. Za částečný výběr se považuje i zápis, kde je vybrán celý vektor, např. `Sig1Dim[7:0]`. Mezemi částečného výběru mohou být celočíselné konstanty nebo také celočíselné konstantní výrazy (viz **odst. 2.4**). Takto zapsaný částečný výběr lze přesněji označit za částečný výběr s konstantními mezemi, nebo také **neindexovaný částečný výběr** (ve Verilogu podle staršího anglického názvosloví *constant part-select*) – druhý z těchto českých termínů (zapsaný tučně) odpovídá novějšímu názvosloví, které je zavedeno v SystemVerilogu, jak uvidíme v dalším výkladu. Analogicky je definován **bitový výběr** (*bit-select*), kde v hranaté závorce je index vybíraného bitu, například `Sig1Dim[3]` má hodnotu `1'b1`. Tento index může být dán konstantou nebo i výrazem, který nemusí být konstantní. Má-li index hodnotu mimo rozsah odpovídající deklaraci nebo má-li některý z bitů proměnné představující index hodnotu `X` nebo `Z`, je index neplatný. Odkaz na výběr s neplatným indexem dává hodnotu `X` u čtyřhodnotových a `0` u dvouhodnotových veličin.

Verilog i SystemVerilog dovoluje zapsat také odkaz na **indexovaný částečný výběr** (*indexed part-select*). Rozsah výběru je zde dán bází výběru (celočíselný výraz) a jeho šířkou uvedenou za dvojicí znaků `+:` nebo `-:` (celočíselná konstanta nebo celočíselný konstantní výraz). Výraz pro bázi nemusí být konstantní. Například, jsou-li vektory `Vec1`, `Vec2` deklarovány zápisem:

```
logic [31:0] Vec1;
logic [0:31] Vec2;
```

pak jsou ekvivalentní následující zápisy:

`Vec1[ 0 +: 8]` je ekvivalentní zápisu `Vec1[ 7:0]`,

`Vec1[15 -: 8]` je ekvivalentní zápisu `Vec1[15:8]`,

`Vec2[ 0 +: 8]` je ekvivalentní zápisu `Vec1[ 0:7]`,

`Vec2[15 -: 8]` je ekvivalentní zápisu `Vec1[8:15]`.

Částečný výběr se zde provádí s šířkou 8 bitů; znaky `+:` znamenají, že se částečný výběr provádí vzestupným směrem, znaky `-:` znamenají částečný výběr sestupným směrem od báze uvedené před těmito znaky. Vzestupnost nebo sestupnost rozsahu vektoru, z něhož se výběr provádí, zůstává zachována.

Podívejme se na další, praktičtější příklad:

```
logic [63:0] Word;                                          // V,SV
logic  [2:0] ByteNo; //poradi vybraneho slova ByteN ve vektoru Word (0-7)
wire   [7:0] ByteN = Word[ByteNo*8 +: 8];
```

Zde `ByteNo*8` je báze indexovaného částečného výběru s šířkou 8 bitů. Tímto zápisem se zde vybírá slabika z proměnné `Word` začínající na pozici určené proměnnou `ByteNo`.

```
package InstrTypes;                              // jen SV  // TXT2.3
  typedef enum {NOP,ADD,SUB,MUL} Instr_t;
endpackage
import InstrTypes::*;                  // import do $unit
module Controller (output logic [7:0] Result,
                   input  logic [1:0] InstrCode,
                   input [3:0]        DataIn,
                   input              Clk,Btn);
  enum {IDLE,GO} PrSt,NxSt; // stav.reg. - souc.(Pr) a pristi (Nx) stav
  Instr_t Instr;
// Debouncer ---------------------------------------
  logic BtnDeb = '0;               // debounced Btn
  logic BtnDebDld = '0;
  logic BtnReg = '0;
  logic [19:0] Debouncer;          // 50 MHz -> 50 Hz
  always_ff @(posedge Clk) begin  // debouncer counter
    BtnReg <= Btn;
    Debouncer <= Debouncer + 1;
    if (Btn^BtnReg) Debouncer <= '0;
    if (&Debouncer) BtnDeb <= Btn;
    BtnDebDld <= BtnDeb;
  end
// Controller State Machine -------------------------
  assign Instr = Instr_t'(InstrCode);  // type casting
  always_ff @(posedge Clk)
    PrSt <= NxSt;
  always_comb
    case (PrSt) // GO is one Clk cycle after BtnDeb rising edge
      IDLE: if (BtnDeb && ~BtnDebDld) NxSt = GO;
            else NxSt <= IDLE;
      GO:   NxSt = IDLE;
    endcase
  always_ff @(posedge Clk)
    if (PrSt == GO)
      case (Instr)
        default: Result <= Result;
        ADD: Result <= Result + DataIn;
        SUB: Result <= Result - DataIn;
        MUL: Result <= Result * DataIn;
      endcase
endmodule
```

Procesor popsaný textem `TXT2.3` reaguje na vzestupnou hranu signálu `Btn` provedením operace určené signálem `InstrCode`. Předpokládá se, že signál `Btn` bude generován mechanickým kontaktem, takže v modelu je zahrnut `Debouncer` – blok pro odstranění zákmitů. Stavy procesoru jsou definovány v modulu `Controller`, nejsou tedy viditelné mimo tento modul. Typy instrukcí jsou definovány ve sloze `InstrTypes`, a do kompilační jednotky jsou importovány. Mohou tedy být využity podobným způsobem i v dalších kompilačních jednotkách, případně mohou být importovány přímo do konstrukčních prvků. Konkrétní typ prováděné instrukce je určen aktuální hodnotou signálu `InstrCode` – tento signál je přiváděn vstupní bránou a je převeden na signál `Instr`. Ten má výčtový typ `Instr_t`, a vnitřně je reprezentován celočíselnými hodnotami. Pokud by vstupem kódu instrukce byla brána tohoto typu, byl by její signál syntetizován jako 32bitový. V uvedeném případě stačí pro vstup kódu instrukce 2bitový signál `InstrCode`.

Jinou, pro syntézu patrně ještě vhodnější možností, by bylo zvolit explicitní specifikaci typu a explicitní enumeraci pro typ `Instr_t` (ve sloze `InstrTypes`), například:

```
module ArrAssignUnpack2Pack (input logic Data2 [3:0][7:0][1:0],
                             output TypeDefinition::UnpckType_t Data1,
                             output TypeDefinition::PackType_t Data0,
                             output TypeDefinition::UnpckTypeInt_t DataX);
  import TypeDefinition::*;
  always_comb begin
    Data1 = UnpckType_t'(Data2); // prevod nesbal <- nesbal logic
    Data0 = PackType_t'(Data2);  // prevod   sbal <- nesbal logic
    DataX = UnpckTypeInt_t'(Data2); // nesbal int <- nesbal logic
  end
endmodule
```

V textu `TXT2.9` jsou uživatelské typy definovány ve sloze `TypeDefinition`. V deklaracích bran jsou na ně uvedeny přímé odkazy, v následujícím textu je importován celý obsah slohy. Jinou možností by zde bylo importovat obsah slohy do kompilační jednotky, nebo přesunout příkaz importu do hlavičky, jak to bylo uvedeno na konci **odst. 2.2.3** – pak by přímé odkazy v deklaracích bran nebyly potřebné. Převod bitovým proudem je zde demonstrován třemi procedurálními přiřazovacími příkazy.

2.4 Výrazy a operátory

Výraz (*expression*) je složen z **operandů** (jako jsou identifikátory signálů včetně vektorů a jejich částečných či bitových výběrů, polí či jejich prvků, volání funkcí, konstantní veličiny zapsané ve formě literálů, parametrů či konstant atd.) a z **operátorů**. Za výraz se považuje i samotný operand. **Konstantní výraz** obsahuje vedle operátorů jen konstanty, případně již definované parametry, a může obsahovat také volání konstantních funkcí (viz **odst. 2.7**). V dalším textu této kapitoly budeme pracovat jen s operandy celočíselných datových typů (**odst. 2.3.5**), nebude-li výslovně uvedeno jinak.

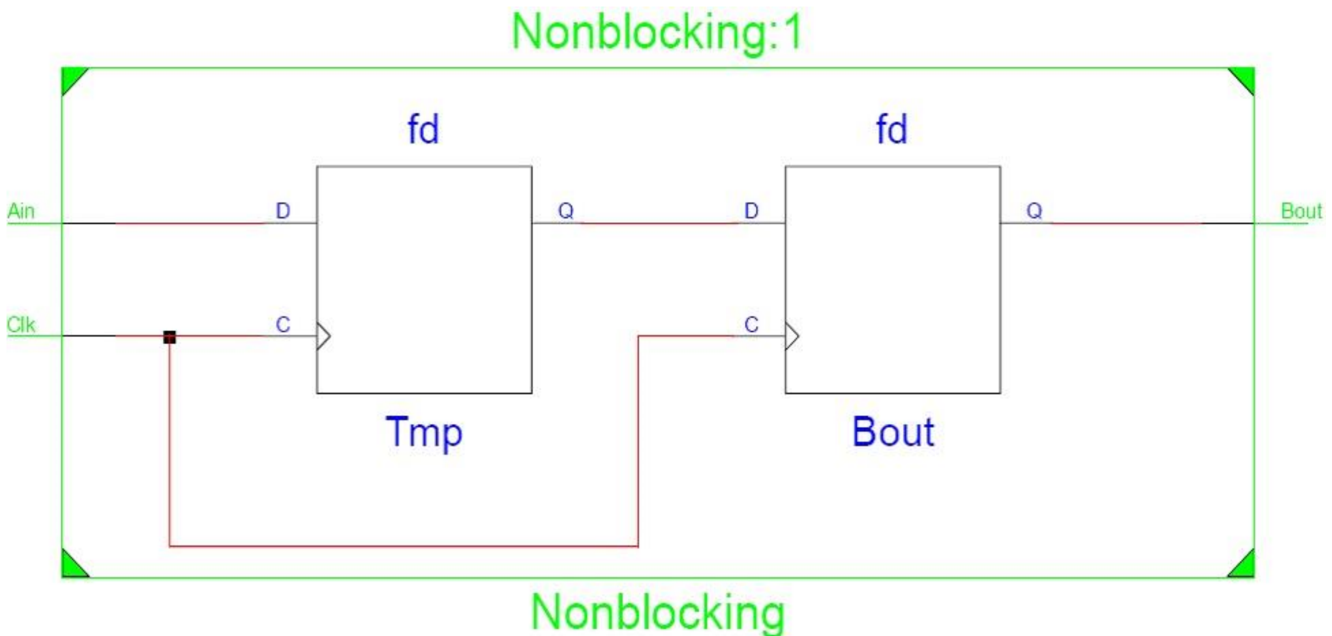
Výrazy jsou při simulaci **vyhodnocovány** (*evaluated*). Výsledkem vyhodnocení výrazu je datový objekt, který může být například částí dalšího výrazu, jeho hodnota může být přiřazena signálu a podobně. Pravidla pro vyhodnocování výrazů budou uvedena v **odst. 2.4.9**.

Operátory ve Verilogu i v SystemVerilogu jsou podobné operátorům v jazyku C. V této podkapitole prodiskutujeme operátory, s nimiž se nejčastěji setkáme ve výrazech. Standard SystemVerilogu definuje ještě řadu dalších operátorů, s nimiž se zejména v syntéze setkáme málokdy; o některých z nich se zmíníme jen stručně, nebo je vynecháme. Pomineme také pravidla týkající se méně běžných operací, zejména většiny operací, kde operandy mají hodnoty `x` a `z`, které se při syntéze vyskytují zřídka. Někdy se méně běžné operátory označují i jinými názvy, než jak je označujeme zde. Úplné informace najde zájemce v manuálu **[6]**.

2.4.1 Bitové a redukční operátory

K často používaným operátorům patří **bitové (bitově orientované) operátory** (*bitwise operators*); někdy se jim říká také **booleovské** (*boolean*) operátory. Jsou to tyto operátory:

- `~` – NOT, negace;
- `&` – AND, logický součin;
- `|` – OR, logický součet;
- `^` – EX-OR, exkluzivní logický součet;
- `~^` nebo `^~` – EX-NOR, negace exkluzivního logického součtu.



Obr. 2.3. *Výsledek zobrazení RTL Schematic u textu* `TXT2.13`

Je zřejmé, že první schéma neodpovídá posuvnému registru, protože při syntéze byl optimalizací vyřazen vnitřní uzel se signálem `Tmp`. Žádoucí výsledek dává v souhlasu s výše uvedeným doporučením neblokující přiřazení.

Mnemotechnická pomůcka k tomu, abychom si zapamatovali správný tvar přiřazovacího operátoru: vytváří-li se v bloku skupiny **`always`** paměťový obvod s hodinovým signálem, který se ve schématech značí šipkou na vstupech klopných obvodů, bude šipka i u přiřazovacího operátoru – neblokující přiřazení. (Operátor pro neblokující přiřazení se však používá i u popisu klopných obvodů se statickým řízením – latch, u něhož ve schematické značce šipku nekreslíme.)

Bohužel i z uvedených pravidel existují výjimky. Neblokující přiřazení je určeno k modelování synchronních subsystémů s klopnými obvody či registry na úrovni RTL, tedy s jedním společným hodinovým signálem, mezi nimiž se v rytmu hodinového signálu přenášejí data. Doporučení však nemusí platit pro modely non-RTL. Jde-li například o subsystém s více hodinovými signály, jako o subsystém s děličem kmitočtu hodinového signálu (podrobnější analýza je v **[12]**), může neblokující přiřazení v něm vést k nesprávnému výsledku simulace RTL u obvodů, které vydělený signál (zde `ClkDiv2`) používají jako hodinový signál. Správný popis takového děliče (se synchronním nulováním) podle **[12]** je:

```
always @(posedge Clk)                  // V,SV
  if (!SRstN) ClkDiv2 = 0;
  else ClkDiv2 = ~ClkDiv2;
```

Při použití neblokujícího přiřazení zde mohou vzniknout kritické souběhy. Syntéza blokujícího přiřazení proběhne správně, protože dělič nedostává data z jiných obvodů ani data nepředává do jiných bloků **`always`**. K rozhodnutí o typu přiřazení správnému za všech okolností pro syntézu i pro simulaci nevede jiná cesta než důkladné porozumění podstatě obou typů přiřazení, což však není jednoduché a nebude zde podrobně probíráno. Nicméně simulace modelu post-synthesis, popř. post-place-and-route (o těchto typech simulace budeme blíže hovořit v **kapitole 4**) by měla dát výsledek odpovídající skutečnému chování cílového obvodu po implementaci. Zájemce najde více informací například v literatuře **[39]**.

Procedurální příkazy v blocích skupiny **`always`** se při simulaci RTL vykonávají v pořadí, v němž jsou zapsány. Je-li výsledek z jednoho přiřazovacího příkazu kombinačního bloku **`always`** vstupní hodnotou druhého příkazu, je k správnému provedení simulace modelu RTL potřebné, aby pořadí jejich zápisu nebylo přehozené, jinak skutečnost, že proměnné při simulaci jsou statické (**odst. 2.6.1**), znamená, že výsledek simulace modelu RTL může být nesprávný – simulátor si pamatuje hodnotu signálu na pravé straně prvního příkazu z předchozího provedení bloku **`always`**. Jednoduchý příklad:

```
module CombRepeater (input In,              // V*,SV          // TXT2.14
                     output logic Out1,Out2);
  always_comb begin
    Out1 = In;
    Out2 = Out1;
  end
endmodule
```

V textu TXT2.14 je modelován opakovač se dvěma výstupy (může sloužit například pro převod vstupně/výstupních standardů). Při simulaci modelu RTL se provedení bloku `always_comb` spustí vždy při změně hodnoty signálu `In`, a všechny tři signály – `In`, `Out1`, `Out2` budou mít hodnotu v každém okamžiku stejnou (což zde asi bylo záměrem konstruktéra). Pokud se však pořadí přiřazovacích příkazů zamění, bude simulovaná hodnota signálu `Out2` rovna předchozí hodnotě signálu `Out1`, tedy inverzi současné hodnoty tohoto signálu – signály `Out1` a `Out2` budou mít hodnoty v každém okamžiku vzájemně komplementární. Při syntéze však paměťová funkce u proměnných nenastává a oba výstupní signály budou mít stejnou hodnotu. S výsledky syntézy tedy výsledky simulace modelu RTL nesouhlasí, měly by však s nimi souhlasit výsledky simulace modelu post-fit, popř. post-place-and-route. Mají-li tedy souhlasit výsledky všech typů simulace a také výsledky syntézy, je třeba věnovat pořadí zápisu takových příkazů v blocích skupiny `always` pozornost – na rozdíl od zápisu kontinuálních příkazů, jejichž pořadí je libovolné, protože tyto příkazy mají souběžný charakter.

2.6.4 SystemVerilog: Přiřazení uvnitř výrazů

Podobně jako v jazyku C je i v SystemVerilogu dovolen zápis **přiřazovacího příkazu s blokujícím přiřazením uvnitř výrazu** (to však není dovoleno ve Verilogu). Výraz může obsahovat blokující přiřazovací příkaz, ve kterém není řízení času. Takový příkaz je nutno uzavřít do závorek. Výraz obsahující přiřazení může být **jen v procedurálním příkazu**, tedy v blocích skupiny `always`, `initial`, v definicích funkcí a úloh, nikoliv v kontinuálním příkazu. Příklad:

```
always_comb begin a = (b = (c = 2)); // priradi se 2 vsem trem promennym
  if ((d=e)) a = (b+=1); // promenne d se priradi hodnota e, potom
                         // jsou-li d, e nenulove, inkrementuje se b,
                         // vysledek se priradi promenne a
end
```

Na rozdíl od syntaxe v jazyku C musí být přiřazovací příkaz zapsaný ve výrazu vždy uzavřen v samostatné závorce, jak je to vidět v uvedeném příkladu: vnější závorka v příkazu `if` zapsaná hned za tímto slovem je prvkem tohoto příkazu, vnitřní závorka uzavírá přiřazovací příkaz, který představuje podmínku `(d=e)`. Při vyhodnocení výrazu v přiřazení se podle [6] postupuje jako při vyhodnocení funkce: vyčíslí se pravá strana, její datový typ se převede na datový typ levé strany, výsledek se uklidí do zásobníku, levá strana se aktualizuje, a nakonec se vrátí hodnota ze zásobníku. Je-li levá strana výrazu v přiřazení zapsána jako sjednocení, pak se vrací hodnota integrálního datového typu bez znaménka s bitovou šířkou odpovídající součtu bitových šířek operandů.

V jazyku C není vnitřní závorka potřebná. Podle [11] byla v SystemVerilogu tato odlišnost od úpravy v jazyku C zavedena z toho důvodu, aby byl jasněji vyjádřen charakter přiřazovacího příkazu uvedeného v závorce (podmínku v příkazu `if` ve výše uvedeném příkladu lze omylem snadno zaměnit za výraz představující porovnání – ten by zde však byl zapsán bez vnitřní závorky a s jiným operátorem, například ==).

S přiřazením uvnitř výrazů je třeba zacházet opatrně. Simulátory a syntetizéry nezpracovávají složitější výrazy tohoto druhu – tak například ve výše uvedeném příkladu hlásí simulátor ModelSim syntaktickou chybu, pokud se před přiřazovací příkaz v podmínce příkazu `if` zapíše operátor inverze: `if (~(d=e)) ...`

3.3.4 Příklad: testování modelu paměti RAM

Uveďme nyní příklad jednoduchého projektu využívajícího rozhraní. Jde o konstrukci paměti RAM (modul `RamTpl`) odpovídající šabloně v systému Quartus II (*Single port RAM with single read/write address*) a o test této paměti. Deklarace vstupních a výstupních datových signálů i adresových signálů v hlavičce je nahrazena použitím rozhraní `RamBus_if`, které tyto deklarace obsahuje, jako brány. Projekt obsahuje benč (zkušební jednotku) `Ram_TF`, kde jsou tyto deklarace opět nahrazeny, a to tentokrát vložením rozhraní `RamBus_if` jako komponenty, k níž je pak připojena odpovídající brána `Bus` modelu paměti. Test paměti je uveden jen náznakem, bez snahy o úplnost, hlavním účelem je zde demonstrace syntaxe. Syntéza modelu paměti `RamTpl` byla úspěšně vyzkoušena v systému Quartus.

```
package ParamDef;                                              // TXT3.1
  parameter DataWdth = 8, AddrWdth = 6;
endpackage
interface RamBus_if(input Clk,WrEn);
  import ParamDef::*;
  logic [(DataWdth-1):0] DaIn,DaOut;
  logic [(AddrWdth-1):0] Addr;
endinterface
module RamTpl(RamBus_if Bus);      // rozhrani jako brana s nazvem Bus
  import ParamDef::*;
  logic [(DataWdth-1):0] RamArr [2**(AddrWdth-1):0];
  logic [(AddrWdth-1):0] AddrReg;  // registr adresy
  always_ff @(posedge Bus.Clk) begin
    if (Bus.WrEn) RamArr[Bus.Addr] <= Bus.DaIn;  // zapis do pameti
    AddrReg <= Bus.Addr;           // ulozeni adresy do registru
  end
  assign Bus.DaOut = RamArr[AddrReg];
// Continuous assignment implies read returns NEW data.
// This is the natural behavior of the TriMatrix memory
// blocks in Single Port mode.
endmodule
module Ram_TF;
  logic Clock,WriteEn;
  RamBus_if Bus(.Clk(Clock),.WrEn(WriteEn)); // vlozeni rozhrani
  RamTpl uut(.*);                            // vlozeni pameti
  initial begin
    WriteEn = 1'b1; Bus.Addr = 0; Bus.DaIn = 8'h11;
    #10 Clock = 1'b1; #10 Clock = 1'b0;
    for (int i = 0;  i <= 7; i++) begin
      Bus.DaIn += 1;
      Bus.Addr += 1;
      #10 Clock = 1'b1; #10 Clock = 1'b0;
    end
    #10 WriteEn = 1'b0; Bus.Addr = 0;
    #10 Clock = 1'b1; #10 Clock = 1'b0;
    for (int i = 0;  i <= 7; i++) begin
      #10 Clock = 1'b1; #10 Clock = 1'b0;
      Bus.Addr += 1;
    end
  end
endmodule
```

V rozhraní `RamBus_if` jsou deklarovány signály společné paměti a benči, přičemž datové šířky jsou určeny pomocí parametrů (o tom jsme podrobněji mluvili v odst. 2.3.10). Syntetizovatelný model paměti (modul `RamTpl`) je vytvořen na základě výše zmíněné šablony systému Quartus, rozhraní je v něm použito