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

nebo z literatury, a měli by být schopni tyto texty převzít, porozumět jim a upravit je podle vlastní potřeby. Proto lze očekávat, že uvítají příručku, která může sloužit i jako učebnice jazyků Verilog a SystemVerilog a může představovat vodítko, jak psát zdrojové texty pro návrhové systémy představující syntetizovatelné modely vytvářených číslicových systémů.

Prostředky jazyků Verilog a VHDL dovolují efektivně modelovat číslicové bloky na úrovni modulů. Oba tyto jazyky podporují ovšem taky hierarchičnost projektů, při projektech rozsáhlých systémů se však popisy zapsané v těchto jazycích stávají dosti nepřehlednými. SystemVerilog, podobně jako jazyk SystemC, obsahuje nové jazykové konstrukty (jako příklad uveďme rozhraní – *interface*), které výrazně zlepšují přehlednost popisů takových systémů a usnadňují práci s nimi, a jsou příslibem jednotné metodiky pro konstrukci a verifikaci celých systémů na čipu (*System-on-Chip*, SoC) od konstukce na úrovni modulů a od simulace na úrovni hradel až po verifikaci na systémové úrovni. Odtud vychází název jazyka. SystemVerilog dále obsahuje proti Verilogu řadu zlepšení, která jej činí přátelštější pro uživatele, a to i u menších projektů.

Textové formy popisu modelů nejsou jediné. Kromě nich se v návrhových systémech setkáme i s dalšími formami popisu, zejména grafickými – jsou to například schémata nebo stavové diagramy. Práce s nimi je značně odlišná u různých návrhových systémů a nebudeme ji zde podrobněji diskutovat. Jen jako příklad alternativy strukturálního popisu v jazyku Verilog uvedeme v odst. 5.1.5 ukázku blokového schématu vytvořeného v editoru schémat systému ISE.

Velká většina zdrojových textů zapsaných v jazyku SystemVerilog, které jsou uvedeny jako příklady v této knize, byla ověřena z hlediska syntaktické správnosti a možnosti simulace a syntézy systémem Quartus II firmy Altera. Odkazy na tento systém i na syntetizér XST (*Xilinx Synthesis Technology*) systému ISE se týkají jeho verze aktuální v době zpracování knihy, tj. verze 11. Příklady textů zapsaných ve Verilogu byly ověřovány také v systému ISE.

Terminologii věnují oficiální dokumenty jazyka Verilog, na který SystemVerilog navazuje, výrazně menší pozornost než odpovídající dokumentace k jazyku VHDL. Některé pojmy jsou v tomto jazyku používány víceméně volně, a často nejsou definovány termíny obdobné termínům jazyka VHDL, i když by se to mohlo jako účelné očekávat. Je to zřejmé zejména u označení typů signálů – například pojem *typ* není ve Verilogu vůbec oficiálně zaveden, i když se často používá ve smyslu dosti podobném jako v jazyku VHDL. V SystemVerilogu se přes zjevnou snahu o nápravu nepodařilo tento nedostatek zcela odstranit, přičemž dosti značnou nesnáz zřejmě představoval požadavek dodržení kompatibility s Verilogem. Pro čtenáře zvyklého na přesnější názvosloví je matoucí (hlavně při zápisu textů určených k syntéze) také označení *register*, používané ve starších anglických publikacích o Verilogu, jako je například [16], pro proměnné typů **`reg`**, **`integer`**, **`time`**, **`real`** a **`realtime`**. Toto označení nemá nic společného s obvodovým prvkem – klopným obvodem či registrem, a myslí se jím paměťové místo v paměti počítače, na němž běží simulace kódu zapsaného ve Verilogu. Označení proměnné slovem *register* bylo ve Verilogu-2001 opuštěno a zbylo po něm jen klíčové slovo **`reg`**. V SystemVerilogu je definován nový typ **`logic`** s vlastnostmi shodnými s typem **`reg`**, jehož název nevyvolává nesprávné asociace (význam obou je však v SystemVerilogu poněkud posunut). Někteří autoři píšící v angličtině se snaží terminologické mezery vyplnit, často právě na základě terminologie zavedené v jazyku VHDL. Terminologie používaná v této knize vychází z dokumentu [6] a z doplnění tam zavedené terminologie ponejvíce podle podle pramenu [7], s některými drobnými odchylkami – například pro popisy zapsané v jazycích HDL se v [7] (často i jinde) používá označení „program“, kdežto autor se tomuto označení vyhýbá, protože podle jeho názoru existuje dosti zásadní rozdíl mezi počítačovým programem a popisem hardwarového uspořádání, kterému odpovídá popis v jazyku HDL; za vhodnější považuje například výrazy „zdrojový text“, „kód“ nebo prostě „popis“. (V SystemVerilogu má slovo **`program`** nově definovaný význam

se stávají), ale hlavně z hlediska kvality výsledků, tedy z hlediska spotřeby strukturních prvků, zpoždění, odběru z napájecího zdroje a podobně. Je také nutno počítat s tím, že různé syntetizéry podporují odlišný rozsah prostředků jazyka, což samozřejmě platí i pro SystemVerilog. Postupem doby, jak se budou syntetizéry zdokonalovat, se patrně bude rozšiřovat oblast jazykových konstruktů využitelných v syntéze, které budou dávat výsledky syntézy skutečně nezávislé na použitých návrhových systémech. Dále je nutno poznamenat, že syntetizéry používají často volnější pravidla syntaxe než simulátory. Hlásí-li například simulátor chybu, syntetizér někdy vydá pouze varování a pokračuje dále v syntéze, pokud může odhadnout úmysl konstruktéra.

I když tedy v principu by měl být kód zapsaný v jazycích Verilog a SystemVerilog přenositelný na různé systémy a cílové obvody, není tato přenositelnost úplná. Pro jednoduché konstrukce, kde se vystačí se základními logickými bloky, nečiní obvykle přenositelnost problémy. Ve složitějších konstrukcích, zejména pokud se v nich využívají speciální bloky typické pro současné obvody FPGA dovolující výrazně zlepšit vlastnosti výsledku, je přenositelnost omezená, a k dosažení optimálního výsledku musí mít konstruktér podrobnější znalosti i o funkci návrhových systémů a o struktuře cílových obvodů. Například v obvodech CPLD řady CoolRunner se vyskytují klopné obvody reagující na obě hrany hodinového signálu. Odpovídající popis (viz odst. 2.6.9) je syntetizovatelný pouze do těchto cílových obvodů. Podobné poznatky jsou v předkládané publikaci zmíněny jen okrajově. Podrobněji se s nimi může čtenář seznámit v katalogových listech obvodů FPGA a CPLD, v uživatelských příručkách a v podobných publikacích výrobců programovatelných obvodů.

Kromě zápisu kódu v jazycích Verilog a SystemVerilog bývá účelné zejména v hierarchických konstrukcích s menším rozsahem používat k popisu také schémata (hlavně bloková, která představují grafickou podobu strukturálního popisu). Tento popis může být výhodnější než textový strukturální popis zejména pro lepší přehlednost, nebývá však přenositelný do jiných návrhových systémů a u rozsáhlejších konstrukcí přestává být použitelný. K prohlížení schématu musíme použít editor z návrhového systému, nestačí k tomu např. běžné textové editory. V současné době většinou nebývá smysluplné popisovat schématem takové obvodové bloky, které je možno popsat textově behaviorálním stylem (schémata tohoto druhu se používala k popisu konstrukcí zejména pro obvody FPGA v době, kdy ještě nebyly k dispozici prostředky pro syntézu z jazyků HDL, a někteří starší konstruktéři jsou na tento způsob popisu zvyklí, i když jej lze dnes pokládat za překonaný). Výhodou textového strukturálního popisu je možnost použití některých vyšších jazykových konstruktů (například smyček, příkazů `generate` a podobně), které mohou textový popis výrazně zkrátit. V SystemVerilogu jsou zavedeny nové konstrukty – například rozhraní, které u rozsáhlých strukturálních popisů výrazně zlepšují jejich efektivitu a přehlednost. (Pojmy behaviorální a strukturální popis budou vysvětleny v odst. 2.1.1.)

V dalším textu se budou vyskytovat termíny kompilace, analýza a elaborace. Připomeneme stručně jejich význam. **Kompilace** (*compilation*) obvykle probíhá ve dvou fázích: analýza a elaborace. **Analýza** (*analysis*) představuje převod zdrojového textu či schématu do vnitřní formy popisu akceptované systémem, 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.

Model **plochého typu** (*flat*) neobsahuje hierarchické členění. Představuje konstrukci jako jeden monolitický blok. Současné syntetizéry a optimalizační programy optimalizují takový model jako celek, zatímco u hierarchických popisů se zpravidla optimalizace provádí jen v rozsahu jednotlivých hierarchických bloků. Návrhové systémy proto před optimalizací často převádějí hierarchické modely na ploché, aby bylo možno provést optimalizaci v celém rozsahu. To však znamená, že orientace v takovém popisu, například nalezení a testování vnitřních signálů při simulaci, je obtížnější. Proto návrhové systémy obvykle dovolují zakázat rozvinutí hierarchického modelu do plochého, což může být výhodné ve fázi ladění a testování, a po odladění se povolením rozvinutí do plochého modelu může získat lépe optimalizovaný výsledek s lepšími parametry.

V souhlasu s terminologií výrobců návrhových systémů budeme označovat souhrn souborů týkajících se konstrukce vytvářené v návrhovém systému názvem **projekt** (*project*).

Příkazy v modulu mohou jeho funkci popisovat **behaviorálním stylem** (*behavioral style*) nebo **strukturálním stylem** (*structural style*). Behaviorální styl popisuje chování částí (komponent) modulu, zatímco strukturální styl představuje soupis komponent obsažených v modulu a jejich propojení (*netlist*). Pro pochopení funkce modelu (či jeho části) popisovaného behaviorálním stylem v podstatě stačí znát syntaxi jazyka, zatímco funkci komponenty ve strukturálním popisu může napovídat jen její název a ostatní údaje

Připomeňme také, že z hlediska syntaxe nemají komentáře význam (můžeme je beze změny funkčnosti popisu vynechat) a že znak nového řádku je prázdným znakem, takže může být nahrazen například mezerou. Hlavičku modulu z textu TXT2.1 tedy můžeme beze změny funkčnosti zapsat i v jednom řádku:

```
module EqComp4(input [3:0] x,y, output z);
```

Kromě deklarací bran s jejich signály obsahují moduly většinou také deklarace vnitřních signálů, které nejsou připojeny k branám – datových objektů, jako jsou proměnné a propojení (o tom více v odst. 2.3.1), a také parametrů, případně deklarace funkcí a úloh. Modul dále obsahuje **příkazy** (*statements*), které specifikují jeho funkci.

V řádku `3` textu TXT2.1 je přiřazovací příkaz; jednoduché rovnítko je přiřazovací operátor, zatímco dvojité rovnítko je operátor porovnání, a výraz v závorce má jedničkovou hodnotu, jsou-li si vektory `x` a `y` rovny. Definice modulu je ukončena klíčovým slovem **`endmodule`**.

V manuálech [1], [6] se bráně definované deklarací odpovídající textu TXT2.1, tedy uvedením identifikátoru jejího signálu v seznamu bran, říká **implicitní brána** (*implicit port*). To je nejčastější způsob definice brány. Jsou tak deklarovány shodné identifikátory brány i signálu této brány, tj. vnitřního signálu připojeného k bráně. Tyto manuály připouštějí také podrobnější definici – **explicitní bránu** (*explicit port*; výstižnější by však asi bylo mluvit o explicitní definici brány), kde je v seznamu bran explicitně uveden identifikátor brány i vnitřního signálu k ní připojeného (oba identifikátory mohou být odlišné); vnitřní signál je pak deklarován uvnitř modulu. Jako příklad uveďme hlavičku modulu, kde je k explicitní bráně `p` připojen vnitřní signál `q`, s následnou deklarací tohoto signálu:

```
module ExplPort (input .p(q), ...);
  logic q; ...    // deklarace vnitrniho signalu pripojeneho k brane
```

V odst. 2.8.2 uvidíme, že tato syntaxe odpovídá také syntaxi vložení komponent.

Standardy Verilogu a SystemVerilogu připouštějí ještě další varianty deklarace bran a jejich signálů. Například signály bran mohou být i bitové nebo částečné výběry z vektorových signálů nebo sjednocení signálů – tzv. **výraz v deklaraci brány** (*port expression*). Takové možnosti deklarací se však používají zřídka a nebudeme je zde podrobně uvádět.

Funkci bran mohou ve Verilogu zastávat propojení nebo proměnné (skalárního nebo vektorového charakteru, ne však pole), v SystemVerilogu to mohou být i pole, rozhraní, struktury nebo uniony. Význam těchto pojmů bude vysvětlen v dalším výkladu.

Text TXT2.1 je typický příklad způsobu zápisu, s nímž se často setkáme u jednoduchých zdrojových textů. V deklaraci bran se zde spoléhá na implicitní nastavení datového typu jejich signálů. Doporučuje se však explicitně uvádět v hlavičce směr i datový typ signálů, a to zejména tehdy, když se předpokládá, že text může být později upravován, přičemž mohou být přidávány další brány – neurčené atributy bran (signálů) se v deklaraci „dědí“ od bran (signálů) deklarovaných dříve, což může být snadno zdrojem omylů při dodatečném vkládání dalších bran do seznamu. Hlavička s explicitním uvedením směru i datového typu signálů v SystemVerilogu (ve Verilogu by byl místo datového typu **`logic`** deklarován datový typ **`wire`**) může mít tvar:

```
module EqComp4 (input logic [3:0] x,y,
                output logic z);
```

Do deklarací vnitřních signálů mohou být přímo zapsány i přiřazovací příkazy, jejichž význam se liší v případech, jde-li o přiřazení hodnoty propojení nebo proměnné – viz odst. 2.5.1, jako příklad může sloužit text TXT5.1.5d.

Brány představují přiřazení s vlastnostmi kontinuálních přiřazovacích příkazů (odst. 2.5.1) – u vstupní brány jde o přiřazení hodnoty jejímu signálu, u výstupní brány se hodnota signálu brány přiřazuje signálu k bráně zevně připojenému.

Omezení kladená na signály přenášené branami ve Verilogu:

- brány mohou představovat budiče jen pro signály typu propojení (*net*), nikoli pro proměnné; brány však mohou získávat hodnoty od signálů typu propojení, od proměnných (**reg** nebo **integer**) nebo i od číselných literálů, což vyplývá z představy brány jako kontinuálního přiřazení;
- objekty typu **real** mohou být přenášeny branami jen po převedení do vektorového tvaru (použitím systémové funkce *$realtobits*), po přenesení mohou být zpět převedeny do reálného tvaru systémovou funkcí *$bitstoreal*;
- branami ve Verilogu nelze přenášet pole (to se netýká vektorů, které se zde nepokládají za pole).

V SystemVerilogu jsou téměř všechna podobná omezení odstraněna. Branami zde mohou být přenášeny jakékoliv signály včetně polí, struktur, unionů a reálných veličin. O předávání polí branami budeme podrobněji mluvit v odst. 2.3.11.7, o strukturách budeme v tomto smyslu diskutovat v odst. 3.1.4.

Naznačíme nyní syntaxi hlavičky ve Verilogu-95, ve Verilogu-2001 a v SystemVerilogu. Úplné znění příslušných pravidel je poměrně komplikované, což je mimo jiné zřejmě důsledkem požadavku zpětné kompatibility SystemVerilogu s Verilogem. Místo podrobného výkladu těchto pravidel si syntaxi vysvětlíme na příkladu, což bude pro velkou většinu praktických případů postačovat.

2.2.1.1 Brány ve Verilogu-95

Často, zejména ve starších textech, se setkáme se zápisem podle syntaxe Verilog-95, která vyžaduje výčet pouhých identifikátorů bran v seznamu a následující samostatnou deklaraci příslušných signálů. Takový zápis se označuje za styl non-ANSI. Pro výše uvedený komparátor má hlavička v této syntaxi tvar:

```
module EqComp4 (x,y,z);
  input [3:0] x,y;
  output z;
```

Zápis hlavičky stylem non-ANSI je u rozsáhlejších konstrukcí dosti nepohodlný, protože se musí vícekrát opakovat specifikace pro jednotlivé brány: nejprve jen identifikátory v seznamu bran, pak znovu identifikátory s deklarací směru a případně rozsahu pro každou bránu, a dále opět identifikátor, rozsah a typ všech bran, které nejsou typu **wire** (jak uvidíme v odst. 2.3.1, u bran, u nichž není explicitně deklarován typ, se předpokládá typ **wire**). V tomto odstavci, kde se jedná o syntaxi Verilogu, se přidržíme názvosloví Verilogu – místo „datový typ“ budeme psát pouze „typ“. Podívejme se na hlavičku trochu složitějšího modulu:

```
module Processor (DataIO,DataIn,DataOut,Abc,Clk,Flg1,Flg2);

  inout  [15:0] DataIO;  // deklarace smeru a rozsahu
  input  [15:0] DataIn;
  output [15:0] DataOut;
  output  [2:0] Abc;
  input         Clk;
  output        Flg1,Flg2;

  tri    [15:0] DataIO;  // deklarace typu a rozsahu
  wire   [15:0] DataIn,DataOut;
  reg     [2:0] Abc;
  wire          Clk;
```

Seznam bran zde definuje identifikátory bran a jejich pořadí, které může být důležité například při vkládání modulu jako komponenty. Následuje specifikace směru a rozsahu každé brány. Není-li rozsah brány uveden, jde o jednobitovou bránu. Dále je třeba ještě deklarovat typ signálů bran, opět následovaný rozsahem

definice modifikovat – pak je modifikaci nutno provést ve všech modulech, které tuto definici (deklaraci) obsahují.

V SystemVerilogu se tyto definice (deklarace) mohou objevit také v definicích rozhraní (mezi klíčovými slovy `interface` a `endinterface`) i v dalších konstrukčních prvcích, opět s lokální působností. Žádná z takových definic (deklarací) není platná mimo konstrukční prvek (jako modul nebo rozhraní), v němž se nachází. Je sice možno používat tzv. hierarchické odkazy, ty však nejsou syntetizovatelné. (SystemVerilog dovoluje také deklarovat lokální proměnné v nepojmenovaných blocích; takové proměnné nejsou viditelné mimo hranice bloku, v němž jsou deklarovány, a nejsou přístupné pro hierarchické odkazy – u textů určených k syntéze to ale nevadí, pokud to nekomplikuje jejich verifikaci.)

Nepohodlí spojené s nutností opakovaně definovat objekty řeší SystemVerilog pomocí **sloh** (klíčové slovo `package`), do nichž je možno definice umístit a odkazovat se na ně v dalších jednotkách, nebo odtud definice či celé slohy importovat, takže objekty jsou definovány jen jednou v celé konstrukci. Podrobnější řeč o slohách bude v odst. 2.2.3. Ve Verilogu nejsou slohy zavedeny.

SystemVerilog zavádí dále pojem **kompilační jednotky** (*compilation unit*), která se v hierarchických odkazech označuje `$unit`. Kompilační jednotku tvoří všechny zdrojové soubory, které se společně kompilují v jednom běhu kompilace. Standard SystemVerilogu nepředepisuje, které soubory budou v konkrétním případě zařazeny do kompilační jednotky, to je podrobně definováno až v jednotlivých nástrojích. Ty však musí dovolovat sestavení kompilačních jednotek přinejmenším oběma následujícími způsoby:

- kompilační jednotku tvoří všechny soubory uvedené na příkazovém řádku nástroje;
- každý soubor je samostatnou kompilační jednotkou.

Jsou také přípustné jiné způsoby sestavení kompilačních jednotek, což závisí na konkrétním nástroji, a není pro ně zaručena přenositelnost. Obsahuje-li některý ze souborů direktivy `` `include `` (jednu nebo několik těchto direktiv), jsou soubory z nich rovněž součástí příslušné kompilační jednotky. Kompilační jednotky tak dovolují programovým nástrojům odděleně kompilovat **podbloky** (*sub-blocks*) v konstrukci, které mohou obsahovat jeden nebo několik souborů, v nichž mohou být definice modulů, rozhraní nebo programových bloků [14].

V prostoru kompilační jednotky může být definována jakákoliv položka, pro jejíž definici je přípustné umístění ve sloze (viz odst. 2.2.3). Tyto položky jsou pak viditelné v prostoru příslušné kompilační jednotky, nejsou však přístupné pod svým jménem mimo tuto kompilační jednotku. Direktivy kompilátoru mají platnost omezenou na kompilační jednotku (direktiva platí jen pro část zdrojového textu uvedenou za ní), což může vést k rozdílům ve výsledcích kompilace při různých způsobech rozdělení zdrojového textu na více kompilačních jednotek.

SystemVerilog tedy dovoluje definovat objekty nejen uvnitř modulů, rozhraní nebo sloh, ale i mimo jejich hranice. Tyto externí definice jsou pak viditelné v prostoru kompilační jednotky, tedy ve všech modulech následně kompilovaných v rámci této jednotky. Při práci na projektu však nemusí být za všech okolností snadno patrné, jaký je rozsah jednotlivých kompilačních jednotek, a pokud je jich více, může se stát, že v některých modulech, jsou-li v různých kompilačních jednotkách, je externí definice viditelná a v jiné ne. Jak jsme viděli, různé programové nástroje mohou rozdělení celého projektu na kompilační jednotky definovat různým způsobem. Dalším problémem může být skutečnost, že i v rozsahu kompilační jednotky je definice viditelná jen pro jednotky, které jsou kompilovány až po kompilaci externí definice (objekt musí být nejprve definován, a až pak může být použit). Použití některých druhů nedeklarovaných objektů je však ve Verilogu i v SystemVerilogu přípustné (implicitní deklarace – viz odst. 2.3.2). Bude-li pak taková deklarace viditelná jen v některých modulech, může být například externě deklarovaný signál v některých modulech interpretován jako více signálů vzájemně nezávislých, a kompilátor tuto chybu nehlásí, protože to nenarušuje syntaxi

aplikacích, kde jde především o syntézu do cílových obvodů PLD a FPGA, se s tím setkáme jen výjimečně a nebudeme nyní o takové možnosti uvažovat, zájemce odkazujeme na manuály [1], [6].

SystemVerilog-2012 zavádí možnost definovat uživatelské typy propojení klíčovým slovem `nettype`, přičemž lze definovat název typu, množinu jeho přípustných hodnot a případně i příslušnou rozhodovací funkci. Podrobnosti i s příklady použití při simulaci smíšených analogově/digitálních konstrukcí bez nutnosti použít kosimulační software může čtenář najít v literatuře [6], [28].

Novým typem zavedeným v této verzi jazyka je typ `interconnect`, který umožňuje deklarovat generické propojení nebo generickou bránu, obojí bez specifikace konkrétního datového typu. Propojení nebo bránu tohoto typu lze použít jen pro vytváření netlistu (strukturální styl), kde datový typ propojení či brány je určen skutečnými datovými typy u komponent, které mají být propojeny – ty lze vybrat pomocí konfigurací nebo podmíněného překladu. To významně usnadňuje vytváření složitých parametrizovatelných modelů. Podrobnosti a příklady lze opět najít v literatuře [6], [28].

Proměnné (*variables*) ve Verilogu i v SystemVerilogu mohou, ale nemusí mít význam fyzických objektů v cílovém obvodu. Ve Verilogu je zaveden typ `reg` jako obecný typ proměnné používaný v procedurálním kódu. V SystemVerilogu se místo něj obvykle používá datový typ `logic` – rovněž čtyřhodnotový. O dalších datových typech proměnných budeme pojednávat níže.

Na rozdíl od signálů typu propojení může být hodnota proměnné ve Verilogu (nejčastěji typu `reg`) měněna pouze procedurálním příkazem (v procedurálním kódu; to neplatí pro SystemVerilog, jak uvidíme dále), a tuto hodnotu si proměnná uchovává až do následující změny dalším procedurálním příkazem. To znamená, že **signály vstupních a obousměrných bran ve Verilogu mohou být jen typu propojení**, obvykle datového typu `wire`, nemohou to být proměnné. Typ propojení mohou mít i signály výstupních bran, to ale mohou být ve Verilogu i proměnné typu `reg`, což je nejčastější, jsou-li vytvářeny procedurálním kódem; v SystemVerilogu to pak bývají proměnné typu `logic`. Signál, kterému se přiřazuje hodnota kontinuálním příkazem, tedy **mimo procedurální kód**, bude **ve Verilogu** typu **propojení** (nejčastěji `wire`) – to se týká i signálů propojujících brány ve strukturálním popisu, zatímco signál, jemuž se hodnota přiřazuje **v procedurálním kódu**, bude typu **proměnné** (ve Verilogu obvykle `reg`, v SystemVerilogu `logic` – toto pravidlo pro signály v procedurálním kódu platí i v SystemVerilogu).

V SystemVerilogu je to jednodušší – statickým proměnným tam lze přiřazovat hodnotu i kontinuálními příkazy (mimo procedurální kód). **Signály vstupních bran zde mohou být** (a také nejčastěji bývají) **proměnné**, obvykle datového typu `logic`; to pak implikuje kontinuální přiřazení vstupních hodnot takovým signálům. Jen signály obousměrných bran a signály k takovým branám zevně připojené (obecně signály buzené z více budičů) musí i zde být typu propojení (nejčastěji `wire`, případně `tri`) – propojení mají přiřazenu rozhodovací funkci, která u takto buzených signálů umožňuje stanovit jejich výslednou hodnotu, což není možné u proměnných. Rozdíl mezi propojením a proměnnou je důležitý také pro přiřazení jejich hodnoty v deklaraci, viz odst. 2.5.1.

Kontinuálními přiřazovacími příkazy nelze ve Verilogu ani v SystemVerilogu přiřazovat hodnotu automatickým proměnným – o těch budeme podrobněji mluvit na konci tohoto odstavce, a vnější signály připojené k výstupům modulů nesmějí být automatické proměnné. To je přípustné jen pro statické proměnné (a jen v SystemVerilogu).

Označení typu `reg` u proměnných ve Verilogu je zavádějící – vyvolává mylný dojem, že jde o výstupní signál klopného obvodu (registru). Bylo zavedeno v době, kdy hlavním použitím Verilogu byla simulace (syntéza se začala rozpracovávat později), a jeho smysl byl v rezervaci paměťového místa (registru) v počítači pro tento signál. V SystemVerilogu je pro označení takové proměnné zavedeno klíčové slovo `logic` (zpětná kompatibilita SystemVerilogu však zaručuje, že lze použít i označení `reg`). Jde tedy jen o jiné, výstižnější označení datového typu proměnných, a proměnné s tímto typem v SystemVerilogu jsou rovnocenné proměnným s typem `reg` (v SystemVerilogu může i signál typu propojení mít datový typ `logic`, nemůže však mít datový typ `reg`). Někteří autoři používají v SystemVerilogu klíčové slovo `logic` k označení kombinačních

jiné účely, než bylo uvedeno. Jiné jejich použití v syntéze může někdy vést k chybným výsledkům. To platí i pro zápis literálů bez specifikace šířky, které se mají automaticky převést do formátu bitových vektorů.

Přehled datových typů a jejich vlastností:

Propojení:

`wire`	dekl.	4h	du		ve Verilogu i v SystemVerilogu
`tri`	dekl.	4h	du		ve Verilogu i v SystemVerilogu
`logic`	dekl.	4h	du	dv	jen v SystemVerilogu

Proměnné (ve Verilogu i v SystemVerilogu):

`reg`	dekl.	4h	du	
`integer`	32b	4h	ds	
`time`	64b	4h	u	
`real`				
`shortreal`				
`realtime`				

Proměnné (jen v SystemVerilogu):

`logic`	dekl.	4h	du	dv
`bit`	dekl.	2h	du	
`byte`	8b	2h	ds	
`shortint`	16b	2h	ds	
`int`	32b	2h	ds	
`longint`	64b	2h	ds	

Vysvětlení:

dekl. – bitová šířka dle deklarace, výchozí (*default*) je jednobitová šířka; 2h – dvouhodnotová veličina; 4h – čtyřhodnotová veličina; du – `unsigned`, ds – `signed`, d zde znamená *default* (výchozí), znaménkovost lze předeklarovat (mimo `time`); dv – výchozí je proměnná, lze ale předeklarovat i jako propojení

Poznámka 1: Datový typ `logic` může být deklarován u proměnné i u propojení. Typ `byte` lze použít i pro znaky ASCII.

Poznámka 2: V manuálu SystemVerilogu [6] jsou typy `byte`, `shortint`, `int`, `longint`, `integer` a `time` označovány souhrnně jako *integer_atom_type*, čímž je myšleno, že mají pevnou (nedělitelnou) bitovou šířku, zatímco typy `bit`, `logic` a `reg` tvoří skupinu s označením *integer_vector_type* – tyto typy je možno použít k vytváření vektorů zvolené bitové šířky přidáním specifikace rozsahu do deklarace.

Poznámka 3: Ve Verilogu-95 nebyla interpretace vektorů jako čísel se znaménkem možná. Manipulaci se znaménkovými bity bylo nutno v případě potřeby explicitně popisovat, s čímž se můžeme setkat ve starších zdrojových textech. Moderní syntetizéry však často dokáží texty s vektory typu `signed` lépe optimalizovat.

2.3.6 SystemVerilog: Typy definované uživatelem

SystemVerilog dovoluje uživateli definovat vlastní typy, kterým budeme stručně říkat **uživatelské**. Ty umožňují vytvářet složité modely na vyšší úrovni abstrakce než v tradičním Verilogu, které jsou však stále syntetizovatelné. Základní tvar definice uživatelského typu je:

```
typedef datový_typ identif_typu;
```

```
    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`.

Příklady interpretace číselných literálů ve výrazu. Jak jsme viděli výše, číslo může být zapsáno třemi způsoby:

- jako celé číslo bez specifikace bitové šířky a základu, například `15`;
- jako celé číslo bez specifikace bitové šířky a se specifikací základu, např. `'d15`, `'sd15`;
- jako celé číslo se specifikací bitové šířky i základu, například `16'd15`, `16'sd15`.

Hodnota záporného čísla zapsaného bez specifikace základu (například `-12`) může být interpretována odlišně od interpretace hodnoty takového čísla zapsaného se specifikací základu (například `-'d12`): číslo bez specifikovaného základu je interpretováno jako číslo typu **`signed`** s bitovým vzorem odpovídajícím dvojkovému doplňku, zatímco číslo se specifikací základu bez písmene `s` před touto specifikací je interpretováno jako typ **`unsigned`**.

V následujícím příkladu jsou uvedeny čtyři možnosti zápisu výrazu s významem „minus 12 děleno 3“. Povšimněte si, že zápisy `-12` a `-'d12` dávají při vyhodnocení stejný bitový vzor, ale ve výrazu zápis `-'d12` nemá charakter záporného čísla, nýbrž je typu **`unsigned`**:

```
module PokusSignedDiv                          // V,SV          // TXT2.4
         (output wire signed [31:0] IntA,IntB,IntC,IntD);
  assign IntA = -12 / 3;          // Vysledek je -4
  assign IntB = -'d12 / 3;        // Vysledek je 1431655761
  assign IntC = -'sd12 / 3;       // Vysledek je -4
  assign IntD = -4'sd12 / 3;      // Vysledek je 1
endmodule
```

V posledním řádku by zápis čísla `-4'sd12` vyžadoval pro korektní zobrazení záporné hodnoty -12 nejméně 5 bitů – bitový vzor `10100`. K dispozici jsou však jen 4 bity, takže dojde k odříznutí nejlevějšího, zde tedy znaménkového bitu, čímž se ztrácí charakter čísla jako záporného – zápis ve 4 bitech představuje kladné číslo 4. Uvedený výsledek pak představuje celou část podílu.

Vyhodnocení výrazů s celočíselnými literály nemusí být pro začátečníka (ale někdy i pro zkušeného konstruktéra) snadno pochopitelné a může vést k poněkud neočekávaným výsledkům, zejména u operandů typu **`signed`**, jak jsme viděli v příkladu (TXT2.4). Je proto vhodné před použitím takových výrazů podrobně prostudovat manuál [6], kde je uvedena řada příkladů, a provést důkladnou simulaci. Nejmenší pravděpodobnost chyby bude patrně tehdy, když místo takového výrazu zapíšeme rovnou zamýšlený výsledek jeho vyhodnocení.

Řetězcové literály, stručněji **řetězce** (*string literals*, *strings*) se zapisují jako posloupnosti znaků ohraničené uvozovkami, například `"ABCD"`. Jsou chápány jako celočíselné konstanty typu **`unsigned`** složené z 8bitových kódových kombinací ASCII odpovídajících znakům zapsaným v řetězci. Je-li řetězec příliš dlouhý na to, aby jej bylo možno zapsat na jeden řádek, zapíše se do něj před znak nového řádku zpětné lomítko (při zápisu kódu se objeví na konci řádku); pak jsou oba tyto znaky při zpracování ignorovány. Řetězce mohou být použity jako operandy ve výrazech. Podle manuálu [1] podporují manipulaci s řetězci všechny operátory jazyka Verilog, přičemž se každý řetězec pokládá za jedno číslo. V manuálu [1] se předpokládá, že řetězce budou přiřazeny proměnným typu **`reg`**, přičemž pro přiřazení platí stejná pravidla jako pro jiné způsoby zápisu literálů, včetně pravidel pro doplnění nulami zleva nebo odříznutí přebytečných bitů zleva, pokud se řetězec přiřazuje proměnné s bitovou šířkou, která neodpovídá osminásobku počtu znaků v řetězci.

Standard SystemVerilogu [6] dovoluje přiřazení řetězců všem objektům s integrálním typem, jako jsou například sbalená pole. S výhodou zde lze použít vícerozměrná pole, u nichž mají podpole (odst. 2.3.11.4) daná nejrychleji se měnícím sbaleným rozměrem šířku 8 bitů, takže každé toto podpole představuje jeden znak řetězce, nebo také pole objektů datového typu **`byte`**.

V SystemVerilogu je dále definován datový typ **`string`** určený především k usnadnění práce s řetězci. Bitová šířka proměnné typu **`string`** se stanoví dynamicky tak, aby proměnná mohla obsáhnout jakýkoliv přiřazovaný řetězec. Při přiřazení zde tedy nedochází k oříznutí ani k doplnění řetězce. Řetězcové literály jsou

dále). Rozsahy uvedené vpravo od těchto názvů odpovídají rozměrům pole, a v SystemVerilogu o nich mluvíme jako o rozsazích nesbalených rozměrů.

Jednorozměrná pole vektorů typu `reg` ve Verilogu se v angličtině označují podle jejich nejčastějšího užití pro modelování pamětí RAM nebo ROM také názvem *memory*. S tímto označením (i pro pole proměnných typu `logic` nebo `bit`) se setkáme také u SystemVerilogu.

Odkaz na prvek jednorozměrného pole se zapíše uvedením identifikátoru pole následovaného indexem prvku v hranatých závorkách, a podobně u vícerozměrných polí.

Pro neplatné indexy platí podobně jako u vektorů: Neplatný je index s hodnotou mimo rozsah odpovídající deklaraci nebo má-li některý z bitů proměnné představující index hodnotu `x` nebo `z`. Čtení z pole s neplatným indexem dává výchozí neinicializovanou hodnotu pro typ čteného prvku (je-li tento prvek skalární, je to `x` u čtyřhodnotových a `0` u dvouhodnotových veličin), zápis do pole s neplatným indexem je ignorován.

U deklarace **vícerozměrných polí** (*multidimensional arrays*) je pro každý rozměr v deklaraci uvedena samostatná položka hranatých závorek s rozsahem vpravo od názvu pole, například pro dvourozměrné pole (stejné ve Verilogu i v SystemVerilogu, až na označení typu – ve Verilogu `reg` místo `logic`):

```
logic [7:0] Mem16_128[1:16][0:127];
```

Závorka vlevo od názvu pole zde znamená, že jde o pole vektorů, jejichž rozsah je uveden v této závorce. Ve Verilogu je přípustný nejvýše jeden rozměr daný závorkou vlevo od názvu, v SystemVerilogu takových rozměrů (sbalených rozměrů pole) může být více.

Pojmy **bitový výběr** a **částečný výběr**, které byly zavedeny u vektorů (odst. 2.3.3), se ve Verilogu i v SystemVerilogu vztahují i na pole. Částečný výběr z vektoru jako prvku pole představuje (podobně jako u vektorů) skupinu sousedních bitů vybraných z tohoto prvku (manuál [6] hovoří poněkud nejasně o částečném výběru jen z jednorozměrného sbaleného pole, tedy z vektoru). Například na vyšší půlslabiku ve vektoru v řádku 15 a sloupci 25 pole `Mem16_128` se odkazuje zápis `Mem16_128[15][25][7:4]` (to odpovídá indexování prvků složených polí v SystemVerilogu naznačenému dále na ***Obr. 2.1***). U polí je tím ve Verilogu i v SystemVerilogu umožněn přístup k jednotlivým bitům nebo k jejich skupinám v takovém vektoru (k bitovému nebo částečnému výběru), ve Verilogu však ne současně k více prvkům (skalárům, vektorům, celočíselným prvkům). V SystemVerilogu je zaveden pojem **řezu**, který umožňuje přístup k části pole obsažené v jednom jeho rozměru, tedy i k více jeho prvkům (o tom více v odst. 2.3.11.3). Verilog nedovoluje kopírovat pole jako celek, SystemVerilog to umožňuje, jak uvidíme dále. O odkazech na části pole (prvky, řezy) v SystemVerilogu bude podrobněji psáno v odst. 2.3.11.4.

Ve Verilogu nemůže být pole deklarováno jako brána modulu (branou zde však může být vektor), v SystemVerilogu je to dovoleno.

2.3.11.1 SystemVerilog: Sbalená a nesbalená pole

V SystemVerilogu se pole deklarovaná výše uvedeným způsobem, který odpovídá polím zavedeným ve Verilogu (s rozsahy uvedenými v deklaraci vpravo od identifikátoru pole), označují za **nesbalená pole** (*unpacked arrays*). U takových polí jsou prvky pole označeny společným názvem, mohou však být umístěny

Pole prvků s celočíselnými typy, které mají předdefinovanou bitovou šířku, tedy s typy **shortint**, **int**, **longint**, **byte**, **integer** a **time**, nemohou mít deklarovány sbalené rozměry. I když takový objekt s předdefinovanou bitovou šířkou `n` není sbaleným polem, je shodný se sbaleným polem s jedním rozměrem o rozsahu `[n-1:0]`. Zápisem:

```
byte DataByte;  // jen SV
integer DataInt;
```

jsou deklarovány proměnné shodného typu (viz odst. 2.3.4) jako následující deklarací:

```
bit signed [7:0] DataByte;
logic signed [31:0] DataInt;
```

U polí deklarovaných druhým zápisem, tj. u polí s typem **bit signed** a podobně, je již možno v SystemVerilogu deklarovat další sbalené rozměry.

Ve standardu SystemVerilogu je také zaveden pojem **typu jednoduchého bitového vektoru** (*simple bit vector type*), který odpovídá jednorozměrnému sbalenému poli bitů. To jsou například vektory tradičního Verilogu nebo objekty celočíselných datových typů s předdefinovanou bitovou šířkou (**integer**, **shortint**, **int**, **longint**, **byte**). Sbalené struktury a vícerozměrná sbalená pole nejsou typu jednoduchého bitového vektoru, jsou však ekvivalentní (viz odst. 2.3.4) některému objektu typu jednoduchého bitového vektoru, na který mohou být snadno převedeny; snadný je i zpětný převod (odst. 2.3.11.10).

Prvky polí v SystemVerilogu mohou být také sbalené nebo nesbalené struktury a uniony. Má-li pak být pole sbalené, mohou být jeho prvky jen sbalené struktury a uniony. Aby byla taková pole syntetizovatelná, musí mít struktury a uniony, které jsou jeho prvky, pojmenovaný typ (definovaný užitím klíčového slova **typedef**, odst. 3.1.1), a uniony musí být sbalené.

Jak jsme již viděli, Verilog dovoluje v poli jeden (nejvýše jeden) sbalený rozměr. V SystemVerilogu může být sbalených rozměrů více. V této souvislosti se někdy mluví o **složeném poli** (pole polí, *array of arrays*), čímž se obvykle rozumí, že jde o nesbalené pole prvků, které mají několik (jeden nebo více) sbalených rozměrů. Sbalené rozměry mohou být nahrazeny jedním rozměrem, takže jde pak o pole vektorů, a definují v takovém vektoru podskupiny. To může být užitečné například při sériovém přenosu dat, kde vektorem bude paket a podskupiny mohou představovat typ paketu, data, kód CRC a podobně.

Sbalená pole jsou užitečná zejména při modelování vektorů vytvořených z prvků bitových datových typů, jako je to u Verilogu, a tam, kde se pracuje samostatně s dílčími částmi polí jako s celky – například při zpracování dat přenášených sériovými přenosovými kanály, jak můžeme vidět v následující definici struktury a deklaraci pole:

```
typedef struct packed {  // definice typu sbalene struktury - jen SV
  logic  [7:0] Header;
  logic [31:0] Data;
  logic  [3:0] Trailer;
} Packet_t;

Packet_t [15:0] PacketArr;  // sbalene pole 16 prvku typu Packet_t
```

Nejprve je zde definován typ sbalené struktury `Packet_t` (odst. 3.1.1), a pak je deklarováno (vloženo) sbalené pole `PacketArr` s 16 prvky tohoto typu. Vytvoření skupin – sbalených polí o různé bitové šířce (`Header`, `Data`, `Trailer`) v prvcích pole je umožněno jejich definováním jako prvků struktury.

Jiný příklad přiřazení v SystemVerilogu – přiřazení hodnot prvkům nesbaleného pole typu **byte** procedurálním kódem (o procedurálním kódu a o konstruktu **always_ff** budeme podrobněji mluvit v odst. 2.6.2):

```
byte UnpckByteArr [3:0][1:0]; // deklarace pole UnpckByteArr - jen SV
always_ff @(posedge Clk) begin                                  // TXT2.8
  UnpckByteArr <= '{default:8'hA}; // prirazeni celemu poli
  UnpckByteArr[3] <= '{'h7,'h6};   // prirazeni rezu
  UnpckByteArr[2][1] <= 8'h5;      // prirazeni jednomu prvku
end
```

V tomto případě získají osmibitové prvky pole (typu **byte**) hodnotu `8'hA` s výjimkou prvků obsažených v řezu `UnpckByteArr[3]` a prvku `UnpckByteArr[2][1]`, jejichž hodnoty budou odpovídat hodnotám jim v bloku **always_ff** explicitně přiřazeným, jak to vyplývá z pravidel pro bloky skupiny **always** (odst. 2.6.2).

Vyznačení korespondence položek v přiřazovacím vzoru s odpovídajícími prvky pole klíčem: V dosud uvedených příkladech byla tato korespondence dána pozicí položek v seznamu. Korespondence však může být vyznačena také **klíčem** (*key*) uvedeným u každé položky, který představuje index prvku pole nebo jeho typ; o klíči se mluví i v souvislosti se slovem **default**. V jednom přiřazovacím vzoru je možno klíče kombinovat. Obdobné ustanovení platí i pro struktury, o čemž budeme mluvit později (odst. 3.1.2). Pravidla pro korespondenci mezi položkami v seznamu přiřazovacího vzoru a prvky pole jsou:

- Zápis *index* : *hodnota* – specifikuje se zde explicitní hodnota indexu korespondujícího prvku. Přiřazovaná hodnota se vyčíslí v kontextu přiřazení indexovanému prvku, její typ musí být kompatibilní pro přiřazení tomuto prvku. Kompatibilní typ je takový typ, na který může být původní typ prvku převeden (*cast*) – viz odst. 2.3.4 a odst. 2.3.8.
- Zápis *typ* : *hodnota* – prvky nebo podpole s typem odpovídajícím tomuto typu, pokud jim nebyla přiřazena hodnota předchozím způsobem (podle indexu), získají hodnotu uvedenou v tomto zápisu. Opět musí být hodnota přiřaditelná prvku, kterému se přiřazuje.
- Zápis **default** : *hodnota* – toto přiřazení se vztahuje na prvky, které nejsou zahrnuty v předchozích bodech. Pro vyčíslení hodnoty platí totéž, co bylo řečeno v prvním bodu.

Jedním z těchto bodů musí být pokryt každý z prvků pole, jemuž se přiřazuje hodnota. Na pozici položek v seznamu přiřazovacího vzoru v tomto případě nezáleží.

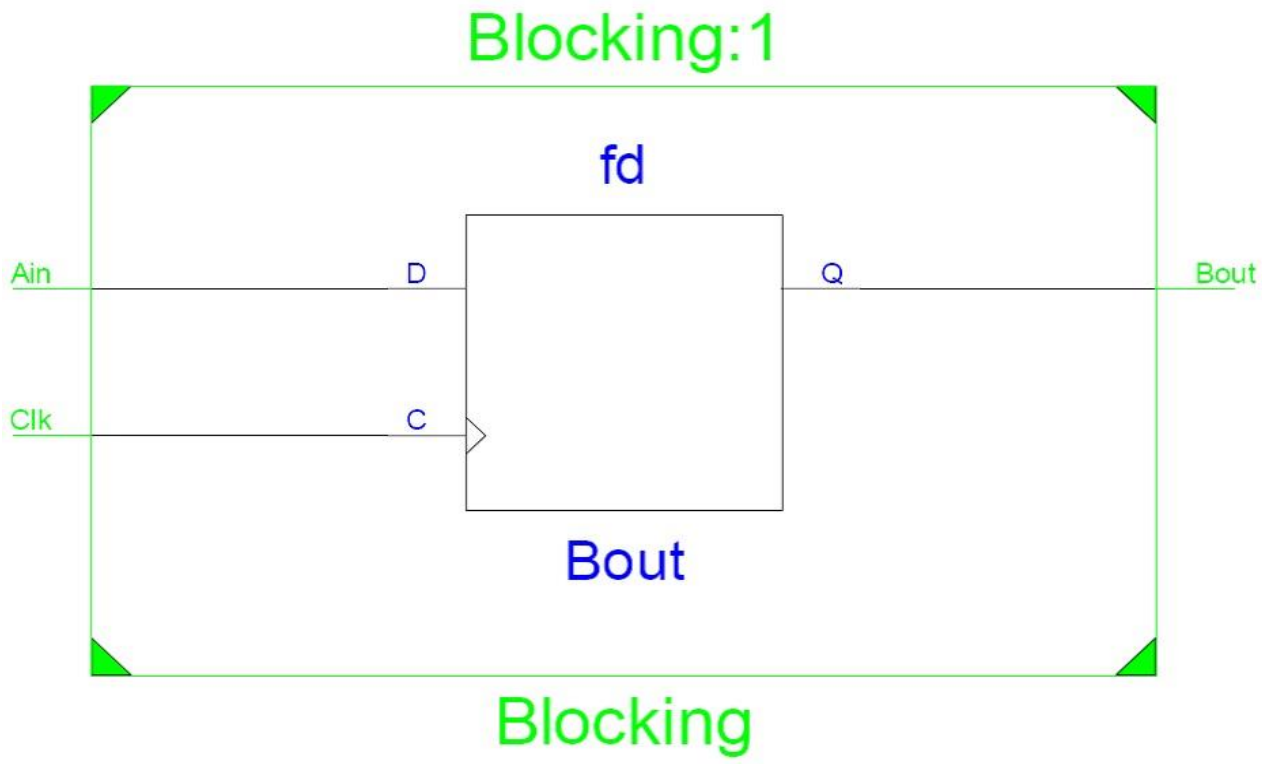
Jak jsme viděli, forma zápisu přiřazovacích vzorů připomíná formu zápisu sjednocení, rozdíl je jen v úvodním apostrofu u přiřazovacích vzorů. Nicméně význam zápisu sjednocení může být výrazně odlišný od významu zápisu přiřazovacího vzoru. Příklad takové odlišnosti:

```
logic [1:0][3:0] Arr1,Arr2; // deklarace pole Arr1,Arr2 - jen SV
assign Arr1 =  {1'b1,1'b1}; // prirazeni sjednoceni poli Arr1
assign Arr2 = '{1'b1,1'b1}; // prirazeni seznamu hodnot poli Arr2
```

Jde o sbalená pole. Pole `Arr1` je při přiřazování chápáno jako vektor, a jeho prvkům tak budou přiřazeny tyto hodnoty: prvku `Arr1[1]` bude přiřazena hodnota `4'b0000`, prvku `Arr1[0]` bude přiřazena hodnota `4'b0011`. Oba prvky pole `Arr2` však budou mít po přiřazení hodnotu `4'b0001`.

Další odlišností zápisu hodnot v seznamu u přiřazovacího vzoru od pravidel pro zápis sjednocení a replikací je, že při zápisu sjednocení a replikací musí být konstanty zapsány včetně uvedení bitové šířky. To neplatí u zápisu seznamu přiřazovacího vzoru, kde se v knize [12] dokonce doporučuje bitovou šířku neuvádět, aby se tím zápis seznamu odlišil od zápisu sjednocení a replikací – tak se zajistí, že zápis nebude chápán jako sjednocení, které je jinak formálně velmi podobné. Seznam u přiřazovacího vzoru může obsahovat také reálné hodnoty.

Syntézou textu TXT2.12 v systému ISE a zobrazením *View RTL Schematic* získáme následující schéma (***Obr. 2.2***):



Obr. 2.2. *Výsledek zobrazení RTL Schematic u textu* TXT2.12

V druhém textu, jinak shodném, použijeme neblokující přiřazení:

```
module Nonblocking(input Clk,Ain,      // V                    // TXT2.13
                   output reg Bout);
    reg Tmp;
  always @(posedge Clk) begin
    Tmp <= Ain;
    Bout <= Tmp;
  end
endmodule
```

Schéma získané syntézou textu TXT2.13 v tomtéž systému a zobrazením *View RTL Schematic* je na ***Obr. 2.3***.

V záhlaví smyčky dovoluje SystemVerilog deklarovat i více řídících proměnných. Deklarace se pak oddělují čárkami. Následující deklarace pak může využívat již deklarované proměnné. Rovněž příkazů pro příští hodnotu může být více. Příklad:

```
for (int i=0, j=2, k=i*2; i<8; i++, j+=2)
  ...
```

Příklad smyčky **for** zapsané v syntaxi Verilogu:

```
initial begin: Test                // V,SV
  integer i;
  for (i=0; i<=255; i=i+1) begin
BinIn = i;
#10; //Wait 10 ns
  end
  #10; $stop;
end
```

Takové smyčky se často používají ve zkušebních jednotkách (benčích). Předpokládá se, že proměnná `BinIn` datového typu **reg**, která představuje vstupní signál testované konstrukce, byla deklarována dříve. Zápis této smyčky v syntaxi SystemVerilogu bude:

```
initial begin: Test                // V*,SV
  for (int i=0; i<=255; i++) begin
BinIn = i;
#10; //Wait 10 ns
  end
  #10; $stop;
end
```

Další příklady smyček **for** budou uvedeny v kapitole 5.

Smyčka **while** má následující syntaxi:

```
while (výraz) procedurální_příkaz_nebo_blok_begin-end
```

Příkaz nebo blok **begin-end** ve smyčce **while** se opakovaně provádí, dokud je *výraz* nenulový (pravdivý). Pravdivost se testuje před provedením příkazů následujících ve smyčce. Je-li *výraz* nepravdivý na počátku, nejsou příkazy vůbec provedeny. Aby byla smyčka syntetizovatelná, musí být možné stanovit počet jejího opakování při kompilaci; bývají i další omezení.

V SystemVerilogu je dále definována **smyčka** **do-while**. Její syntaxe je:

```
do procedurální_příkaz_nebo_blok_begin-end
while (výraz);
```

Od smyčky **while** se liší tím, že se nenulovost (pravdivost) výrazu testuje až po provedení příkazů ve smyčce, takže tyto příkazy jsou provedeny nejméně jednou.

Smyčky **repeat** a **forever** se používají nejčastěji pro generování stimulačních signálů ve zkušebních jednotkách. Smyčka **repeat** může být omezeně použita i v syntéze, jak naznačuje příklad modelu násobičky uvedený v manuálu [1], [6]. Syntaxe této smyčky je:

```
repeat (výraz) procedurální_příkaz_nebo_blok_begin-end
```

Příkaz nebo blok **begin-end** uvedený ve smyčce **repeat** se provádí v počtu určeném výrazem v závorce. Smyčka má stejný účinek jako příkaz nebo blok **begin-end** zapsaný opakovaně za sebou v tomto počtu.

nenapíše nic (závorka se ponechá prázdná), nebo může být položka v seznamu se jménem nepřipojené brány vynechána. Je-li v SystemVerilogu zapsáno přidružení formou *dot-name* (odst. 2.8.6), je třeba za jménem brány bez připojení napsat v seznamu přidružení prázdné závorky.

Jsou-li v definici komponenty brány s bitovou šířkou odlišnou od bitové šířky příslušných signálů ve vložení komponenty, je přidružení provedeno zarovnáním vpravo (*right justification*), případně odříznutím (*truncation*) přebytečných bitů zleva. Syntetizéry na to obvykle reagují varováním.

Příkazem vložení je možno vložit i celé pole komponent uvedením jeho rozsahu za identifikátorem vložení. Příklad vložení třístavových budičů (primitiva `bufif0`) jako pole komponent:

```
bufif0 ThreeSt[3:0](Out,In,En);
```

Signály `Out`, `In` jsou zde čtyřbitové vektory (jejich bitová šířka je dána rozsahem v příkazu vložení), signál `En` je skalární signál ovládající stav všech čtyř budičů. Se stejnou syntaxí se vkládají pole komponent představujících moduly. Další podrobnosti i s příklady najde zájemce v manuálech [1], [6].

Obsahuje-li definice komponenty parametry, můžeme hodnoty těchto parametrů při jejím vložení **redefinovat** – zvolit jejich hodnoty odlišně od hodnot uvedených v definici komponenty, jak jsme to již naznačili v odst. 2.3.10. Jako příklad uvažujme multiplexor, u něhož bude bitová šířka dána parametrem:

```
module Mx4to1 #(parameter Wdth = 1)        // V*,SV          // TXT2.31
               (input logic [Wdth-1:0] Di0,Di1,Di2,Di3,
                input logic [1:0] Sel,
                output logic [Wdth-1:0] MxOut);
  always_comb
    case (Sel)
      2'b00: MxOut = Di0;
      2'b01: MxOut = Di1;
      2'b10: MxOut = Di2;
      2'b11: MxOut = Di3;
    endcase
endmodule
```

Připomeňme, že čísla na pozici volby v příkazu `case` je nutno zapisovat ve formátu se specifikovanou bitovou šířkou, jak je zdůvodněno v poznámce v odst. 2.6.5.2.

Tento multiplexor můžeme jako komponentu vložit do hierarchicky vyššího modulu, v němž `W`, `X`, `Y`, `Z`, `R` jsou osmibitové vektory a `S` je dvoubitový vektor, příkazem:

```
Mux4to1 #(8) G (.MxOut(R),.Di0(W),.Di1(X),.Di2(Y),.Di3(Z),.Sel(S));
```

Hodnota parametru `Wdth` uvedená v definici je zde nahrazena hodnotou `8` z příkazu vložení, `G` je identifikátor vložení.

Pokud je parametrů více, provádí se přiřazení podle pořadí jejich zápisu, hodnoty parametrů v závorce se oddělují čárkami. Je-li ve vkládané komponentě deklarováno více parametrů, než je počet redefinujících hodnot v závorce ve vložení, jsou redefinovány jen odpovídající parametry, které jsou první v pořadí. Tento

```
            end
      end
    assign RamIf.Data = DataVar;
endmodule

module Ram (Intf1_if.Ram_mp RamIf); // rozhrani jako brana s nazvem RamIf
  logic [7:0] MemArr [15:0];
  logic [7:0] DataVar;
  always @(posedge RamIf.Clk, posedge RamIf.Rst)
    if (RamIf.Rst) begin
      DataVar <= 'Z;
      for (int i=0; i<=15; i++) MemArr[i] <= '0;
    end
    else begin
      DataVar <= 'Z;
      if (RamIf.WrEn) MemArr[RamIf.Addr] <= RamIf.Data;
      else DataVar <= MemArr[RamIf.Addr];
    end
  assign RamIf.Data = DataVar;
endmodule

module Top (input Clk,Rst,
            input Instr_t Instr_st, // struktura Instr_st jako brana
                                    // typu Instr_t
            output [7:0] DaOut);
  Intf1_if TopIf (Clk,Rst);  // vlozeni rozhrani, obsahuje Data,Addr,WrEn
  ControllerIF Inst1(.RamIf(TopIf),.*);
//            .Instr_st(Instr_st), // tato dve pridruzeni jsou nahrazena
//            .DaOut(DaOut));      // zapisem .* na radku vyse
  Ram Inst2(.RamIf(TopIf));
endmodule
```

Ve sloze `Def` je definován typ `Instr_t`. Sloha je pak importována do kompilační jednotky, takže její obsah je viditelný ve všech následujících modulech. V definici rozhraní `Intf1_if` jsou v seznamu bran vyznačeny jen brány `Clk`, `Rst` se směrem **`input`**. Signály deklarované dále v textu nemají deklarovaný směr; ten je určen pro jednotlivé moduly příslušným příkazem **`modport`**. Všechny tyto signály kromě signálu `Data` nemají explicitně určený typ (**`logic`** je datový typ, nikoli typ signálu). Podle pravidel uvedených v odst. 2.3.1 jsou tedy signály bran (`Clk`, `Rst`) typu propojení (**`wire`**), tento typ musí mít i obousměrný signál `Data`, a signály `Addr` a `WrEn` jsou proměnné.

V deklaraci bran modulu `ControllerIF` je jako brána `RamIf` použito rozhraní `Intf1_if`. Směr signálů této brány je určen prvním příkazem **`modport`** obsaženým v definici rozhraní. Dále je zde na základě typu `Instr_t` deklarována struktura jako vstupní brána modulu s názvem `Instr_st`.

V následujícím modulu `Ram` je opět použito rozhraní `Intf1_if` jako brána s názvem `RamIf`, rovněž zde je směr signálů určen příkazem **`modport`**. Je zřejmé, že použití rozhraní v obou těchto modulech představuje úsporu v zápisu identifikátorů bran, které by se jinak musely opakovaně vypisovat. Úspora by byla ještě větší, pokud by takových modulů bylo v konstrukci více, což bývá časté u sběrnicových systémů.

Do vrcholového modulu `Top` je vloženo rozhraní `Intf1_if` jako vnitřní signál s identifikátorem `TopIf`. Tento identifikátor může být libovolný, a byl zvolen tak, aby odpovídal funkci signálu. Není však žádný zvláštní důvod, proč by zde nemohl být použit jiný identifikátor, například `RamIf`. To by však mohlo být poněkud matoucí, protože jde o signál poněkud jiného druhu, než jsou signály bran obou modulů, které mají

a výstupních signálů. Uvedeme nejprve popis v SystemVerilogu, kde deklarace vstupního signálu může být výhodně zapsána ve tvaru dvourozměrného vektoru, přičemž první rozměr udává počet dekadických číslic výsledku a druhý vyjadřuje počet bitů v dekádě:

```
module BCD2BinMultSV(input logic [2:0][3:0] BCDin, //jen SV // TXT5.1.2a
                     output logic [7:0] BinOut);
  always_comb begin
    BinOut = BCDin[0];
    BinOut = BinOut + BCDin[1] * 10;
    BinOut = BinOut + BCDin[2] * 100;
  end
endmodule
```

U popisu ve Verilogu syntaxe nedovoluje, aby signál brány byl vícerozměrný, a nemůže to být ani pole. Ve Verilogu (a samozřejmě i v SystemVerilogu) je možno například deklarovat vstupní signál `BCDin` jako jednorozměrný vektor s bitovou šířkou rovnou čtyřnásobku počtu dekadických číslic tohoto signálu:

```
module BCD2BinMult(input [11:0] BCDin,        // V         // TXT5.1.2b
                   output reg [7:0] BinOut);
  always @* begin
    BinOut = BCDin[3:0];
    BinOut = BinOut + BCDin[7:4] * 10;
    BinOut = BinOut + BCDin[11:8] * 100;
  end
endmodule
```

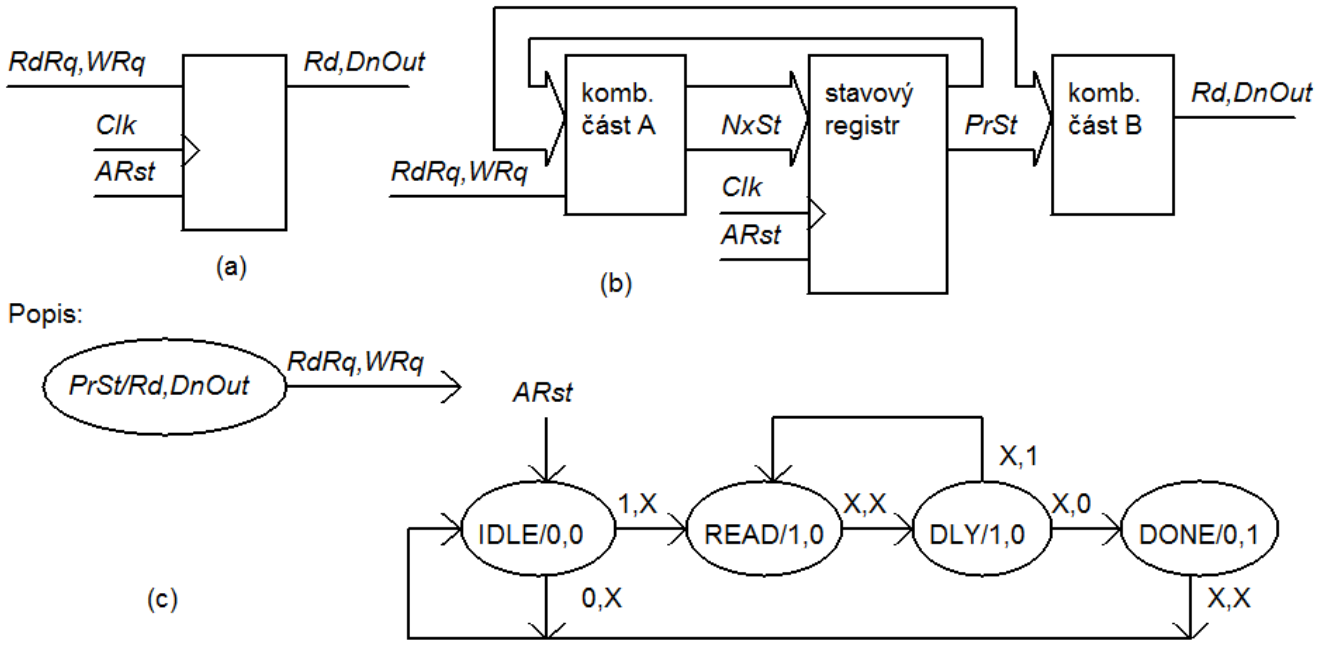
Implementací do řady XC9500XL v systému ISE zjistíme, že převodník lze vložit do obvodu XC95144XL, přičemž je využito 55 % makrobuněk, 60 % součinových termů a 45 % vstupů funkčních bloků z cílového obvodu. Zpoždění nejdelší cesty zjištěné statickou časovou analýzou je 15,1 ns.

Odpovídající benč ve Verilogu (akceptovatelný i v SystemVerilogu) vytvořený z šablony generované systémem ISE (*Project => New Source... => Verilog Test Fixture*) doplněním stimulačních signálů je:

```
module BCD2Bin_TF;                        // V              // TXT5.1.2c
  reg [11:0] BCDin;          // vstup uut
  wire [7:0] BinOut;         // vystup uut
  BCD2BinMult uut (.BCDin(BCDin),.BinOut(BinOut)); // Unit Under Test
  initial begin: TB
    integer i,j,k;
// Add stimuli here
    for (k=0; k<=2; k=k+1) begin: kLoop
      for (j=0; j<=9; j=j+1) begin: jLoop
        for (i=0; i<=9; i=i+1) begin: iLoop
          BCDin[11:8] = k;
          BCDin[7:4]  = j;
          BCDin[3:0]  = i;
          if (BCDin >= 12'h256) disable iLoop;
          if (BCDin >= 12'h256) disable jLoop;
          if (BCDin >= 12'h256) disable kLoop;
          #10; // cekej 10 ns
        end
      end
    end
        $stop;
  end
endmodule
```

Příkazy **`disable`** ukončí provádění bloků **`begin-end`** ve smyčkách a tedy i simulaci, když signál `BCDin` dosáhne hodnoty 255. Bez těchto příkazů by simulace pokračovala až do hodnoty 299 tohoto signálu.

signál `Rd` (Read). Je-li ve stavu `DLY` aktivní vstupní signál `WRq` (Wait Request), vrací se obvod do stavu `READ`. Při neaktivním signálu `WRq` přechází do stavu `DONE`, přičemž aktivuje po dobu jednoho taktu výstupní signál `DnOut`, načež se vrací do klidového stavu `IDLE`.



Obr. 5.14. *Blokový symbol (a), blokové schéma (b) a stavový diagram (c) řídícího obvodu čtení Hodnota* `X` *vstupního signálu znamená, že na ní nezáleží*

Automat sestavíme v Moorově verzi. Popis ve Verilogu bude:

```
module RdCtrl (input Clk,RdRq,WRq,ARst,           // V     // TXT5.5.3a
               output reg Rd,DnOut);
   reg [1:0] PrSt,NxSt; // stav.reg. - soucasny (Pr) a pristi (Nx) stav
   parameter IDLE=2'b00, READ=2'b01, DLY=2'b11, DONE=2'b10;
   always @(PrSt,RdRq,WRq) begin
     NxSt = 'bx;
     Rd = 1'b0; DnOut = 1'b0;
     case (PrSt)
       IDLE: if (RdRq)   NxSt = READ;
             else        NxSt = IDLE;
       READ: begin       NxSt = DLY;  Rd = 1'b1; end
       DLY:  begin                    Rd = 1'b1;
               if (!WRq) NxSt = DONE;
               else      NxSt = READ;
             end
       DONE: begin       NxSt = IDLE; DnOut = 1'b1; end
     endcase
   end
   always @(posedge Clk, posedge ARst)
     if (ARst) PrSt <= IDLE;
     else      PrSt <= NxSt;
endmodule
```

Popis v SystemVerilogu:

```
module RdCtrlSV (input logic Clk, RdRq,WRq,ARst, // jen SV  // TXT5.5.3b
                 output logic Rd,DnOut);
```