Jazyky Verilog a SystemVerilog

a jejich užití při modelování a syntéze číslicových systémů

PŘÍRUČKA ZAČÍNAJÍCÍHO UŽIVATELE

Jaromír Kolouch

Jazyky Verilog a SystemVerilog

a jejich užití

při modelování a syntéze číslicových systémů

Příručka začínajícího uživatele

Jaromír Kolouch

Lektorovali:
doc. Ing. Martin Poupa, Ph.D.
Ing. Marcela Šimková

ISBN 978-80-214-5216-9

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
NAKLADATELSTVÍ VUTIUM

Obsah

Zkratky

ASIC	Application Specific Integrated Circuits
CAD	Computer Aided Design
CPLD	Complex Programmable Logic Devices
DPI	Direct Programming Interface
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
IP	Intellectual Property
LUT	Look-up Table
PAL	Programmable Array Logic
PLA	Programmable Logic Array
PLD	Programmable Logic Devices
RTL	Register Transfer Level
SOP	Sum of Products

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 – je klíčovým slovem s významem blízkým počítačovému programu, s použitím hlavně při verifikaci.) Dále budeme termínem „objekt“ rozumět datové objekty, nikoliv instance tříd, jak je to obvyklé u objektového programování, které SystemVerilog ve své nesyntetizovatelné části podporuje.

Obvykle se v literatuře o Verilogu a o SystemVerilogu také nerozlišuje mezi definicí a deklarací, zejména ve vztahu k funkcím, úlohám, uživatelsky definovaným typům a k rozhraním. Tyto pojmy se často používají záměnně. V této knize se budeme snažit o rozlišení těchto pojmů.

Termíny používané v dalším textu a vztahující se k práci s návrhovým systémem odpovídají terminologii zavedené v systému ISE firmy Xilinx, v případě potřeby doplněné o terminologii používanou v systému Quartus firmy Altera. U jiných návrhových systémů mohou být (a také bývají) odchylky.

Autor bude vděčný laskavým čtenářům za upozornění na chyby a nepřesnosti, které najdou v tomto textu. Zašlete je prosím na adresu:

kolouch@feec.vutbr.cz

Opravy chyb, které budou nalezeny v této knize (autor věří, že jich nebude mnoho), budou dostupné na stránce:

http://www.urel.feec.vutbr.cz/~SystemVerilog/

1.1 Klíčová slova jazyků Verilog a SystemVerilog

KLÍČOVÁ SLOVA jazyka *Verilog-95* s doplněním o klíčová slova verze *Verilog-2001* a *Verilog-2005* (kurziva), [1]:

always	**endmodule**	***liblist***	**rcmos**	**tranif1**
and	**endprimitive**	***library***	**real**	**tri**
assign	**endspecify**	***localparam***	**realtime**	**tri0**
automatic	**endtable**	**macromodule**	**reg**	**tri1**
begin	**endtask**	**medium**	**release**	**triand**
buf	**event**	**module**	**repeat**	**trior**
bufif0	**for**	**nand**	**rnmos**	**trireg**
bufif1	**force**	**negedge**	**rpmos**	***unsigned***
case	**forever**	**nmos**	**rtran**	***use***
casex	**fork**	**nor**	**rtranif0**	***uwire *)***
casez	**function**	***noshowcancelled***	**rtranif1**	**vectored**
cell	***generate***	**not**	**scalared**	**wait**
cmos	***genvar***	**notif0**	***showcancelled***	**wand**
config	**highz0**	**notif1**	***signed***	**weak0**
deassign	**highz1**	**or**	**small**	**weak1**
default	**if**	**output**	**specify**	**while**
defparam	***ifnone***	**parameter**	**specparam**	**wire**
design	***incdir***	**pmos**	**strong0**	**wor**
disable	***include***	**posedge**	**strong1**	**xnor**
edge	**initial**	**primitive**	**supply0**	**xor**
else	**inout**	**pull0**	**supply1**	
end	**input**	**pull1**	**table**	
endcase	***instance***	**pulldown**	**task**	
endconfig	**integer**	**pullup**	**time**	
endfunction	**join**	***pulsestyle_onevent***	**tran**	
endgenerate	**large**	***pulsestyle_ondetect***	**tranif0**	

***) Poznámka:** Klíčové slovo `uwire` je zavedeno ve verzi Verilog-2005 (jediné nové klíčové slovo zavedené v této verzi).

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

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

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**

1.2 Slovníček termínů užívaných v jazycích Verilog a SystemVerilog

Termíny používané v anglických textech o Verilogu a SystemVerilogu nemají dosud v četných případech všeobecně akceptovaný český překlad. Autor se pokusil doplnit českou terminologii, a pro snadnější orientaci čtenáře je zde připojen malý slovníček těchto termínů. Pro stručnost v něm nejsou obsaženy termíny, jejichž překlad je totožný či téměř totožný s původním slovem, a termíny, jejichž překlad je všeobecně známý a akceptovaný.

array	pole	bitový výběr	bit-select
assignment	přiřazení	brána	port
association	přidružení	citlivostní seznam	sensitivity list
bit-select	bitový výběr	částečně typový	semi-strongly typed
built-in	vestavěný	částečný výběr	part-select
code	kód (zdrojový text)	hlavička	header
concatenation	sjednocení	kód (zdrojový text)	code
concurrent statement	souběžný příkaz	nesbalený	unpacked
connection	připojení	odskočený identifikátor	escaped identifier
context-determined	s kontextovým určením	ořízntutí (zkrácení)	truncation
enumerated type	výčtový typ	pole	array
escaped identifier	odskočený identifikátor	proměnná	variable
expression	výraz	propojení (typ signálu)	net
full	úplný (příkaz `case`)	prostor (kompilační jednotky)	scope
handle	úchyt	prostor jména	space of a name
header	hlavička	přenositelnost	portability
inferencing	rozpoznání	převod typu	type casting
instantiation	vložení	přidružení	association
interface	rozhraní	příkaz	statement
justification	zarovnání	připojení	connection
layout (e.g., of an array)	uspořádání (např. pole)	přiřazení	assignment
loop	smyčka	rozhraní	interface
nested module	vnořený modul	rozpoznání	inferencing
net	propojení (typ signálu)	rozsah (v deklaraci)	range
package	sloha	rozsah působnosti	scope
packed	sbalený	rozšíření (doplnění)	padding
padding	rozšíření (doplnění)	s kontextovým určením	context-determined
part-select	částečný výběr	s vlastním určením	self-determined
port	brána	sbalený	packed
portability	přenositelnost	sjednocení	concatenation
range	rozsah (v deklaraci)	sloha	package
scope	rozsah působnosti, prostor (kompilační jednotky)	smyčka	loop
		souběžný příkaz	concurrent statement
self-determined	s vlastním určením	šablona	template
semi-strongly typed	částečně typový	úchyt	handle
sensitivity list	citlivostní seznam	úloha	task
side effect	vedlejší účinek	úplný (příkaz `case`)	full
source text	zdrojový text	uspořádání (např. pole)	layout (e.g., of an array)
space of a name	prostor jména	vedlejší účinek	side effect
statement	příkaz	vestavěný	built-in
task	úloha	vložení	instantiation
template	šablona	vnořený modul	nested module
truncation	ořízntutí (zkrácení)	výčtový typ	enumerated type
type casting	převod typu	výraz	expression
unpacked	nesbalený	zarovnání	justification
variable	proměnná	zdrojový text	source text

2 Syntaxe jazyků Verilog a SystemVerilog a modelování číslicových systémů

2.1 Úvodní poznámky

Jazyky Verilog a SystemVerilog patří do skupiny jazyků HDL (*Hardware Description Language*), které se používají pro návrh aplikací obvodů PLD a FPGA. Dalším jazykem tohoto druhu, v současnosti zhruba stejně rozšířeným, je jazyk VHDL. Jak jazyk VHDL, tak i Verilog (na který navazuje SystemVerilog) byly původně vyvinuty pro účely simulace a dokumentace, nikoliv pro syntézu. Na mnoha jejich konstruktech je to znát, mnohé z nich nemají pro syntézu vůbec význam. Zde se budeme zabývat především použitím jazyků Verilog a SystemVerilog k vytváření modelů určených pro syntézu číslicových systémů; jak již bylo řečeno v úvodu, většinu nesyntetizovatelných konstruktů zde nebudeme diskutovat. Simulaci budeme využívat hlavně pro ověření správnosti funkce syntetizované konstrukce. Lze ji užít i pro zjištění časových parametrů, k čemuž v jazycích HDL slouží speciální jazykové konstrukty, zde však budeme k tomuto účelu předpokládat především použití statické časové analýzy.

České termíny budou v prvním výskytu zapsány **tučně**. Často tyto termíny nejsou ustálené, a proto budeme uvádět i jejich anglické ekvivalenty, které jsou většinou používány standardně. Bude-li to pro orientaci v textu účelné, mohou tak být vyznačeny termíny i na více místech (opakovaně). Anglické termíny a často užívané fráze jsou v textu uváděny *kurzivou* (zpravidla v závorce). Tučně a modrou barvou jsou také psána klíčová slova. Slova podobného charakteru, která však nejsou klíčovými slovy ve smyslu jazyků Verilog a SystemVerilog (jako jsou například direktivy nebo názvy systémových funkcí), budou psána fontem odpovídajícím zápisu zdrojových textů, avšak kurzivou a nikoliv tučně.

Snahou autora bylo podchytit v textu jak to, co se týká Verilogu, tak i to, co je specifické pro SystemVerilog, a to bez opakování, které by bylo nutné, pokud by se měly popsat konstrukty platné ve Verilogu i v SystemVerilogu samostatně. V některých případech by tak mohlo být nesnadné poznat, co platí pro Verilog a co jen pro SystemVerilog, a to zejména v **odst. 2.3.1**, a také v **odst. 2.3.9** a v **odst. 2.3.10**. Proto jsou v těchto podkapitolách uvedeny části platné jen v SystemVerilogu hnědou barvou. Jiné kapitoly či podkapitoly, kde tento problém není tak závažný, jsou však psány běžným způsobem (černým písmem).

Předmět popisu konstrukce, ať již textového, grafického či smíšeného, budeme označovat za její **model**. Pro popis modelu zapsaný v jazycích Verilog i SystemVerilog budeme také používat termín **zdrojový text** nebo stručněji **kód**, což odpovídá často užívaným anglickým termínům *source text* (*code*).

Základní vlastnosti jazyků Verilog a SystemVerilog (totéž platí i pro jazyk VHDL):

- Představují **otevřený standard** (*open standard*). K jejich použití pro sestavení návrhových systémů není třeba licence jeho vlastníka, jako je tomu u některých jiných jazyků HDL (například u jazyka ABEL). Asi se nenajde návrhový systém, který by jazyky Verilog a VHDL nepodporoval, a v dohledné době to zřejmě bude platit i pro SystemVerilog.
- Umožňují pracovat na návrhu, aniž je předtím zvolen cílový obvod (*device-independent design*). Ten může být zvolen až v době, kdy jsou známy definitivní požadavky na prostředí, v němž má navrhovaná konstrukce pracovat, a je možno jej ve značné míře měnit podle potřeby při zachování popisu v jazyku HDL. Může být zvolen obvod PLD nebo FPGA, aniž se volba dotkne popisu, pokud ovšem tento obvod obsahuje bloky, které popis předpokládá.
- Je možno provést simulaci navrženého obvodu na základě téhož zdrojového textu, který pak bude použit pro syntézu a implementaci v cílovém obvodu. Zdrojový text je možno zpracovávat v různých simulátorech a v syntetizérech různých výrobců (opět s určitým omezením daným zejména vnitřní strukturou cílových obvodů). Odsimulovaný text může být použit v dalších projektech s různými cílovými obvody, což je umožněno tím, že jazyky Verilog i SystemVerilog podporují hierarchickou strukturu projektů, kde je vytvářený systém složen z dílčích bloků. Tyto bloky mohou představovat typické funkční celky, jako jsou například převodníky kódu, čítače, aritmetické obvody, procesory a podobně, které mohou být

opakovaně využity v dalších projektech a mohou být implementovány do jiných cílových obvodů, případně zpracovávány na jiných návrhových systémech. Této vlastnosti jazyka se říká **přenositelnost** (*portability*) kódu.

- V případě úspěšného zavedení výrobku na trh lze popis konstrukce v jazycích HDL použít jako podklad pro její implementaci do obvodů ASIC vhodných pro velké série.

Základní verze jazyka Verilog byla přijata jako standard IEEE číslo 1364 v roce 1995. Konstrukty odpovídající tomuto standardu se označují jako konstrukty jazyka Verilog-95. Na základě zkušeností s touto verzí byla v roce 2001 přijata nová verze, Verilog-2001, [1]. V této verzi je provedena řada úprav a zdokonalení a podporuje ji většina současných návrhových systémů. Následující – poslední verze Verilogu [2] z roku 2005 obsahuje jen malé korekce, jejichž podpora ve vývojových nástrojích není příliš rozšířená. Pro tyto příručky i pro podobné dokumenty platné pro SystemVerilog se často používá označení LRM (*Language Reference Manual*).

V roce 2002 přijala organizace IEEE standard IEEE Std 1364.1™-2002 [3], který definuje podmnožinu konstruktů jazyka Verilog-2001 vhodnou pro syntézu na úrovni RTL (význam této zkratky je vysvětlen v **odst. 2.1.1**). Tento standard zajišťuje přenositelnost i pro výsledky syntézy u syntetizérů, které jej splňují.

Jazyk SystemVerilog navazuje na Verilog. Byl vyvíjen organizací Accelera s úmyslem doplnit standard Verilogu o nové možnosti potřebné zejména pro konstrukci rozsáhlých systémů. O procesu vývoje tohoto jazyka referují podrobně prameny [26], [27].

V roce 2005 byly organizací IEEE přijaty dva standardy: Verilog-2005 (*IEEE 1364-2005*) a SystemVerilog-2005 (*IEEE 1800-2005*). Tyto standardy byly sjednoceny v roce 2009 pod označením SystemVerilog-2009 (*IEEE 1800-2009*). Jazyk Verilog-2005 zde představuje podmnožinu jazyka SystemVerilog-2009. SystemVerilog je tedy zpětně kompatibilní s Verilogem a **nástroje pro SystemVerilog zpracovávají zdrojové texty zapsané ve Verilogu bez problémů.** Renomovaní autoři, jako je například Stuart Sutherland, tvrdí, že se další vývoj již bude týkat jen jazyka SystemVerilog, viz [26], [27], a že v současnosti všichni, kdo pracují s Verilogem, používají ve skutečnosti SystemVerilog (i když si to mnohdy neuvědomují). Další verze, SystemVerilog-2012, [6], přináší pro syntézu jen nevelké úpravy. Všechny změny provedené ve verzi 2012 jsou přehledně uvedeny v [28].

V manuálech SystemVerilogu [3], [5], [6] není syntetizovatelná podmnožina jazyka vyznačena. Do značné míry nahrazují tento nedostatek články [29], [30] a kniha [11].

V dalším textu budeme označením Verilog myslet verzi Verilog-2001 podle standardu [1], která je v současnosti nejrozšířenější verzí. Pokud bychom měli na mysli jinou verzi, bude to uvedeno jejím explicitním číselným označením. Podobně označení SystemVerilog se bude týkat verze SystemVerilog-2012 [6].

Za základní dokument, který by měl definovat chování nástrojů, jako jsou simulátory, syntetizéry a podobně, se většinou považují manuály popisující standardy IEEE, tedy v současnosti publikace [1], [6] (standard pro Verilog je patrně možno považovat za konečný, u SystemVerilogu dochází přibližně v pětiletých intervalech k inovacím). V některých případech však jejich text není zcela vyčerpávající a srozumitelný, což se týká zejména syntetizovatelnosti, jak již bylo zmíněno. Pro SystemVerilog je pak vhodné doplnit informace z manuálu další literaturou. Pro naše účely je k tomu vhodná zejména kniha [11], jejíž autoři patří ke tvůrcům jazyka. Stává se občas (sice zřídka, ale přece jen), že se manuál a uvedená kniha v některých podrobnostech rozcházejí, viz např. **odst. 3.1** nebo **odst. 3.3.7.1**. V takových případech se autor tohoto textu snažil na to upozornit – může se stát, že chování některých nástrojů bude spíš odpovídat tomu, co je psáno v knize [11] než v manuálu [6], nemusí však tomu tak být vždy. Informace z literatury, zejména z internetu, je však třeba přebírat s opatrností, nezřídka se tam vyskytují nepřesnosti či dokonce chyby. Proto se autor snažil v tomto textu uvádět časté odkazy na to, odkud jsou informace doplňující referenční manuály převzaty, aby si čtenář mohl jejich věrohodnost snadněji ověřit.

Inovace v SystemVerilogu jsou zaměřeny především na usnadnění práce s rozsáhlými projekty, i když mnoho zlepšení zavedených v tomto jazyku je užitečné i u menších projektů. Jedním z důležitých zlepšení je omezení redundance zápisu. Ve Verilogu je například nutné při vkládání komponent strukturálním stylem

vypsat seznam bran vkládané komponenty do příkazu vložení. U malých projektů to příliš nevadí a může se to zdát jako drobnost. V moderních konstrukcích však komponenty často mívají stovky i více bran, jejichž seznam je nutno ve všech vloženích znovu vypisovat. Dojde-li při vývoji ke změně v branách, je nezbytné všechna vložení opravit. V SystemVerilogu je vylepšenou syntaxí dosaženo toho, že většinou stačí opravit seznam bran jen v jednom místě, není potřebné tuto opravu provádět u každého vložení této komponenty. Obecně je tam kladen důraz na zásadu, že malé změny v konstrukci by neměly vyvolávat nutnost oprav na mnoha místech.

Jazyk Verilog tedy můžeme v současné době považovat za podmnožinu jazyka SystemVerilog, Pokud nebude výslovně uvedeno jinak, platí v následujícím textu vše, co bude psáno pro Verilog, i v SystemVerilogu (samozřejmě však ne naopak). Přesto však někdy, když to bude účelné, budeme o Verilogu a o SystemVerilogu mluvit jako o dvou jazycích, i když to není zcela přesné.

V jazycích Verilog i SystemVerilog je možno vytvořit neefektivní konstrukce, přičemž efektivnost (nebo její nedostatek) nemusí být na první pohled ze zdrojového textu patrná. To však platí i o jiných jazycích vyšší úrovně. Výsledná efektivnost konstrukce závisí jak na kvalitě programových návrhových prostředků, tak i na zkušenosti konstruktéra (návrháře) – výhodu zde mají konstruktéři, kteří prošli přípravou s jazykem nižší úrovně, jako je například ABEL, kde je nižší stupeň abstrakce a je více zřejmá souvislost mezi popisem a jeho fyzickou realizací. Jak vyzrávají návrhové systémy, přebírají na sebe stále větší díl potřebné dovednosti a „chytrosti“, dříve nezbytných u konstruktéra. Tyto vlastnosti samozřejmě bude konstruktér nadále potřebovat, budou však nasměrovány poněkud jinak než dříve.

Vztah k jazyku VHDL: Jak již bylo uvedeno, vedle jazyků Verilog a SystemVerilog (v tomto odstavci budeme o obou dále mluvit jako o Verilogu) se setkáme také s jazykem VHDL, který má podobné použití. Jazyk Verilog (a zejména SystemVerilog) má syntaxi podobnou jazyku C, zatímco jazyk VHDL odpovídá spíše jazykům skupiny Ada. Podobnost se u jazyka SystemVerilog projevuje především u pokročilejších jazykových konstruktů, jako jsou struktury nebo uniony. Verilog je aspoň ve svých základních konstruktech poněkud jednodušší a zápisy v něm jsou kratší než ve VHDL, kde zvláště práce s typy dat může začátečníkovi činit obtíže. Uvádí se, že z praktického hlediska jsou jazyky Verilog i VHDL zhruba stejně použitelné, i když zejména podpora rozsáhlých modelů se zdá být u SystemVerilogu dokonalejší, a že pokud návrhář zvládne jeden z nich, není pro něj velký problém přejít na druhý. O všech se říká, že je vcelku snadné naučit se jim, ale je obtížné zvládnout je mistrovsky. Pro běžného uživatele není aspoň z počátku příliš smysluplné usilovat o podrobné a aktivní zvládnutí obou jazyků Verilog i VHDL. Nejnápadnější rozdíl mezi těmito jazyky pravděpodobně spočívá v tom, že jazyk VHDL má značně přísnější pravidla syntaxe než Verilog. V textech zapsaných ve Verilogu se počítá s tím, že se simulátor nebo syntetizér bude snažit správně pochopit, co mu návrhář chce textem říci. V jazyku VHDL však mívají i drobné odchylky od předepsané formy zápisu za následek přerušení syntézy či simulace a hlášení chyby, zatímco ve Verilogu bývají častěji různé formy zápisu tolerovány. To může být pro uživatele Verilogu příjemné, ale může se také stát, že systém porozumí textu jinak, než jak to návrhář myslel, a ten pak může být překvapen výsledkem. Toto překvapení bývá tím nepříjemnější, čím v pozdější fázi práce na projektu se projeví. Příkladem typické chyby tohoto druhu je, když vynecháme rozsah v deklaraci proměnné, které se dále přiřazuje celočíselná hodnota – syntakticky je vše v pořádku, ale tato proměnná se pak chápe jako jednobitová, což neumožňuje rozlišit více hodnot než nula a nenula. Odhalení takové chyby nemusí být ve složitějších konstrukcích snadné. Lze také doporučit, aby se návrháři při psaní textu ve Verilogu příliš nesnažili experimentovat s jazykovými konstrukty a používali osvědčený styl zápisu, u něhož si ověří, jak mu systém rozumí, a pracovali s ním třeba i za cenu poněkud složitějšího vyjadřování, než by mohlo být nezbytné. Druhou možností je důkladnější ověření funkce simulací nebo jinými metodami, než jak je to nutné u jazyka VHDL, to však obvykle bývá pracnější cesta.

V dále uvedeném výkladu různých jazykových konstruktů se občas vyskytují slova a fráze jako „obvykle“, „měl by být“ a podobně. Tím je většinou myšlena skutečnost, že současně existující návrhové systémy jsou na různé úrovni vyzrálosti a inteligence. Systém, který je možno označit jako vyspělý, bude správně sestavené popisy schopen zpracovat a optimalizovat bez problémů. Ne všechny současné systémy to však zvládnou stejně úspěšně. Obecně lze říci, že čím blíže odpovídá popis fyzické realizaci popisované struktury v cílovém obvodu, tím pravděpodobnější je optimální výsledek syntézy. Například některé způsoby popisu s algoritmickým charakterem mohou působit při syntéze problémy, i když jsou syntakticky správné,

protože zapojení, které má být výsledkem syntézy, žádný algoritmický charakter ve skutečnosti nemá. Potíže tohoto druhu se vyskytovaly zejména u starších verzí programových nástrojů, u současných moderních verzí jsou již většinou překonány. Začneme-li však pracovat s novým systémem, bývá vždy vhodné zjistit rozsah syntetizovatelných konstruktů jazyka v dokumentaci systému, jako je například **[48]** nebo **[58]**, a ověřit si na jednoduchých příkladech, zda syntéza probíhá podle očekávání. Někdy se může stát, že zdánlivě jasný text je syntetizován do struktury, která obsahuje nadbytečnou logiku, případně její část může být nevhodná (mohou například vznikat problémy s hazardy) nebo zcela nefunkční. Lze sice čekat, že syntetizéry budou stále dokonalejší a podobných problémů bude čím dál méně, dovednost a zkušenosti konstruktéra však sotva kdy plně nahradí.

K přenositelnosti kódu je nutno připojit poznámku. Jazyk SystemVerilog je velmi rozsáhlý, a do značné míry to platí i pro jazyk Verilog, který byl původně určen především pro simulaci číslicových systémů. Jeho použití pro syntézu se začalo rozpracovávat později a využívá se při něm jen část jeho prostředků. V počátečních fázích tohoto směru využití jazyka Verilog byly vytvářeny jednoduché syntetizéry se značně omezeným rozsahem jazykových konstruktů, které jimi byly podporovány. Postupně, jak syntetizéry vyspívaly, se tento rozsah rozšiřoval. Proto je možno očekávat, že základní konstrukty jazyka budou různými syntetizéry zpracovány stejně, ale neobvyklé způsoby popisu mohou dávat různé výsledky při zpracování různými syntetizéry – obvykle sice ne z hlediska funkčního, což by byla vyloženě chyba systému (ale i ty 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 o komponentě musí uživatel znát (popřípadě si je najít v popisu komponenty). Behaviorální popis je tedy obvykle pro člověka srozumitelnější, a budeme jej zde, pokud to bude možné, používat. Použití strukturálního stylu je typické například u interních souborů generovaných návrhovým systémem, které jsou určeny pro další strojové zpracování (počítač nemusí popisu rozumět, na rozdíl od člověka mu stačí syntaktická správnost). Označení těchto stylů však není součástí definice jazyka a slouží jen pro orientaci uživatele. V jednom modulu můžeme styly libovolně kombinovat, pokud dodržíme syntaktická pravidla jazyka. Někteří autoři zavádějí ještě pojem **stylu popisujícího tok dat** (*dataflow style*), který zhruba řečeno spočívá v použití kontinuálních přiřazovacích příkazů (klíčové slovo `assign` – viz **odst. 2.5.1**), zde však budeme popis tohoto druhu počítat k behaviorálnímu stylu.

Setkáme se také s pojmem **behaviorální modelování** (modelování na **behaviorální úrovni**), čímž se rozumí modelování na vysoké úrovni abstrakce, kde například není ještě definována šířka dat a simulací se ověřuje chování takového abstraktního modelu. Modely určené k syntéze se nejčastěji popisují na **úrovni RTL** (*Register Transfer Level*), kdy se v modelu vyčlení paměťové prvky (registry) a kombinační obvody mezi nimi, a ty se popíší – nejčastěji behaviorálním stylem. Popis na úrovni RTL pak specifikuje přenos signálů mezi registry, včetně jejich případné modifikace kombinačními obvody. Ještě nižší úroveň abstrakce je u strukturálního modelování, kde již jde o popis na **úrovni hradel** (*gate level*) – jde obvykle o popisy generované návrhovými systémy, které slouží například k časové analýze.

Terminologické poznámky: Pojmy jako „behaviorální styl“ nebo „strukturální styl“ nejsou v oficiální terminologii Verilogu a SystemVerilogu (obsažené např. v dokumentech **[1]**, **[6]**) zavedeny, jsou to jen pomocné vyjadřovací prostředky, které mají za účel usnadnit orientaci ve způsobech popisu.

Podobně není v této terminologii zaveden pojem **komponenty** – tou budeme rozumět dílčí část konstrukce (například modul) vkládané strukturálním stylem do jiné jednotky. Místo něj se někdy užívá výraz „*child module*“; modul, do něhož je komponenta vkládána, se označuje výrazem „*parent module*“. Autorovi se nepodařilo najít vhodné české výrazy pro tyto termíny, a proto budeme pro vkládaný modul používat podobně jako v jazyku VHDL označení „komponenta“. Nicméně se i v anglické literatuře týkající se Verilogu a SystemVerilogu dosti často setkáme s označením *component* místo výrazu *child module*. V SystemVerilogu mohou být kromě modulů vloženy do vyšší jednotky také rozhraní a programy. V dokumentech SystemVerilogu se označení *parent – child* v tomto smyslu používá jen zřídka.

2.1.2 Lexikální jednotky

Zdrojový text ve Verilogu i v SystemVerilogu je tvořen sledem **lexikálních jednotek** (*lexical tokens*). Mezi tyto jednotky patří zejména:

- prázdné znaky;
- klíčová slova, atributy, direktivy pro kompilátor, názvy systémových funkcí a úloh;
- uživatelské identifikátory;
- operátory;
- číselné, řetězcové, strukturní literály a literály pro zápis hodnot polí;
- komentáře.

Prázdné znaky (*white spaces*), tedy mezery, tabulátory a znaky nového řádku slouží podobně jako v jiných jazycích jako oddělovače, nemají další význam a můžeme je používat podle libosti k rozčlenění dalších lexikálních jednotek v textu tak, aby byl text přehledný (s malými výjimkami, například je-li mezera součástí textového řetězce). Někdy mohou být tyto oddělovače vynechány, lze-li hranice lexikálních jednotek rozpoznat jinak (jak je to například před seznamem bran v textu `TXT2.1`).

Vyhrazená (klíčová) slova (*reserved words*, *keywords*) musí být ve Verilogu i v SystemVerilogu psána malými písmeny. Jsou to slova, jejichž význam je stanoven v definici jazyka. Jejich seznam je uveden v **odst. 1.1**. Editory návrhových systémů tato slova obvykle vybarvují, čemuž odpovídá dříve uvedená poznámka, že zde je budeme zapisovat tučně modrou barvou.

Jednotlivé verze Verilogu definují klíčová slova, přičemž vyšší verze přebírají klíčová slova nižších verzí. V SystemVerilogu je situace poněkud odlišná: je zde možno zvolit verzi jazyka, jejíž klíčová slova jsou v určitém bloku zdrojového textu rozeznávána (považována za platná klíčová slova), přičemž jiná slova jsou považována za běžné identifikátory, i když jsou ve vyšších verzích klíčovými slovy. K tomu jsou zde zavedeny direktivy `` `begin_keywords `` a `` `end_keywords ``, přičemž první z nich je následována specifikátorem verze. Platné specifikátory jsou: `1800-2012`, `1800-2009`, `1800-2005`, `1364-2005`, `1364-2001`, `1364-1995` (je ještě definován specifikátor `1364-2001-noconfig`, kde jsou z verze Verilog-2001 vynechána některá klíčová slova související s konfigurací, viz např. **[6]**). Tyto direktivy musí být umístěny mimo konstrukční prvky (jako jsou například moduly, podrobněji v **odst. 2.2**), čili v celém konstrukčním prvku musí být platná stejná sada klíčových slov. Není-li tímto způsobem verze specifikována, odpovídají rozeznávaná klíčová slova výchozímu stavu příslušné implementace.

Pragmata (*pragmas*) jsou podle standardu pro syntézu jazyka Verilog 1364.1-2002 **[3]** konstrukty, které nemají ve standardu jazyka definovaný sémantický význam, jsou však užívány pro řízení programových nástrojů, jako jsou syntetizéry, při zpracování zdrojových textů. Uvedený standard připouští jedinou formu zápisu pragmat ve zdrojových textech zapsaných ve Verilogu určených k syntéze ve formě **atributů** (*attributes*). Ve standardech Verilogu i SystemVerilogu slouží pro ohraničení atributů dvě speciální dvojice znaků `(*` a `*)`. Mnoho syntetizérů však připouští také použití pragmat zapsaných jako metakomentáře. V SystemVerilogu má termín *pragma* specifický význam – patří mezi direktivy; nebudeme jej však zde rozebírat. Slovem „atributy“ budeme také označovat vlastnosti objektů, jako je znaménkovost nebo rozsah signálů a podobně.

V publikaci **[3]** je uvedeno 15 atributů, které mají být podporovány syntetizéry Verilogu odpovídajícími standardu 1364.1-2002. O některých z nich se zmíníme později. Je však dovoleno, aby syntetizéry podporovaly ještě další vlastní specifické atributy, jejichž význam a způsob použití je nutno najít v dokumentaci k těmto nástrojům.

O **direktivách pro kompilátor** bude řeč později (**odst. 2.9**). Znakem dolaru (`$`) začínají názvy **systémových funkcí a úloh**, o nichž budeme mluvit v **odst. 2.7**.

Uživatelské identifikátory (uživatelské symboly, *user identifiers*, stručněji často jen identifikátory) jsou názvy sloužící k označení signálů, parametrů, konstant, funkcí a podobných objektů. Volí je návrhář (uživatel), a samozřejmě nesmějí být totožné s klíčovými slovy. Budeme také někdy mluvit v užším smyslu

o **jménu** nebo **názvu** objektu, jehož identifikátor pak může obsahovat například i odkaz na slohu (**odst. 2.2.3**), v níž je objekt definován. Jméno v uživatelském identifikátoru musí začínat písmenem anglické abecedy nebo podtržítkem a na dalších pozicích může obsahovat písmena, číslice `0` až `9`, podtržítko nebo znak dolaru (`$`). Vestavěné systémové funkce a úlohy mají identifikátory začínající znakem dolaru. V identifikátorech ve Verilogu i v SystemVerilogu se rozlišují malá a velká písmena, jinak řečeno, tyto jazyky jsou **citlivé na malá a velká písmena** (*case sensitive*). Není však příliš vhodné na to spoléhat, zejména pokud v projektu máme také jednotky zapsané v jazyku VHDL, kde se malá a velká písmena nerozlišují. Vícepísmenné uživatelské identifikátory budeme obvykle pro snadnější orientaci zapisovat s velkým počátečním písmenem; pokud budou odpovídat víceslovnému názvu, budou takto vyznačena počáteční písmena těchto slov. Identifikátory dovolují zapisovat odkazy na objekty, které označují, tedy umožňují přístup k těmto objektům, jak bude podrobněji uvedeno dále (**odst. 2.1.3**).

Dosud jsme měli na mysli identifikátory, kterým se někdy říká **jednoduché identifikátory** (*simple identifiers*). Verilog i SystemVerilog dovolují také používat **odskočené identifikátory** (*escaped identifiers*), které začínají zpětným lomítkem (*backslash*) a jsou ukončeny prázdným znakem – mezerou, tabulátorem nebo znakem nového řádku **[1]**, **[6]**. Zpětné lomítko a ukončující prázdný znak se zde nepovažují za část identifikátoru, takže například odskočený identifikátor `\abc` je považován za totožný s jednoduchým identifikátorem `abc`. Odskočené identifikátory mohou obsahovat jakékoliv tisknutelné znaky ASCII. Je však potřebné používat je s opatrností, protože mohou vést k problémům – viz např. **[12]**. Řetězec znaků odpovídající klíčovému slovu, kterému předchází zpětné lomítko, není považován za klíčové slovo.

Poznámka: V SystemVerilogu se vyskytuje větší množství různých druhů objektů označovaných uživatelskými identifikátory. Pro lepší orientaci budeme pro některé z nich používat identifikátory s příponou, která je bude odlišovat od identifikátorů jiných druhů objektů. Pro označení uživatelsky definovaných typů budeme používat identifikátory tvaru *jméno*`_t`. Podobně pro označení struktur budeme používat identifikátory tvaru *jméno*`_st`, rozhraní budou mít identifikátory *jméno*`_if`, konstrukty `modport` budou označeny *jméno*`_mp`. Značky pro možné hodnoty výčtových typů pak budeme psát velkými písmeny.

O **operátorech** a **literálech** budou podrobné informace uvedeny v dalším textu.

Komentáře (*comments*) ve Verilogu i v SystemVerilogu začínají dvěma lomítky (`//`) a končí na konci řádku (jednořádkový komentář); má-li takový komentář pokračovat na dalším řádku, musí toto pokračování opět začínat dvěma lomítky. Za komentář je považován také text uzavřený mezi dvojicemi znaků `/*` a `*/` (jako v jazyku C). Takový komentář může být víceřádkový. Text následující za komentářem uzavřeným do těchto znaků je opět zpracováván jako platný kód, i když není od předcházejícího komentáře oddělen novým řádkem.

Někdy se setkáme s tzv. **metakomentáři** (*meta comments*), které jsou analogické pragmatům. Ty se používají v návrhových systémech k řízení programových nástrojů (syntetizéru apod.). Metakomentáře vypadají formálně jako komentáře, nástroje je však mohou rozpoznat a řídit se jejich významem, který je definován specificky pro jednotlivé nástroje. Standard 1364.1-2002 však metakomentáře pro řízení syntézy textů zapsaných ve Verilogu výslovně nepřipouští.

Důležité upozornění: Nepoužívejte znaky s diakritikou (jako jsou česká písmena s háčky a čárkami) ani v komentářích! Simulátory a syntetizéry je někdy mohou vyhodnotit tak, že výsledky zpracování takového textu jsou zcela nesmyslné, nebo tyto znaky způsobí obtížně identifikovatelné chyby. Podobné problémy mohou působit také mezery nebo diakritika v názvech souborů a složek, i když to operační systém počítače dovoluje. Také texty získané kopírováním ze souborů pořízených jinými editory (např. ve formátu doc, pdf apod.) mohou obsahovat znaky, které nejsou viditelné ve výpisu a způsobují chybnou funkci nástrojů.

2.1.3 Definice, deklarace, odkazy, přístupnost objektů

Definice (*definitions*) představují vymezení pojmů, s nimiž v textu pracujeme. V užším smyslu je zde budeme používat k vymezení objektů, jako jsou například moduly, typy signálů a podobně. Důležitým aspektem takových definic je jejich syntaktická správnost.

Deklarace (*declarations*) vycházejí zpravidla již z definovaných pojmů a zavádějí jejich označení (jména, identifikátory), pomocí nichž se na deklarované objekty odkazujeme. V některých případech mohou být deklarace implicitní, jako to je u implicitních deklarací propojení (**odst. 2.3.2**).

Rozdíl mezi definicí a deklarací je jemný. I když se budeme většinou snažit o rozlišení mezi těmito pojmy, nebude to vždy snadné a někdy se nevyhneme jejich překrývání.

Odkazy (*references*) slouží pro **přístup** (*access*) k deklarovaným objektům. **Přístupností** (*accessibility*) objektu (nebo jeho části, např. části pole) se rozumí to, že je definováno jeho syntakticky správné označení (jako jsou například bitové nebo částečné výběry z vektorů), které může být použito ve výrazech, v přiřazovacích příkazech a podobně.

2.2 Jednotky, konstrukční prvky

Ohraničeným částem projektu, v jejichž rozsahu jsou například platné určité definice či deklarace, budeme říkat **jednotky**. Jako příklady jednotek můžeme uvést kompilační jednotky, moduly, rozhraní, programy (to jsou konstrukty vyskytující se především ve verifikačních projektech, zde se o nich zmíníme jen okrajově), bloky `begin-end` a `fork-join`, úlohy, funkce. Ve standardu SystemVerilogu [6] je dále zavedeno označení **konstrukční prvky** (*design elements*) jazyka pro moduly, rozhraní, slohy (`package`), a také pro další jednotky používané především při verifikaci – programy, kontrolní jednotky (*checker*), primitivy a konfigurace. Někdy se setkáme také s jejich označením **konstrukční jednotka** (*design unit*).

Primitivy představují hradla a spínače používané na nízké úrovni modelování (na úrovni hradel). Standardy Verilogu a SystemVerilogu obsahují několik zabudovaných primitiv (*built-in primitives*), o nichž se blíže zmíníme v **odst 2.8**. Kromě toho může uživatel vytvořit své vlastní primitivy označované zkratkou UDP (*user-defined primitive*). Tyto primitivy se používají hlavně při modelování na úrovni hradel a zde je nebudeme podrobněji rozebírat, zájemce najde bližší informace v manuálech [1], [6].

2.2.1 Moduly a jejich brány

V tomto odstavci i v následujících pododstavcích se předběžně pracuje s některými pojmy, jejichž význam bude podrobněji vysvětlen později. Jsou to pojmy jako typ signálu (proměnná, propojení) a datový typ, o nichž budeme blíže mluvit v **odst. 2.3** a **odst. 2.3.1**, o vektorech a jejich rozměrech bude pojednávat **odst. 2.3.3**.

Jednou ze základních jednotek popisu modelu vyvíjené konstrukce v jazycích Verilog a SystemVerilog (a pro úvodní seznámení s těmito jazyky patrně jednotkou nejdůležitější) je **modul** (`module`). Typický modul zhruba odpovídá tomu, co i v běžném jazyce označujeme za modul, tedy části konstrukce s definovanými vlastnostmi. Ve Verilogu nemůže být modul definován uvnitř jiného modulu (může však do něj být vložen jako komponenta), v SystemVerilogu je taková vnořená definice modulu nacházející se uvnitř jiného modulu dovolena (**odst. 2.8.5**).

Důležitou částí definice modulu je deklarace jeho **bran** (*ports*). Pokud je popisovaný modul určený k syntéze nejvyšší hierarchickou jednotkou – vrcholovou jednotkou (podrobněji o hierarchii budeme mluvit v **odst. 2.8.1**) nebo pokud jde o popis plochého typu, budou brány odpovídat signálům na vývodech cílového obvodu, do něhož bude konstrukce implementována. Je-li však tento modul použit jako komponenta nižší úrovně vložená do vyšší jednotky v hierarchickém modelu, budou brány představovat připojovací prvky modulu pro signály obsažené (viditelné) v jednotce, do níž je komponenta vložena, kterými komponenta komunikuje s dalšími částmi této jednotky. Přenos signálů branami se děje podle pravidel pro kontinuální přiřazovací příkazy, jak to blíže uvidíme v **odst. 2.5.1**.

Jako jednoduchý příklad syntaxe zdrojového textu uvedeme modul představující model čtyřbitového komparátoru, u kterého bude zápis ve Verilogu i v SystemVerilogu stejný. V příkladu jsou řádky opatřeny vpravo čísly (komentář) pro usnadnění odvolávek na řádky.

```
module EqComp4(input [3:0] x,y, //1   x,y jsou 4bitove vektory // TXT2.1
               output z);       //2   z je skalarni (jednobitovy) signal
  assign z = (x == y);          //3
endmodule                       //4
```

Definice (popis) modulu začíná **hlavičkou** (*header*), která obsahuje název modulu a dále v závorce **seznam bran** (*port list*), jimiž modul komunikuje s okolím. V textu `TXT2.1` je v řádcích `1` a `2` hlavička modulu s názvem `EqComp4`. V seznamu bran jsou zde deklarovány brány modulu (zde `x`, `y`, `z`), čímž jsou současně deklarovány vnitřní signály modulu připojené k branám – budeme jim říkat signály (těchto) bran. Zápis seznamu bran uvedený v textu `TXT2.1` odpovídá verzi Verilog-2001 a označuje se jako zápis ve stylu ANSI.

V hlavičce zapsané ve stylu ANSI se specifikuje **směr** (*direction*) bran, tj. brány mohou být **vstupní** – `input`, **výstupní** – `output` nebo **obousměrné** – `inout`. Dále zde může být uveden **datový typ** signálu brány včetně **rozsahu** (*size*), je-li tento signál vektorový (často se pak mluví o datovém typu brány, i když ve skutečnosti jde o typ jejího signálu), a u datových typů, které to připouštějí, zde může být určeno, vyjadřuje-li příslušný signál čísla se znaménkem nebo bez něj (tzv. znaménkovost, viz **odst. 2.3.5**). Jak bude dále podrobněji rozebráno, uvedení všech těchto atributů v hlavičce není povinné; pokud některé nejsou uvedeny, nastaví se automaticky (implicitně) podle pravidel, která budou naznačena dále.

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 stejným jako v deklaraci směru; není-li typ explicitně deklarován, automaticky se signálu takové brány přiřadí **výchozí typ** **wire** (výchozí typ lze změnit direktivou `` `default_nettype `` – to zde platí pro brány `Flg1` a `Flg2`. Typ signálu brány `Clk` je explicitně deklarován a je to tedy vždy **wire**.

2.2.1.2 *Brány ve Verilogu-2001*

Verilog-2001 zavádí výrazné zjednodušení zápisu deklarace bran – styl ANSI, kde jsou jejich identifikátory, a také rozsah, směr a typ signálů jednotlivých bran uvedeny přímo v seznamu bran (rovněž zde zůstaneme u termínu „typ“ Verilogu):

```
module Processor (inout  tri  [15:0] DataIO,
                  input  wire [15:0] DataIn,
                  output wire [15:0] DataOut,
                  output reg   [2:0] Abc,
                  input  wire        Clk,
                  output             Flg1,Flg2);
```

Na prvním místě je zde deklarován směr, pak (nepovinně) typ a rozsah, a konečně identifikátor brány. Není-li explicitně deklarován typ a/nebo rozsah signálu brány, platí zde pravidla jako ve Verilogu-95. U skupin bran se stejným směrem a typem, lišících se ale rozsahem, musí být směr a případně typ deklarován u každé skupiny samostatně (deklarace směru a případně typu se musí u každé takové skupiny opakovat). Za typem může být u vektorů deklarována znaménkovost – **signed**, **unsigned**. Syntaxe Verilogu-95 je ve Verilogu-2001 samozřejmě také platná.

2.2.1.3 Brány v SystemVerilogu

V SystemVerilogu jsou zavedeny další úpravy. Zůstává zde samozřejmě možnost deklarovat brány ve stylu Verilog-95 i ve stylu ANSI. Nemusí však zde být uveden směr brány – výchozí směr brány v SystemVerilogu je `inout`: není-li směr specifikován explicitně, je brána obousměrná. Dále, je-li u druhé nebo další brány definován datový typ, ale ne směr, přebírá tato brána směr od předchozí brány. Hlavička modulu procesoru může být nyní psána takto:

```
module Processor (     tri [15:0] DataIO,
                 input  wire[15:0] DataIn,  wire Clk,
                 output wire[15:0] DataOut, reg[2:0] Abc, wire Flg1,Flg2);
```

Zpětná kompatibilita SystemVerilogu se všemi verzemi Verilogu je zajištěna pravidlem, podle kterého platí, že pokud u první brány v seznamu není explicitně specifikován směr ani datový typ (ani její rozsah, má-li tato brána charakter vektoru – jak vyplývá z doplnění deklarace u modulu `m1` v následujících příkladech), je celá hlavička podřízena syntaxi Verilogu-95. Žádná další brána nesmí v tomto případě mít v seznamu specifikován směr ani datový typ, ty pak musí být uvedeny v samostatných deklaracích.

V SystemVerilogu je ale situace poněkud zkomplikována tím, že je zde zaveden nový datový typ `logic` (pojem datového typu je blíže vysvětlen v **odst. 2.3**), který může být přiřazen jak proměnné, tak i propojení. Datový typ `reg` implikuje vždy proměnnou, v SystemVerilogu ale nepoznáme jen podle datového typu `logic`, jedná-li se u uvažovaného signálu o proměnnou nebo o propojení. Nicméně se tam obvykle proměnné datového typu `reg` nahrazují stejně pojmenovanými proměnnými s datovým typem `logic`. Pak bude mít hlavička modulu procesoru tvar:

```
module Processor (   tri [15:0] DataIO,
                 input  wire[15:0] DataIn,  wire  Clk,
                 output wire[15:0] DataOut, logic[2:0] Abc, wire Flg1,Flg2);
```

Deklarace výstupní brány v SystemVerilogu s datovým typem `logic` pak specifikuje její signál jako proměnnou, což odpovídá obdobným deklaracím ve Verilogu (u vstupní nebo obousměrné brány je však takový signál specifikován jako propojení). Pokud ale není v deklaraci výstupní brány uveden datový typ, je inferováno propojení výchozího typu (pak jde o tzv. **implicitní datový typ**, *implicit data type*). Tato zdánlivá složitost je cenou za zpětnou kompatibilitu SystemVerilogu s Verilogem, prakticky však nepředstavuje závažný problém, jak se o tom přesvědčíme na příkladech konstrukcí v dalších kapitolách. Zcela přesné formulace jsou však ještě složitější a lze je najít v příručce **[6]**.

Pro ilustraci uvedeme několik příkladů deklarací bran v SystemVerilogu:

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

module m10(integer x, signed y);

module m11(output var x,
           input y);
module m12(output logic x, y);

module m13(output signed [3:0] x,
           integer y);
```

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

Má-li tedy signál brány deklarované klíčovými slovy `input logic` být proměnná, musí se to v deklaraci explicitně uvést klíčovým slovem `var`.

SystemVerilog zavádí další typ brány – **referenční bránu** (*reference port*) označenou klíčovým slovem `ref`. Referenční brána slouží k předání hodnoty signálu odkazem z jednoho modulu do druhého modulu. Není to syntetizovatelný konstrukt a dále se jím zde nebudeme zabývat.

V SystemVerilogu je možno použít jako bránu také rozhraní. Syntaxi a další podrobnosti uvedeme v **odst. 3.3**.

2.2.2 Prostory jmen a rozsahy působnosti definic a deklarací

Objekty, s nimiž ve Verilogu i v SystemVerilogu pracujeme, jako jsou signály, jejich typy, funkce, úlohy a podobně, se v popisu deklarují. To lze učinit několika způsoby, které zde stručně uvedeme (jsou však přípustné i implicitní deklarace – viz **odst. 2.3.2**). Důležité je vědět, kde se tyto deklarace mohou nacházet a jaký je jejich **rozsah působnosti** (*scope*), v němž jsou deklarované objekty **viditelné** (*visible*), tj. kde s těmito objekty pak můžeme pracovat. Podobně je důležité, v jakém rozsahu jsou viditelné definice konstruktů, jako jsou například uživatelsky definované nebo výčtové typy v SystemVerilogu. Anglický termín *scope* se někdy překládá také jako **obzor** nebo **prostor**.

Dokumenty Verilogu [1] a SystemVerilogu [6] mluví v této souvislosti o **prostorech jmen** (*name spaces*). V SystemVerilogu je pro identifikátory zavedeno osm prostorů jmen; dva jsou **globální** (prostor definic a sloh) – obsahují celý projekt, další dva jsou globální v kompilační jednotce (prostor kompilační jednotky a textových maker), zbývající čtyři (prostor modulů, bloků, bran a atributů) jsou **lokální** – jejich rozsah je omezen na příslušný konstrukt (modul, blok apod.). Identifikátory deklarované v každém z těchto prostorů jsou v něm viditelné a musí tedy být jedinečné. Tyto prostory v SystemVerilogu jsou (čtenář neobeznámený s termíny v tomto výčtu může jej přeskočit a vrátit se k němu v případě potřeby později):

- **Prostor definic** je globální a týká se definic všech modulů, rozhraní, programů a primitiv, které nejsou vnořeny do jiných definic. Je-li jméno užito v definici některého z těchto konstruktů v některé kompilační jednotce, nesmí být užito k nevnořené definici jiného takového konstruktu v jakékoliv kompilační jednotce.
- **Prostor sloh** je také globální a týká se identifikátorů sloh. Je-li jméno užito v definici nějaké slohy v některé kompilační jednotce, nesmí být užito k definici jiné slohy v jakékoliv kompilační jednotce.
- **Prostor kompilační jednotky** existuje mimo definice modulů, rozhraní, sloh, kontrolních jednotek (*checkers*), programů a primitiv v kompilační jednotce. Tento prostor se týká funkcí, úloh, kontrolních jednotek, parametrů, uživatelských typů, deklarací propojení, proměnných a pojmenovaných událostí v příslušné kompilační jednotce.
- **Prostor textových maker** je globální uvnitř kompilační jednotky. Jména textových maker jsou definována v pořadí, v němž jsou seřazeny soubory, které tvoří kompilační jednotku. Případná další definice téhož jména textového makra přepisuje pro zbývající soubory jeho předchozí definici.
- **Prostor modulů** představuje rozsah modulů (tedy textu ohraničeného klíčovými slovy `module` a `endmodule`), rozhraní, sloh, primitiv, programů a kontrolních jednotek, a týká se definic vnořených modulů, rozhraní, funkcí, úloh, identifikátorů vložení komponent, parametrů, uživatelských typů, pojmenovaných bloků, programů, kontrolních jednotek, deklarací propojení, proměnných a pojmenovaných událostí.
- **Prostor bloků** je v pojmenovaných či nepojmenovaných blocích, ve funkcích, úlohách a specifikacích (`specify`). Týká se definic pojmenovaných bloků, funkcí, úloh, parametrů, uživatelských typů, deklarací proměnných a pojmenovaných událostí v těchto konstruktech.
- **Prostor bran** představuje rozsah modulů, rozhraní, primitiv a programů. Překrývá prostory jmen modulů a bloků. Týká se identifikátorů objektů propojujících různé prostory jmen. Jméno brány deklarované v tomto prostoru může být znovu použito v prostoru modulu pro deklaraci propojení nebo proměnné se stejným jménem, jako je jméno brány.

- **Prostor atributů** je uzavřen mezi dvojicemi znaků `(*` a `*)` připojenými k jazykovému prvku specifikujícími jeho atribut. Jméno atributu může být definováno a použito jen v tomto prostoru, a žádné jiné jméno zde nemůže být definováno.

Ve Verilogu je zavedeno jen sedm prostorů jmen: prostor definic a textových maker (globální prostory), a dále lokální prostory modulů, bloků, bran, specifikačních bloků a atributů. Tyto prostory jsou analogické prostorům v SystemVerilogu. Podrobnosti najde čtenář v manuálu [1].

O globálních objektech, jejichž definice není vnořena do jiné definice, se někdy také říká, že jsou definovány (popř. platné) na **vrcholové úrovni** (*top level*) projektu.

V další části tohoto odstavce budeme mluvit o definicích objektů; budeme tím myslet i deklarace takových objektů, kde to z povahy těchto objektů vyplývá (jako jsou proměnné, propojení, parametry, konstanty – v anglické dokumentaci se pojmy definice a deklarace často nerozlišují či zaměňují).

Ve Verilogu se datové objekty, úlohy a funkce definují (či deklarují) uvnitř modulů, mezi klíčovými slovy `module` a `endmodule`, a takto definované objekty jsou zde lokální, tj. definice jsou viditelné (platné) jen uvnitř modulu, v němž se nacházejí. Ve Verilogu je také možno deklarovat lokální proměnné v pojmenovaných blocích, ve funkcích a v úlohách. Definice a deklarace objektů, které mají být používány ve více modulech (například v hierarchické konstrukci), je ve Verilogu nutno opakovat v každém takovém modulu. To v menších projektech příliš nevadí, ale v rozsáhlých projektech to může snadno vést k chybám, zejména pokud je třeba 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 SystemVerilogu. **Používání definic a deklarací platných v prostoru `$unit` se proto nedoporučuje**, jak uvádí např. **[11]**, i když to pravidla syntaxe SystemVerilogu dovolují. Všechny definice a deklarace, které mají být sdíleny více jednotkami (moduly, rozhraními), by měly být obsaženy ve slohách.

2.2.3 SystemVerilog: Slohy

Definice a deklarace objektů uvedené v jednotkách jako jsou moduly, rozhraní, programy a kontrolní jednotky mají rozsah působnosti omezený na tyto jednotky. Chceme-li s těmito objekty pracovat ve více jednotkách a pokud bychom se přitom omezili na pravidla Verilogu, bylo by potřebné jejich definice uvádět opakovaně v každé takové jednotce. To však bývá zdrojem chyb zejména při jejich úpravách (hlavně u složitějších definic – například u definic uživatelských typů, struktur, funkcí a podobných konstruktů), které je pak nezbytné provádět v každé jednotce obsahující tyto definice. SystemVerilog tuto nesnáz odstraňuje zavedením **sloh** (*packages*) – klíčové slovo `package`, poskytujících rozsah působnosti, který mohou sdílet další jednotky. Definice a deklarace uvedené ve slohách mohou být využívány v dalších jednotkách nebo i v dalších slohách. Stačí pak tedy upravit takovou definici či deklaraci v jednom místě – ve sloze, kde je uvedena, a oprava se automaticky projeví ve všech jednotkách, které se na ni odkazují.

Slohy mohou obsahovat definice či deklarace následujících syntetizovatelných konstruktů:

- deklarace parametrů (`parameter`, `localparam`);
- deklarace konstant (`const`);
- definice uživatelských typů (`typedef`);
- definice automatických funkcí a úloh (`function`, `task`);
- definice přetížení operátorů (`bind`).

Kromě toho může být užití sloh vhodné pro definice a deklarace celé řady nesyntetizovatelných konstruktů; do slohy lze rovněž importovat obsah dalších sloh. Slohy lze také použít pro deklarace proměnných a propojení (to však může někdy působit problémy a používá se zřídka, viz **[38]**).

Ve slohách se mohou vyskytovat procesy (**odst. 2.6**) jen uvnitř kontrolních jednotek (*checkers*). Přiřazení v deklaraci propojení (**odst. 2.5.1**) zde není přípustné.

Definice funkcí a úloh se statickými proměnnými, které jsou uvedeny ve slohách, nejsou syntetizovatelné. Podmínkou jejich syntetizovatelnosti je, že tyto funkce a úlohy i všechny proměnné v nich obsažené musí být automatické. O statických a automatických proměnných budeme podrobněji mluvit v **odst. 2.3.1**, o statických a automatických funkcích v **odst. 2.7**.

Definice sloh se umisťují do vrcholové úrovně, podobně jako jiné vrcholové jednotky (nevnořené moduly, rozhraní apod.). Syntaxe definice slohy je následující:

```
package jméno_slohy;
  deklarace_parametrů_konstant_typů_atd;
endpackage
```

Odkazy na obsah sloh: O každém objektu, který je ve sloze definován či deklarován, mluvíme jako o položce slohy. Tyto položky pak lze využívat následujícími způsoby:

- uvedením položky s přímým odkazem na slohu:

```
jméno_slohy::jméno_položky;
```

- importem určité položky obsahu slohy:

```
import jméno_slohy::jméno_položky;
```

- importem celého obsahu slohy:

```
import jméno_slohy::*;
```

Operátor v odkazu tvořený dvojicí dvojteček (tedy `::`) se nazývá **odkazovací operátor** (*scope resolution operator*). Hvězdička (`*`) v posledním způsobu odkazování je **zástupný znak** (*wildcard*), který zastupuje všechny položky obsažené ve sloze.

První způsob s přímým odkazem na slohu při každém použití položky znamená nutnost zapsat identifikátor položky vždy včetně jména slohy a odkazovacího operátoru. To může být účelné, není-li odkazů na slohu v jednotce mnoho. Je-li těchto odkazů více, bývá vhodnější položku nebo celý obsah slohy do jednotky importovat. Příkaz importu se pak zapisuje do odkazující se jednotky místo definic či deklarací položek, a potom se jako identifikátor každé takové položky může v jednotce používat pouhé její jméno, bez odkazu na slohu. Použití slohy uvidíme např. v textu `TXT2.9` (**odst. 2.3.11.10**).

Lokální definice a deklarace v modulu nebo v rozhraní i import určité položky obsahu slohy mají přednost před importem celého obsahu slohy se znakem `*`. Z hlediska konstruktéra přidává import celého obsahu slohy tuto slohu k rozsahu, v němž programové nástroje prohledávají zdrojový text při hledání definice či deklarace identifikátoru – prohledávají nejprve lokální definice a deklarace (v modulu apod.), pak ve slohách, jejichž celý obsah byl importován, a nakonec v prostoru kompilační jednotky (`$unit`). Podrobná pravidla pro případ, že položky s určitými identifikátory jsou definovány či deklarovány v jednotce, v níž je také zviditelněn obsah slohy obsahující stejné identifikátory, jsou poměrně složitá a lze je i s instruktivními příklady najít ve standardu [6].

Použití obsahu slohy v hlavičce modulu: Příkaz importu zapsaný uvnitř jednotky za její hlavičkou umožňuje odkazovat se na importované položky v následujícím textu jednotky, ne však v hlavičce – v seznamu bran či v deklaraci parametrů. Zde je možno použít přímý odkaz na slohu. Je-li však odkazů více, může být vhodnější zapsat příkaz importu do vrcholové úrovně podobně jako definici slohy, čímž se položky importují do kompilační jednotky a v celé této kompilační jednotce je pak možno odkazovat se na ně bez uvedení jména slohy a odkazovacího operátoru. Jak jsme však viděli v **odst. 2.2.2**, kompilační jednotky mohou být sestaveny odlišně v různých nástrojích, takže to může znamenat zhoršení přenositelnosti. SystemVerilog-2009 zavádí možnost zapsat příkaz importu přímo do hlavičky před seznam bran a před deklaraci parametrů. Příkaz se zapíše ve stylu ANSI za jméno modulu, a pak mohou následovat deklarace parametrů, jak to uvidíme na příkladu v **odst. 2.3.10** (`TXT2.6` – tam však není odkaz na slohu) a seznam bran. Příklad syntaxe takové hlavičky modulu (lomenými závorkami jsou vyznačeny nepovinné položky hlavičky):

```
module jméno_modulu <import jméno_slohy::*;> <#(deklarace_parametrů)>
                    <(seznam_bran)>;
```

V celém modulu včetně deklarací parametrů a seznamu jeho bran je pak možno používat položky deklarované v importované sloze.

Další možnosti použití sloh a práce s nimi jsou dosti široké, zájemce odkazujeme na příručku [6].

2.3 Datové objekty, jejich typy a hodnoty

Data, která se mohou v čase měnit, jsou ve Verilogu a v SystemVerilogu reprezentována signály. Jsou zde dvě základní skupiny signálů – propojení a proměnné, podrobnosti o nich budou uvedeny v **odst. 2.3.1**. Jako třetí skupina se někdy uvádějí parametry, které jsou z hlediska syntaxe signálům podobné, kde však jde o neproměnná data, nikoli o signály, které se mohou měnit; v SystemVerilogu jsou zavedeny ještě konstanty (deklarované klíčovým slovem `const`). O parametrech a konstantách budeme blíže mluvit v **odst. 2.3.10**. V literatuře o Verilogu se tyto skupiny obvykle označují za **typy**. Samotný pojem „typ“ se ve starší literatuře o Verilogu používal dosti vágním způsobem, postupně se konkretizoval až v dalších verzích. Tento termín zde budeme používat, podobně jako je to běžné v anglické literatuře pojednávající o jazycích Verilog a SystemVerilog, především v uvedeném smyslu, ale nevyhneme se jeho použití i v jiných souvislostech a s poněkud jiným významem. Také pojem signálu není v manuálech [1], [6] explicitně definován, používá se tam však způsobem, který odpovídá jeho významu v tomto textu.

Ve standardu SystemVerilogu [6] je zavedeno přesnější názvosloví. Je zde definován pojem **datového objektu** (*data object*), který představuje pojmenovanou entitu, k níž je přiřazena **datová hodnota** (*data value*) a **datový typ** (*data type*). K datovým objektům počítáme zejména propojení, proměnné, konstanty a parametry. Datový typ má dvě složky: množinu hodnot, kterých může příslušný datový objekt nabývat, a množinu operací, které s ním mohou být prováděny. V tomto smyslu se pak v SystemVerilogu mluví o čtyřhodnotových a dvouhodnotových datových typech, jak uvidíme dále. Datový typ zahrnuje také informaci o znaménkovosti (`signed`, `unsigned`) a o případných sbalených rozměrech příslušného objektu (má-li tento objekt charakter vektoru, viz **odst. 2.3.3**). Ve Verilogu [1] se pojem datového typu také vyskytuje, nerozlišuje se však mezi ním a pojmem typ. V SystemVerilogu je termín „datový typ“ definován výše uvedeným způsobem přesněji než ve Verilogu, pojem „typ“ pak slouží především k odlišení dvou zmíněných skupin signálů, tj. propojení a proměnných, jak o tom budeme podrobně mluvit v **odst. 2.3.1**. Často se však i tam a zejména v navazující literatuře tento termín používá volnějším způsobem.

Poznámka: Jak bylo řečeno výše, ve Verilogu se nerozlišují pojmy typ a datový typ. I v literatuře o SystemVerilogu se dosti často setkáme s nepřesným použitím těchto názvů, popř. s jejich záměnou. Svádí k tomu i snaha o stručné vyjadřování, kdy se slovo „datový“ zejména při opakovaném použití ne zcela správně vypouští, což platí i pro oficiální dokumenty, jako je například manuál [6]. Zde budeme většinou používat terminologii odpovídající standardu SystemVerilogu; jen někdy, zejména bude-li řeč pouze o Verilogu, případně pokud nebude nebezpečí záměny, vynecháme pro stručnost slovo „datový“.

Datové typy v SystemVerilogu mohou podle [6] být **singulární** (*singular*) nebo **agregátní** (*aggregate*). K agregátním datovým typům se řadí nesbalená pole, nesbalené struktury a nesbalené uniony; všechny ostatní datové typy jsou singulární. Proměnná nebo výraz singulárního typu představuje číselnou hodnotu, symbol nebo **úchyt** (*handle*). Agregátní proměnná nebo výraz je sada singulárních objektů. Singulární jsou všechny objekty integrálního typu (viz **odst. 2.3.5**), i když tyto objekty mohou být rozděleny do více singulárních objektů. Také řetězce představují singulární datový typ.

V jazycích Verilog i SystemVerilog mohou jednobitové signály, parametry a konstanty mít čtyři hodnoty:

- `0` (logická nula, nepravda, *false*);
- `1` (logická jednička, pravda, *true*);
- `x` nebo `X` (malé nebo velké písmeno – neznámá hodnota, *unknown*);
- `z` nebo `Z` (malé nebo velké písmeno, nebo i symbol `?` – vysoká impedance, *high-z*).

V SystemVerilogu jsou navíc zavedeny také dvouhodnotové datové typy s možnými hodnotami `0` a `1`. Propojení mohou být v SystemVerilogu jen čtyřhodnotová, zatímco proměnné mohou být dvouhodnotové i čtyřhodnotové. Nejpoužívanější datový typ `logic` je čtyřhodnotový. V anglické literatuře se místo o počtu hodnot datových objektů mluví častěji o počtu jejich stavů (*2-state*, resp. *4-state data object*), což však může působit nedorozumění při úvahách o stavových automatech. Zde budeme proto používat výše uvedenou terminologii.

Datové objekty s dvouhodnotovými datovými typy se používají pro modelování složitých systémů na vysoké úrovni abstrakce – na systémové či transakční úrovni. Simulace je pak méně náročná na výkon počítače a na rozsah jeho paměti než při použití čtyřhodnotových datových typů. Lze je použít i pro vytváření syntetizovatelných modelů na úrovni RTL, tam ale bývá užitečné mít k dispozici prostředky pro modelování nepřipojených vývodů nebo třístavových sběrnic (hodnota `z`) a pro detekci nedefinované hodnoty (`x`) – to může pomoci při hledání konstrukčních chyb (`x` je při simulaci výchozí hodnota čtyřhodnotového signálu, kterou signál má, pokud není jeho hodnota definována, což je ve výsledcích simulace dobře patrné, zatímco u dvouhodnotového signálu je výchozí hodnotou `0`). Zde jsou pak obvykle vhodnější čtyřhodnotové signály.

SystemVerilog dovoluje vzájemně přiřazovat hodnoty mezi dvouhodnotovými a čtyřhodnotovými datovými objekty. Při přiřazení hodnoty čtyřhodnotového datového objektu dvouhodnotovému se hodnoty `x` a `z` automaticky převedou na hodnotu `0`.

Hodnoty jednobitových dvouhodnotových datových objektů mohou být interpretovány jako **logické hodnoty** (logická nula, jednička), jako **číselné hodnoty** (binární číselná nula, jednička), nebo také jako **hodnoty pravdivostní** (nepravda nebo pravda); čtyřhodnotové datové objekty mohou navíc obsahovat také hodnotu nedefinovanou (`x`) nebo mohou představovat stav vysoké impedance (`z`). Datové objekty typu propojení mohou mít ještě přiřazenu **úroveň intenzity** (*strength level*) používanou při simulaci, o níž zde nebudeme podrobněji uvažovat. Při uvážení této možnosti se počet možných hodnot těchto jinak čtyřhodnotových datových objektů podle [11] zvyšuje na více než 120.

2.3.1 Propojení a proměnné

V tomto odstavci i v některých dalších (viz **odst. 2.1**) budeme pro lepší orientaci čtenáře psát to, co se týká pouze SystemVerilogu, hnědým písmem; to, co je platné i ve Verilogu, bude psáno černě.

Jak již bylo naznačeno, ve Verilogu i v SystemVerilogu rozeznáváme dvě hlavní skupiny – typy datových objektů: propojení a proměnné (konstanty a parametry zatím opomineme). V tomto odstavci budeme dále ve spojení „datový typ“ slovo „datový“ pro stručnost většinou vynechávat, jak je to také obvyklé v anglické literatuře, a budeme předpokládat, že čtenář má na paměti rozdíl mezi pojmy typ a datový typ.

Propojení (*net*) zhruba odpovídá fyzickému propojení mezi prvky v konstrukci (při syntéze však může být propojení eliminováno). Nejpoužívanější datový typ propojení je označován klíčovým slovem **`wire`**. Někdy se setkáme také s typem **`tri`**, který je synonymem typu **`wire`** a používá se při modelování třístavových sběrnic s větším počtem budičů – dvojí označení naznačuje účel propojení a pomáhá tak k lepší orientaci v konstrukci. Ve Verilogu-2005 je dále zaveden typ **`uwire`** (*unresolved* **`wire`**) – signály tohoto typu mohou být buzeny pouze jedním budičem, na rozdíl od signálů typu **`wire`**, kde může být budičů více, jinak je tento typ shodný s typem **`wire`**. Typ **`uwire`** tak vylučuje možnost vzniku chyby způsobené neúmyslným vícenásobným buzením téhož signálu. Ostatní typy propojení se používají v modelech určených k syntéze do obvodů PLD a FPGA zřídka a nebudeme je zde podrobně diskutovat. Typ **`wire`** je výchozí (*default*) typ signálů; signály bran, deklarované v seznamech bran modulů ve Verilogu, jsou automaticky považovány za propojení typu **`wire`**, není-li explicitně definován jejich typ jinak. V SystemVerilogu jsou pravidla pro stanovení typu takových signálů složitější, což je mimo jiné zřejmě dáno změnami v definici proměnných a snahou o kompatibilitu s Verilogem; pro naše účely však můžeme předpokládat, že výchozím typem signálů je rovněž propojení, zde však s datovým typem **`wire logic`**, s výjimkou výstupních bran, které v tomto případě mohou někdy představovat proměnné – viz **odst. 2.2.1.3**. U vnitřních signálů datového typu **`logic`** nebo **`bit`** deklarovaných jinde než v hlavičce, u nichž není v deklaraci explicitně uvedeno, jde-li o propojení nebo o proměnnou, se však v SystemVerilogu inferuje proměnná. V SystemVerilogu nejsou zavedeny až do doby psaní tohoto textu (do verze SystemVerilog-2012) žádné nové typy propojení (s výjimkou typu **`interconnect`**), používají se zde stejná propojení jako ve Verilogu; návrhář však může definovat uživatelské typy propojení, jak uvidíme dále.

Termín „propojení“ se v češtině zatím nepoužíval, a budeme proto často uvádět i bližší vysvětlení nebo jeho anglický název (*net* – obvykle v závorce).

Signály typu propojení mohou být v behaviorálním popisu buzeny pouze výstupy budičů, které jsou zde modelovány kontinuálními přiřazovacími příkazy – to jsou příkazy s klíčovým slovem **`assign`**, viz **odst. 2.5.1**

(vstupní brány jednotek představují v tomto smyslu také kontinuální přiřazení). Ve strukturálním popisu mohou být buzeny výstupy hradel (**odst. 2.8**) či komponent, a mohou zde sloužit k propojování bran jednotek (ve Verilogu bran modulů, v SystemVerilogu také rozhraní) – v tomto smyslu představuje brána budič analogický kontinuálnímu přiřazení. Ve Verilogu ani v SystemVerilogu nemůže být signálům typu propojení (např. `wire`) přiřazena hodnota v tzv. **procedurálním kódu** (**odst. 2.6**), tedy v blocích skupiny `always` nebo `initial`, ve funkcích a v úlohách – takové přiřazení je možné jen pro proměnné. V manuálech **[1]**, **[6]** se uvádí, že propojení nemůže uchovávat hodnotu, ta je dána budičem, k němuž je propojení připojeno.

Jak jsme již uvedli, signály typu propojení mohou být buzeny větším počtem budičů. V takovém případě je výsledná hodnota dána tzv. **rozhodovací funkcí** (*resolution function*; někdy se o ní také mluví jako o rozlišovací funkci – o tom se ještě zmíníme v dalším textu, a také v **odst. 2.5.1**). V našich předpokládaných 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 signálů (signálů na výstupech kombinačních obvodů) a klíčové slovo **reg** pro registrové signály (na výstupech paměťových prvků), což je přípustné, původně však nezamýšlené.

Ve Verilogu (i v SystemVerilogu) se používají také proměnné datového typu **integer**, **real**, **time** a **realtime**; nově jsou v SystemVerilogu zavedeny pro proměnné dvouhodnotové datové typy **bit**, **byte**, **shortint**, **int**, **longint**, a datový typ **shortreal** obdobný typu *float* jazyka C. Snahou je přiblížit datové typy jazyku C, což má usnadnit převod algoritmů z tohoto jazyka do SystemVerilogu a naopak. Proměnná datového typu **shortint** má 16 bitů, **longint** má 64 bitů. Proměnná datového typu **integer** má 32 bitů, což platí i pro proměnnou datového typu **int** v SystemVerilogu. Proměnná datového typu **byte** je osmibitová; může představovat také znak ASCII. Bitová šířka proměnných s datovými typy **bit** a **logic** v SystemVerilogu je volitelná, je dána jejich deklarací; totéž platí ve Verilogu pro typ **reg**. Datový typ **bit** (dvouhodnotový) je analogický čtyřhodnotovému datovému typu **logic**, rozdíl je v počtu hodnot. Jak již bylo uvedeno, propojení v SystemVerilogu mohou být jen čtyřhodnotová.

Jak naznačuje název, objekty datového typu **integer** i **int** jsou celočíselné proměnné. Jako všechny typy v jazyku Verilog je proměnná datového typu **integer** čtyřhodnotová, naproti tomu proměnná datového typu **int** (SystemVerilog) je dvouhodnotová.

V textech určených k syntéze je vhodné používat proměnné s datovým typem **logic**, popř. **reg**. Výjimkou jsou případy proměnných, kdy syntézou přímo nevznikají obvodové struktury, jako například indexy bitů vektorů a prvků polí nebo řídící proměnné smyček, kde se používají typy **integer** nebo **int**. **Přehled datových typů a jejich vlastností je uveden v odst. 2.3.5.**

Je-li v SystemVerilogu deklarován vnitřní signál pouze svým datovým typem, bez bližší specifikace toho, zda jde o propojení nebo proměnnou, je tento signál považován za proměnnou. Například:

```
logic SigVar;  // SigVar je promenna
```

Skutečnost, že jde o proměnnou, může být v deklaraci zdůrazněna klíčovým slovem **var**. Toto zdůraznění může sloužit pro jasnější vyjádření typu, obvykle se však neuvádí:

```
var logic SigVar;  // SigVar je opet promenna
```

Na druhé straně však lze i propojení deklarovat jako datový typ **logic**:

```
wire logic SigNet;  // SigNet je propojeni s datovym typem logic
```

Je-li naopak v SystemVerilogu deklarován signál pouze jako propojení s deklarací **wire**, **tri** apod., je u něj inferován datový typ **logic**. Například deklarace

```
wire w;
```

je ekvivalentní deklaraci

```
wire logic w;
```

Signál výstupní brány (**output**) typu propojení (zpravidla **wire** nebo **tri**) není v konstrukci dostupný – lze mu jen přiřadit hodnotu, nemůže však být použit například pro vytváření výrazů. Je-li však deklarován jako proměnná – ve Verilogu obvykle klíčovým slovem **reg**, v SystemVerilogu klíčovým slovem **logic**, je dostupný i uvnitř konstrukce.

Propojení (ne však proměnná) může mít v SystemVerilogu (nikoli ve Verilogu) více názvů, které se **ztotožní** použitím klíčového slova `alias`:

```
wire Clock;
wire Clk;
alias Clock = Clk;
```

Ztotožňované názvy nemusí být předem oba deklarovány. Stačí, když je deklarován jeden z nich, a druhý se pak deklaruje implicitně. Ztotožněním je možno také označovat samostatným názvem části vektorů. V následujícím příkladu tak identifikátory `HiByte`, `LoByte` představují osmibitové vektory datového typu `wire`:

```
wire [15:0] Data;
alias HiByte = Data[15:8];
alias LoByte = Data[7:0];
```

Jak bylo uvedeno výše, ve Verilogu může být hodnota proměnné (obvykle typu `reg`) měněna pouze v procedurálním kódu. Pro zápis procedurálního kódu, kterým se má přiřazovat hodnota výstupnímu signálu modulu, použijeme tedy ve Verilogu jeden z následujících dvou přístupů:

- deklarujeme signál výstupní brány jako typ `reg` a přiřadíme mu procedurálním kódem hodnotu přímo;
- pokud je z nějakých důvodů potřebné, aby tento signál byl typu propojení (například je-li to signál obousměrné brány – obvykle typu `wire` nebo `tri`), deklarujeme vnitřní proměnnou typu `reg` a přiřadíme jí hodnotu v procedurálním kódu, a signálu brány pak přiřadíme hodnotu této proměnné příkazem `assign`. Tímto způsobem můžeme také obejít omezení pro výstupní signály typu `wire`, které potřebujeme mít dostupné v konstrukci.

V SystemVerilogu může být výstupní signál modulu deklarován jako datový typ `logic` bez ohledu na způsob přiřazení hodnoty tomuto signálu. Podle **odst. 2.2.1.3** pak bude tento signál automaticky považován za proměnnou. Není-li však v deklaraci výstupní brány datový typ uveden, jde o implicitní datový typ a je inferováno propojení.

Ve Verilogu lze přiřadit hodnotu proměnné procedurálním kódem, tj. v bloku `always` nebo `initial` (o nich budeme mluvit podrobněji v **odst. 2.6**); jedné proměnné je syntakticky přípustné přiřadit tímto způsobem hodnotu i ve více takových blocích, doporučuje se však tuto možnost přiřazení hodnoty proměnné ve více blocích nevyužívat.

V SystemVerilogu může být proměnné přiřazena hodnota jedním (jen jedním) z následujících způsobů (všechny dále uvedené bloky budou blíže vysvětleny v **odst. 2.6**):

- stejně jako ve Verilogu přiřazením hodnoty v jakémkoliv počtu bloků `always` nebo `initial` (opět se v běžných případech doporučuje nevyužívat možnost přiřazení hodnoty jedné proměnné ve více blocích, i když je to syntakticky přípustné);
- přiřazením v jediném z bloků `always_comb`, `always_ff` nebo `always_latch`;
- přiřazením v jediném kontinuálním přiřazovacím příkazu – příkazem `assign`, **odst. 2.5.1** (ve Verilogu lze v takových příkazech přiřazovat hodnotu jen propojením);
- proměnná může získat hodnotu také z jediné výstupní nebo obousměrné brány vložené komponenty nebo primitivy (ve Verilogu může takto získat hodnotu pouze propojení);
- proměnná může v SystemVerilogu být také vstupním signálem, tedy signálem vstupní brány (`input`) jednotky (modulu, rozhraní), na rozdíl od Verilogu, kde vstupním signálem může být jen propojení (připojení signálu k bráně má charakter kontinuálního přiřazovacího příkazu).

V bloku `always` nebo `initial` může být ve Verilogu i v SystemVerilogu jedné proměnné přiřazena hodnota ve více procedurálních příkazech. Výsledná hodnota, kterou proměnná v bloku získá, je pak dána posledním provedeným přiřazením. Totéž platí v SystemVerilogu i o blocích `always_comb`, `always_ff` a `always_latch`.

Tato pravidla dovolují v SystemVerilogu deklarovat většinu signálů, vyjma signálů buzených z více budičů a signálů obousměrných bran, jako proměnné (obvykle typu `logic`) bez ohledu na to, jak se jim v modelu přiřazuje hodnota. Proměnné však lze ve Verilogu i v SystemVerilogu přiřadit hodnotu pouze jedním budičem – tím může být blok `always`, v němž je proměnné přiřazena hodnota, v SystemVerilogu to může být i kontinuální přiřazení či výstupní brána vložené komponenty. Proměnná také nemůže být použita pro deklaraci obousměrné brány. Tato omezení pomáhají odhalit konstrukční chyby, pokud se proměnné neúmyslně přiřadí signál z více budičů. Výjimkou je ve Verilogu i v SystemVerilogu možnost přiřadit proměnné hodnotu ve více blocích `always`. To je v SystemVerilogu dovoleno kvůli kompatibilitě s Verilogem. Doporučuje se proto používat v SystemVerilogu místo bloků `always` bloky `always_comb`, `always_ff` a `always_latch` – přiřazení hodnoty téže proměnné ve více těchto blocích představuje syntaktickou chybu.

V SystemVerilogu je dále pro proměnné zaveden typ `void`, který představuje neexistující data a může být specifikován jako návratový typ funkcí, které nevracejí číselnou hodnotu, a může také být použit pro členy označených (`tagged`) unionů.

Statické a automatické proměnné. Ve Verilogu-95 jsou všechny proměnné statické, což znamená, že jsou při simulaci uloženy v definovaném místě paměti počítače a během celé simulace zůstává jejich hodnota zachována. Verilog-2001 zavádí pojem **automatických proměnných** (říká se jim také **dynamické proměnné**) – ty mohou být deklarovány ve funkcích a v úlohách. Tyto proměnné jsou vždy znovu alokovány a inicializovány při volání, a automaticky dealokovány při ukončení funkce nebo úlohy, přičemž se jejich hodnota ztrácí. Automatické proměnné byly zavedeny především pro účely verifikace, a je s nimi možno vytvářet rekurzivní funkce (tj. funkce, které mohou volat samy sebe) a reentrantní úlohy (tj. úlohy, které mohou být volány ve více současných voláních, například v paralelních blocích, nebo mohou být volány v době, kdy jejich předchozí volání ještě běží). Ve Verilogu-2001 může být funkce nebo úloha deklarována jako automatická (viz **odst. 2.7**), a pak proměnné v ní deklarované jsou automatické.

V SystemVerilogu mohou být proměnné explicitně deklarovány jako statické nebo automatické s použitím klíčových slov `static` nebo `automatic`. Automatické proměnné mohou být deklarovány ve funkcích, v úlohách a v blocích `begin-end` nebo `fork-join`. Proměnné deklarované na úrovni modulu jsou vždy statické a nemohou být explicitně deklarovány jako statické nebo automatické. Nejsou-li proměnné deklarované ve funkcích a úlohách explicitně deklarovány jako statické nebo automatické, jsou ve statických funkcích a úlohách statické a v automatických funkcích a úlohách automatické.

Konstanty (*constants*) jsou podle **[1]**, **[6]** pojmenované proměnné, které se nemohou měnit. Verilog pro ně zavedl konstrukty `parameter` a `localparam`, v SystemVerilogu je přidán konstrukt `const`. Podrobnější řeč o nich bude později – **odst. 2.3.10**. Dalším takovým konstruktem je `specparam`, který slouží při verifikaci pro zadávání údajů o čase a o zpožděních; zde o něm nebudeme podrobněji mluvit.

Doporučení pro volbu statických a automatických proměnných v SystemVerilogu podle **[11]**:

Rekurzivní funkce a reentrantní úlohy musí být automatické. Také všechny proměnné v nich mají být automatické, není-li zvláštní důvod, aby jejich hodnoty byly zachovány z jednoho volání do druhého (například proměnná počítající počet volání funkce či úlohy při simulaci musí být statická).

V blocích skupiny `always` nebo `initial` je vhodné užití statických proměnných, nejsou-li tyto proměnné inicializovány v deklaraci, v opačném případě jsou vhodnější automatické proměnné.

Funkce a úlohy, které představují při syntéze hardwarovou jednotku, nejsou rekurzivní či reentrantní a mají být statické, a všechny proměnné v nich mají také být statické.

Automatickým proměnným nelze přiřadit hodnotu kontinuálním přiřazením nebo výstupem modulu, protože automatické proměnné nemusí existovat v celém průběhu simulace, což je v takovém případě nutné.

2.3.2 Deklarace signálů a jejich výchozí hodnoty

Signály by měly být deklarovány, než jsou použity. Deklarovány mohou být v hlavičce – v seznamu bran (signály bran) nebo uvnitř popisu jednotky (vnitřní signály). Deklarace bran a jejich signálů byly prodiskutovány v **odst. 2.2.1** a dále. Syntaxi **deklarací vnitřních signálů** uvedeme pro první seznámení ve formě příkladů (ve Verilogu bude místo datového typu `logic` typ `reg`):

```
wire ident, ..., ident;
wire [msb:lsb] ident, ..., ident;

logic ident, ..., ident;
logic [msb:lsb] ident, ..., ident;

integer ident, ..., ident;
```

Zde *ident* znamená identifikátor signálu, *msb* a *lsb* jsou indexy nejvýznamnějšího a nejméně významného bitu v deklaraci jednorozměrného vektorového signálu. V deklaraci signálu může být uvedena i znaménkovost signálu (například `signed`, viz **odst. 2.3.5**); v **odst. 2.5.1** uvidíme, že deklarace propojení může obsahovat i kontinuální přiřazení, a v deklaraci proměnné může být uvedena její počáteční hodnota.

Ve Verilogu i v SystemVerilogu představuje použití nedeklarovaného identifikátoru v procedurálním příkazu nebo na pravé straně kontinuálního přiřazovacího příkazu (vysvětlení těchto pojmů bude uvedeno později, viz **odst. 2.5.1**) syntaktickou chybu. V obou těchto jazycích je však dovoleno použití takových identifkátorů na levé straně kontinuálních přiřazovacích příkazů a ve strukturálních popisech na pozici signálu, který je ve vložení komponenty připojen k bráně (příklad je uveden dále, `TXT2.2`). Pak jde o tzv. **implicitní deklarace** propojení. Jejich účelem je usnadnit popis složitých strukturálních modelů s vloženými kontinuálními přiřazovacími příkazy. Zde implicitní deklarace dovolují vynechat zápis explicitních deklarací propojovacích signálů, což může popis výrazně zestručnit. Kompilátor automaticky inferuje propojení a přiřadí příslušným identifikátorům datový typ `wire`. To však může vést k obtížně odhalitelným chybám. Těžko se zejména hledají chyby způsobené překlepem v identifikátorech na těchto pozicích – ty nejsou kompilátorem vyhodnoceny jako chybné. Mezi nimi je obzvláštní lahůdkou záměna písmene `O` a číslice `0` nebo písmene `l` a číslice `1` – tyto dvojice jsou v některých typech písma téměř nerozeznatelné. Bezchybnému zápisu takových identifikátorů je proto vhodné věnovat zvýšenou pozornost. Implicitní deklarace je možno ve Verilogu i v SystemVerilogu zakázat direktivou `` `default_nettype none ``. Názory na vhodnost použití implicitních deklarací i této direktivy se u jednotlivých autorů různí, viz např. **[35]**. Při použití direktivy je nutno si uvědomit, že její účinek není omezen na soubor, v němž je umístěna, ale působí na práci kompilátoru až do okamžiku, kdy je zrušena jinou direktivou definující výchozí typ. Doporučuje se proto ukončit její působnost v témže souboru (nebo raději v téže jednotce) direktivou `` `default_nettype wire ``, čímž se obnoví výchozí nastavení. Místo složitějších strukturálních popisů (nejsou-li generovány automaticky) může proto být lepší volbou (odolnější vůči překlepům) popis schématem. To však přestává být schůdné u konstrukcí s desítkami a větším počtem bran. Zejména pro takové konstrukce (ale nejen pro ně) zavádí SystemVerilog konstrukt **rozhraní** (`interface`) a významná zjednodušení v zápisu připojení bran vkládaných komponent – implicitní připojení bran, což odstraňuje nutnost opakovaného vypisování stejných identifikátorů v textech psaných ve Verilogu, a automatická kontrola zápisu omezuje nebezpečí překlepů. To bude podrobněji popsáno dále (**odst. 2.8.6**).

Jako příklad chyby maskované implicitní deklarací uvedeme zkrácený popis převodníku binárního kódu na kód BCD, do nějž je vložena sčítačka pracující v kódu BCD:

```
module AddBCD(input  logic [3:0] Ain,Bin, input  logic Cin,    // TXT2.2
              output logic [3:0] Sout,    output logic Cout);  // V,SV
  ...   // popis scitacky
endmodule
```

```
module Bin2BCDstruct5bErr(input  logic [4:0] BinIn,            // V*,SV
                          output logic [1:0][3:0] BCDout); //sbalene pole
  logic [3:0] S0out,BCD1out,BCD0out;
  logic C0out,Ca1H;
  wire [3:0] Aa0 = {1'b0,BinIn[2:0]},  // prirazeni v deklaraci propojeni
             Ba0 = {BinIn[3],3'b0},    // {...} znaci sjednoceni
    ... // dalsi deklarace
  AddBCD St0 (.Ain(Aa0),.Bin(Ba0),.Cin(1'b0),.Sout(S0out),.Cout(COout)),
    ... // popis dalsich stupnu a vystupniho signalu
endmodule
```

V tomto textu se vyskytují některé konstrukty, které budou blíže vysvětleny později, a není nezbytné studovat nyní jejich přesný význam: vektory (odst. 2.3.3), sbalené pole (odst. 2.3.11.1), sjednocení (odst. 2.4.7) a vložení komponenty (odst. 2.8.2). Uvedený text představuje upravený popis převodníku z kapitoly 5 (`TXT5.1.5d`), kde je ve vložení komponenty sčítačky `AddBCD` záměrně vytvořen překlep – je zde k bráně `Cout` připojen signál `COout`, kde místo číslice `0` je písmeno `O` (zvýrazněno tučným písmem). Při zpracování tohoto textu systémem Quartus se sice objeví varování, to ale lze snadno přehlédnout. Není také zaručeno, že podobné varování budou dávat i jiné systémy. Výsledky simulace i syntézy pak samozřejmě neodpovídají očekávání. Ve složitějších případech nemusí být snadné chybu tohoto druhu odhalit.

Bitová šířka signálu vytvořeného implicitní deklarací, který je připojen k bráně modulu, v němž byl vytvořen, odpovídá bitové šířce této brány. Jestliže však takový signál propojuje jen komponenty vložené do modulu, inferuje se pouze jednobitový signál bez ohledu na šířku bran komponent, které propojuje. To rovněž může být zdrojem problémů.

Výchozí (počáteční, *default*) hodnoty signálů: ve Verilogu i v SystemVerilogu mají signály typu **propojení** při simulaci, pokud jim není explicitně přiřazena hodnota, výchozí hodnotu `z` (vektory hodnotu `'bz`, tj. všechny bity vektoru mají hodnotu `z`; význam tohoto zápisu bude podrobněji vysvětlen v odst. 2.3.9). Výjimkou jsou propojení typu **`trireg`**, jejichž výchozí hodnota je `'bx`, o propojeních tohoto typu však zde nebudeme podrobněji uvažovat. Je-li však propojení připojeno na výstup hradla nebo komponenty, získává výchozí hodnotu odtud. Výchozí hodnotou explicitně neinicializovaných **proměnných** typu **`reg`**, **`time`** a **`integer`** ve Verilogu je hodnota `'bx`. Proměnné mohou být explicitně inicializovány přiřazením hodnoty v bloku **`initial`** (to byla jediná možnost ve verzi Verilog-95) nebo přiřazením v deklaraci – viz odst. 2.5.1. V SystemVerilogu mají integrální proměnné se čtyřhodnotovými typy také výchozí hodnoty `'bx`, integrální proměnné s dvouhodnotovými typy mají výchozí hodnoty nulové, tj. `'b0` (zhruba řečeno, integrální typy jsou typy používané pro práci s celočíselnými veličinami, které představují vektory, o nichž bude řeč v odst. 2.3.3; podrobněji bude pojem integrálních typů vysvětlen v odst. 2.3.5).

Inicializace proměnných může být důležitá v modelech určených k syntéze zejména u proměnných představujících obsah paměťových prvků – syntetizéry často hodnotu uvedenou v inicializačním příkazu interpretují jako požadavek na obsah prvku po připojení napájecího napětí (*power-up state*), a pokud to cílový obvod dovoluje, nakonfigurují jej příslušným způsobem. Další informace o inicializaci paměťových prvků budou uvedeny v odst. 2.6.10.

2.3.3 Skalární a vektorové signály

Deklarace signálu bez zapsaného rozsahu specifikuje **skalární**, tj. jednobitový signál. U v**ektorového signálu** (často se mluví také o **bitových vektorech**) je každému jeho bitu přiřazen **index**. Vektorový signál se deklaruje uvedením **rozsahu** (*range*) indexů jeho bitů v hranatých závorkách před jeho názvem (viz následující deklarace signálů `Sig1Dim` a `Sig2Dim`). Označením **bitová šířka** (*size*) vektoru se rozumí počet jeho bitů. Skalární signál lze v určitých situacích považovat i za vektor s jednobitovou šířkou (viz např. odst. 2.3.8). Vektorový charakter mohou mít kromě signálů i další objekty, jako jsou parametry a konstanty – o těch více v odst. 2.3.10. Ve Verilogu mohou být vektory jen jednorozměrné, a typy vektorových objektů mohou být propojení (obvykle **`wire`**) nebo proměnné (**`reg`**), případně parametry. V SystemVerilogu mohou vektory být i vícerozměrné – pak je rozsah uveden u jednoho vektoru vícekrát, a datový typ objektů může být i **`logic`** (nejčastější) nebo **`bit`**; o vektorech se tam mluví také jako o **sbalených polích** (o polích bude podrobnější řeč v odst. 2.3.11). Jak jsme již uvedli, rozsahy sbalených rozměrů jsou součástí datového typu. Hranicemi rozsahů

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`.

Terminologie, kterou používáme, vychází z verze SystemVerilog-2009 [5], kde je zavedena (a převzata do verze 2012) nová, od starší verze poněkud odlišná terminologie: bitový výběr s konstantním indexem vybíraného bitu se zde nazývá *constant bit-select* (**konstantní bitový výběr**), neindexovaný částečný výběr se zde nazývá *non-indexed part-select*. Termínem *constant part-select* (**konstantní částečný výběr**) se zde označuje neindexovaný částečný výběr nebo indexovaný částečný výběr, kde je výraz pro bázi konstantní (výraz pro šířku indexovaného částečného výběru je vždy konstantní).

Vektory mohou být také vytvořeny **sjednocením** (*concatenation*) výrazů dávajících bitový nebo vektorový výsledek, které může být kombinováno s **replikací** (*replication*). Podrobnější diskuse o těchto operacích a příslušných operátorech bude v odst. 2.4.7.

V SystemVerilogu je zaveden pojem **sbalených polí** (odst. 2.3.11.1), jichž jsou vektory zvláštním případem.

Důležitá jsou také pravidla pro přiřazování hodnot vektorovým signálům v přiřazovacích příkazech. Podrobněji o nich budeme mluvit v odst. 2.5.

2.3.4 Kompatibilita typů

Tradiční Verilog je označován jako jazyk s volnými typy (**volně typový jazyk**, *loosely typed language*), což znamená, že hodnota signálu jednoho typu může být přiřazena signálu odlišného typu, přičemž je jeho hodnota automaticky převedena do nového typu podle pravidel standardu Verilogu. Některé konstrukty jazyka však v takovém případě vyžadují určitou úroveň kompatibility typů zpracovávaných dat.

Ve standardu SystemVerilogu [6] je tento rys jazyka zpřesněn zavedením pěti kategorií (úrovní) pro kompatibilitu typů. Podle nich může být vzájemný vztah typů následující:

- typy **shodné** (*matching*),
- typy **ekvivalentní** (*equivalent*),
- typy **kompatibilní pro přiřazení** (*assignment compatible*),
- typy **kompatibilní pro převod** (používá se také označení **kompatibilní pro změnu**, typy **změnově kompatibilní** – *cast compatible*),
- typy **nekompatibilní** (*incompatible*).

Přesná definice těchto úrovní kompatibility uvedená v [6] pro **shodné** a **ekvivalentní** typy je dosti složitá a rozsáhlá a nebudeme ji zde podrobně rozebírat. Jen jako příklady uvedeme, že ke shodným typům patří všechny zabudované typy se stejným názvem, a také uživatelské typy definované v téže definici pomocí klíčového slova `typedef` v rozsahu platnosti této definice. Ekvivalentní jsou pak zejména všechny shodné typy a také výčtové anonymní typy (definované klíčovým slovem `enum` – viz odst. 2.3.7), a to u datových objektů, které jsou deklarovány v tomtéž příkazu deklarace.

Pro typy **kompatibilní pro přiřazení** platí, že to jsou všechny ekvivalentní typy a také neekvivalentní typy, pro které jsou definována implicitní pravidla převodu typu. Například všechny integrální typy (odst. 2.3.5) jsou vzájemně kompatibilní pro přiřazení. Tato kompatibilita může být jednosměrná – například objekt výčtového typu může být převeden na integrální typ bez explicitního převodu typu, obráceně to však neplatí. Při převodu může docházet ke ztrátě informace vlivem zkrácení (oříznutí) nebo zaokrouhlení dat (k zaokrouhlení může docházet například při převodu reálných typů na celočíselné).

Kompatibilní pro převod jsou všechny typy kompatibilní pro přiřazení, a také všechny neekvivalentní typy, které mají definována explicitní pravidla pro převod typu – viz odst. 2.3.8. Například objekty s integrálními typy je možno převést na výčtový typ použitím explicitního převodu typu.

Typy, jejichž vztah neodpovídá výše uvedeným úrovním kompatibility, tedy neekvivalentní typy, které nemají definována implicitní ani explicitní pravidla pro vzájemný převod, se pokládají za **vzájemně nekompatibilní**.

V manuálu [6] se mluví i o typech **totožných** (*identical*), například typ `int` je totožný s uživatelským typem definovaným jako `bit signed [31:0]`; tato úroveň kompatibility však není nikde v SystemVerilogu vyžadována, a proto není zvlášť uvedena.

2.3.5 Celočíselné datové typy

Jak již bylo naznačeno v odst. 2.3.1, jsou ve Verilogu zavedeny **celočíselné datové typy** (*integer data types*) proměnných: `reg`, `integer` a `time`. V SystemVerilogu jsou tyto typy rozšířeny o datové typy `shortint`, `int`, `longint`, `bit`, `byte`, `logic`. (Za neceločíselné datové typy se považují typy `shortreal`, `real` a `realtime`.) Standard SystemVerilogu dále zavádí pojem **integrálního typu** (*integral type*), čímž se rozumí datové typy, které mohou představovat některý z těchto základních celočíselných datových typů, dále sbalená pole, sbalené struktury, sbalené uniony, výčtové typy, a pojem **typu jednoduchého bitového vektoru** (*simple bit vector type*), o němž se blíže zmíníme v odst. 2.3.11.1.

Objekty s celočíselnými datovými typy mohou představovat čísla **se znaménkem** (`signed`) nebo **bez znaménka** (`unsigned`) – tomu říkáme **znaménkovost** (*signedness*). Informace o znaménkovosti je součástí datového typu. V dosud uvedených příkladech byly vektory chápány jako čísla bez znaménka. Vektory představující proměnné, propojení nebo výstupy funkcí mohou být deklarovány jako čísla se znaménkem tak, že se deklarace doplní klíčovým slovem (modifikátorem) `signed` zapsaným za specifikací typu. Takto označené operandy se ve výrazech mohou za jistých podmínek zpracovávat podle pravidel aritmetiky čísel se znaménkem, jak bude podrobněji rozebráno v odst. 2.4.9. Deklarace takových vnitřních proměnných (o deklaraci bran byla řeč v odst. 2.2.1 a dále) může mít tvar:

```
logic signed [msb:lsb] ident, ..., ident;
```

Výchozím stavem (*default*) pro objekty datových typů `shortint`, `int`, `longint`, `integer` a `byte` je typ `signed`, pro objekty datových typů `bit`, `logic` a `reg` je to typ `unsigned`; totéž platí pro pole těchto typů. Lze je však pomocí modifikátorů `signed`, `unsigned` předeklarovat; na rozdíl od jazyka C zde musí být modifikátor uveden až za označením datového typu. Na objekty základních celočíselných datových typů **lze pohlížet i jako na pole**, jak bude blíže uvedeno později (viz jednoduché bitové vektory, odst. 2.3.11.1; příklad takové interpretace v přiřazovacím vzoru bude uveden v odst. 2.5.3).

Pro převod vektorů typu `signed` na typ `unsigned` a naopak (*type casting*) slouží **systémové konverzní funkce** `$unsigned`, `$signed`. Je zde potřebné rozlišovat samotný převod a případné následné přiřazení výsledku jinému vektoru. Při převodu se bitový vzor argumentu funkce nemění, mění se však typ (interpretace) argumentu a způsob jeho následného zpracování v aritmetickém výrazu. Bitový vzor výsledku se pak může měnit při jeho přiřazení signálu s jinou bitovou šířkou v souhlasu s pravidly uvedenými v odst. 2.4.9. Příklad popisu sčítačky a násobičky čísel se znaménkem a použití konverzních funkcí je v odst. 5.1.8.

SystemVerilog má pro převod typů ještě další prostředek – explicitní převod typu, který je ekvivalentní uvedeným systémovým konverzním funkcím; o tom bude více v odst. 2.3.8.

Proměnné typu `integer` nebo `int` se nejčastěji používají jako řídící proměnné smyček nebo ve výrazech vyjadřujících indexy. Má-li být taková proměnná syntetizována, modeluje se obvykle jako vektor s šířkou 32 bitů. V textech určených k syntéze bývá proto většinou vhodné vyhnout se použití takových proměnných pro 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