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**

neodpovídá osminásobku počtu znaků v řetězci.

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

V SystemVerilogu je dále definován datový typ **string** určený především k usnadnění práce s řetězci. Bitová šířka proměnné typu **string** se stanoví dynamicky tak, aby proměnná mohla obsáhnout jakýkoliv přiřazovaný řetězec. Při přiřazení zde tedy nedochází k oříznutí ani k doplnění řetězce. Řetězcové literály jsou při přiřazení takové proměnné implicitně převedeny na typ **string**, a totéž platí, jsou-li užity ve výrazech obsahujících operandy typu **string**. SystemVerilog obsahuje také řadu metod pro práci s řetězci se syntaxí podobnou metodám užívaným u výčtových typů (odst. 2.3.7).

Zpracování řetězců najde své uplatnění zejména při simulaci, například při organizaci výpisů. Další podrobnosti o řetězcích a o operacích s nimi najde zájemce v manuálech [1], [6].

Literály pro zápis hodnot polí, stručněji **literály polí** (*array literals*) v SystemVerilogu patří k přiřazovacím vzorům pro pole (odst. 2.3.11.5), které však obsahují jen konstanty a/nebo konstantní výrazy. Jsou zapisovány do složených závorek, přičemž první závorku předchází apostrof, a položky v nich jsou odděleny čárkami. Jsou syntakticky podobné inicializátorům v jazyku C, v SystemVerilogu je však u nich dovoleno použití replikačních operátorů (odst. 2.4.7). Používají se pro přiřazování konstantních hodnot polím a pro jejich inicializaci. Příklad inicializace pole při jeho deklaraci s použitím literálu polí:

```
shortint Arrn[1:2][1:3] = ,{,{2,1,0},'{3{2}}};          // jen SV
```

Tento zápis je ekvivalentní zápisu bez použití replikace:

```
shortint Arrn[1:2][1:3] = ,{,{2,1,0},'{2,2,2}};          // jen SV
```

Vnořené složené závorky musí odpovídat struktuře pole. Není přípustné vynechat vnitřní závorky s apostrofem, jako je to možné v jazyku C. Literál polí musí mít přiřazen typ – ten může být dán kontextem přiřazení (odst. 2.5.2), jako je to ve výše uvedených příkladech, kde je typ literálu dán typem definovaným na levé straně příkazu, nebo může být uveden explicitně jako prefix (před apostrofem). Uvažujme jako příklad následující fragment textu:

```
typedef shortint Arr3nums[1:3];                          // jen SV
initial begin
  ...
  $display(„%p",Arr3nums'{0,1,2});
end
```

Formátovací posloupnost „%p" zde specifikuje přiřazovací vzor; o systémové úloze *$display* bude více řečeno v odst. 2.7. Při simulaci se na konzolu vypíše `,{0, 1, 2}`.

V literálech polí mohou být použity také klíče – index, typ nebo klíčové slovo **default**, jak bude podrobněji vysvětleno v odst. 2.3.11.5.

Strukturní literály (*structure literals*) jsou přiřazovací vzory pro struktury (odst. 3.1.2) obsahující konstanty a/nebo konstantní výrazy, které lze použít pro přiřazování konstantních hodnot strukturám a k jejich inicializaci – o strukturách si povíme více v odst. 3.1. Strukturní literál musí stejně jako literál polí mít přiřazen typ, který může být uveden explicitně jako prefix, nebo může být dán kontextem přiřazení – například:

```
typedef struct {shortint v1; shortreal v2;} v12_t;      // jen SV
v12_t a_st;       // struktura a_st je typu v12_t
a_st = '{1, 2.0}; // typ literalu je urcen kontextem podle leve strany
```

Při zápisu literálu v případě pole struktur musí být vzory přiřazované prvkům pole uzavřeny ve složených závorkách, jako je to v následujícím příkladu inicializace pole `v12arr`:

```
v12_t v12arr[1:0] = ,{,{2, 1.0},'{2, 2.0}};             // jen SV
```

Také zde není přípustné vynechat vnitřní složené závorky s apostrofem. Podobně jako u literálů polí je zde možno použít replikace a klíče, jak bude podrobněji uvedeno v odst. 3.1.2.

Mezi literály se řadí také **zápisy reálných konstant**, o nichž zde nebudeme podrobněji diskutovat, a literály vyjadřující **časové údaje**, které se zapisují jako celá čísla nebo čísla ve formátu pevné řádové tečky (*fixed-point format*), následovaná bez mezery symbolem časové jednotky – `fs`, `ps`, `ns`, `us`, `ms` nebo `s`, například `20ns` nebo `5.5ps`.

2.3.10 Parametry a konstanty

Místo zápisu konstantních číselných hodnot ve formě literálů je ve zdrojových textech často vhodnější používat **parametry** (*parameters*) – symbolické označení těchto konstant. Parametry jsou deklarovány klíčovými slovy **parameter** a **localparam** (nebudeme zde diskutovat tzv. specifikační parametry – **specparam** používané ke specifikaci časových údajů a zpoždění). Parametry jsou inicializovány během fáze zpracování modelu zvané elaborace, během níž se sestavuje hierarchická struktura modelu, která v této fázi ještě nemusí být k dispozici (o elaboraci bylo více uvedeno v odst. 2.1). Jejich hodnota proto nemůže být odvozena z jiných částí hierarchie. Příručka [6] o nich mluví také jako o **parametrických konstantách** (*parameter constants*), na rozdíl od konstant deklarovaných klíčovým slovem **const**, o nichž bude řeč na konci tohoto odstavce; zde ale zůstaneme u původního stručnějšího označení z Verilogu – parametry.

Parametry mohou být deklarovány v modulech, ve statických funkcích a úlohách. V SystemVerilogu mohou být deklarovány také v rozhraních a v programech. Nemohou být deklarovány v automatických funkcích a úlohách, v blocích **begin-end** a **fork-join**.

Deklarace parametru může obsahovat specifikaci typu a/nebo rozsahu parametru. Podle manuálů [1], [6] zde platí následující pravidla (typem se zde zřejmě rozumí i znaménkovost – **signed**, **unsigned**):

- není-li typ ani rozsah specifikován, jsou tyto atributy dány hodnotou, která je parametru v deklaraci přiřazena. Je-li přiřazovaná hodnota typu

Pro syntézu má význam především inicializace paměťových prvků. Tyto prvky musí být reprezentovány proměnnými, takže nemohou být inicializovány kontinuálním přiřazením. Behaviorálně jim lze přiřazovat hodnoty jen v procedurálních blocích, jak bylo naznačeno v textu `TXT2.8`. Podrobněji se inicializací paměťových prvků budeme zabývat v odst. 2.6.10.

2.3.11.6 SystemVerilog: Operace s poli

Jak již bylo uvedeno v odst. 2.3.11.5, říkáme o polích, že mají stejné uspořádání, mají-li stejný počet rozměrů a jsou-li stejné jejich rozsahy; nemusí však souhlasit takové atributy jako je vzestupnost či sestupnost indexování a hranice indexů. Jsou-li `Arr1` a `Arr2` pole prvků stejného typu (sbalená či nesbalená) a mají-li stejné uspořádání, mohou s nimi být v SystemVerilogu prováděny následující operace (zčásti o tom již byla řeč v odst. 2.3.11.5):

- přiřazování jednoho pole druhému poli (kopírování), např. `Arr1 = Arr2`,
- přiřazování (kopírování) řezu z pole, např. `Arr1[i:j] = Arr2[i:j]`,
- přiřazování (kopírování) řezu z pole s proměnným počátečním bodem, např. `Arr1[a+:j] = Arr2[b+:j]`,
- přiřazování prvku pole, např. `Arr1[i] = Arr2[i]` (platné i ve Verilogu),
- operace porovnání polí nebo řezů, např. `Arr1 == Arr2` nebo `Arr1[i:j] != Arr2[i:j]` (to jsou platné relační výrazy, viz odst. 2.4.5).

Se sbalenými (ne však s nesbalenými) poli je navíc možno v SystemVerilogu provádět v procedurálních blocích (viz poznámku v odst. 2.3.11.3) aritmetické, relační, bitové a logické operace ve výrazech stejně jako s vektory také s objekty, jako jsou bitové a částečné výběry ze sbalených polí, sjednocení takových polí nebo výběrů – například:

```
logic [7:0][3:0] Op1,Op2,Result;  // jen SV
  Result = (Op1 + Op2) >> 2;
```

Pro další příklady operací s poli uvažujme složené pole `Ar20` dané následující deklarací:

```
bit [3:0] [7:0] Ar20 [1:20];  // jen SV
```

Toto nesbalené pole je složeno z 20 prvků, z nichž každý obsahuje 32 bitů ve dvou rozměrech (4x8 bitů). Jsou pak přípustné například následující operace:

```
Ar20[20] = Ar20[19] + 1;       // inkrementace prvku (4x8 bitu)
Ar20[10][1:0] = Ar20[9][3:2]; // zkopirovani rezu - dvou 8bitovych skupin
```

2.3.11.7 Předávání polí branami, funkcím a úlohám

Verilog dovoluje předávání vektorů (v SystemVerilogu považovaných za jednorozměrná sbalená pole) branami modulů. Přímé předávání nesbalených polí není ve Verilogu možné – pro předání takového pole je nutno vytvořit samostatnou bránu pro každý prvek pole nebo vytvořit bránu jako vektor, do něhož se prvky pole „sbalí“ smyčkou, jde-li o výstup; při vstupu se naopak takový vektor musí smyčkou „rozbalit“ do prvků pole. Příklad takového předávání ve Verilogu i v SystemVerilogu lze najít v článku [30].

SystemVerilog dovoluje předávání jakýchkoliv polí branami modulů a rozhraní. V tom případě musí být brána deklarována také jako pole. Pole, která jsou bránou předávána, musí splňovat pravidla a omezení, která platí pro kopírování polí (přiřazování jednoho pole jinému poli).

Totéž platí i pro předávání polí jako argumentů funkcím a úlohám.

2.3.11.8 SystemVerilog: Systémové funkce pro pole

Efektivní práci s poli umožňují **systémové dotazovací funkce pro pole** (*array querying system functions*). Jejich využitím lze psát příkazy bez uvedení číselných hodnot rozsahů polí ve tvaru, který se automaticky přizpůsobí konkrétním číselným hodnotám rozsahů uvedeným v deklaracích polí. V případech, kdy se pracuje s několika poli, jejichž rozsahy jsou vzájemně závislé (či dokonce stejné), tyto funkce umožňují zapsat deklarace polí tak, že při případné následné změně rozsahů stačí zapsat změnu jen jednou, a změny v deklaracích ostatních polí se pak provedou automaticky. Jsou to tyto funkce:

`$dimensions`(*ident*)
`$unpacked_dimensions`(*ident*)
`$left`(*ident*,*rozměr*)
`$right`(*ident*,*rozměr*)
`$low`(*ident*,*rozměr*)
`$high`(*ident*,*rozměr*)
`$increment`(*ident*,*rozměr*)
`$size`(*ident*,*rozměr*)

Uvedené funkce vracejí hodnotu typu **`integer`**. Funkce *`$dimensions`* a *`$unpacked_dimensions`* mají jeden argument *ident* – identifikátor pole, kterého se týkají, ostatní mají dva argumenty: prvním je opět identifikátor pole, a druhý, *rozměr* (nepovinný), určuje pořadové číslo rozměru (dimenze), jehož se funkce týká. Není-li tento druhý argument uveden, předpokládá se, že má hodnotu `1`. Pořadové číslo rozměru může být dáno číslem nebo výrazem. Pořadová čísla jednotlivých rozměrů odpovídají jejich indexování v odkazech a jsou v souladu s náčrtkem na ***Obr. 2.1***, tedy například pro složené pole deklarované zápisem:

```
logic [7:0][3:0] ArrOfArr [1:256][0:3];
```

má rozměr `[1:256]` pořadové číslo 1, rozměr `[0:3]` pořadové číslo 2, rozměr `[7:0]` pořadové číslo 3 a rozměr `[3:0]` pořadové číslo 4.

Hodnoty vracené těmito funkcemi jsou následující (nebudeme uvažovat případy, kdy argumentem je identifikátor fronty nebo dynamického pole – tyto i další informace najde zájemce v manuálu [6]):

Funkce *`$dimensions`* vrací hodnotu představující celkový počet rozměrů pole (sbalených i nesbalených); je-li v argumentu identifikátor řetězce nebo jiného objektu s charakterem nikoli pole a s typem ekvivalentním typu jednoduchého bitového vektoru (odst. 2.3.11.1), vrací hodnotu `1`, a pro jakýkoliv jiný typ objektu vrací hodnotu `0`.

Funkce *`$unpacked_dimensions`* vrací hodnotu představující celkový počet nesbalených rozměrů pole pro identifikátor objektu typu pole, a

provedením předchozích příkazů v bloku se v dalších příkazech akceptuje (viz např. text TXT5.1.2a). Postup zpracování bloku tedy odpovídá tomu, co je obvyklé u běžných programovacích jazyků. V paměťovém bloku **always** (**always_ff**, **always_latch**) s neblokujícími přiřazeními bez specifikovaného zpoždění však proměnná provedením příkazu novou hodnotu ještě nezískává – ve všech příkazech (i v podmínkách obsahujících tuto proměnnou) v bloku se vychází z hodnoty, kterou proměnná měla při vstupu do bloku, a proměnná získává novou hodnotu až po ukončení bloku (případně simulačního cyklu) z posledního provedeného přiřazení; případná přiřazení nových hodnot této proměnné v bloku se před ukončením bloku na její hodnotě neprojeví – to uvidíme například v textu TXT5.4.3a.

Účelem těchto zdánlivě složitých pravidel je zajistit, aby se při simulaci choval model stejně, jako se chová jeho technická realizace po implementaci do cílového obvodu, kde jeho jednotlivé části, jako hradla, klopné obvody a podobně, reagují na signály samostatně, zatímco v počítačovém modelu nemůže simulace probíhat jinak než postupně, v jednom „vláknu“. Podrobněji o tom budeme mluvit v kapitole 4.

Rozdíly mezi blokujícím a neblokujícím přiřazením jsou dosti jemné. V běžných situacích však v nich neuděláme chybu, když se budeme řídit následujícími jednoduchými pravidly, která jsou pro Verilog uvedena například v knize [7], a podobně v [11] pro SystemVerilog:

- Používejte **blokující přiřazení** (=) v blocích **always** a **always_comb**, které mají vytvořit **kombinační** logický obvod.
- Používejte **neblokující přiřazení** (<=) v blocích **always**, **always_latch** a **always_ff**, které mají vytvořit **paměťový** logický obvod (registr, čítač, latch a podobně).
- V každém bloku skupiny **always** používejte jen jeden z těchto typů přiřazení.
- Nepřiřazujte téže proměnné hodnotu ve dvou nebo ve více blocích **always**.

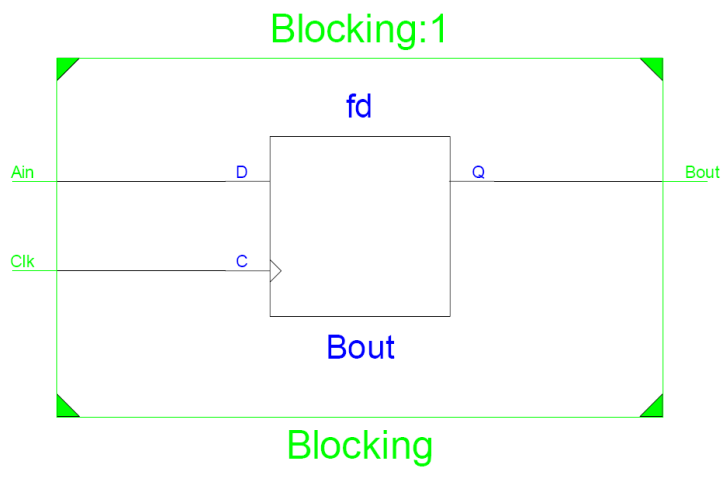
Výjimku ze třetího pravidla mohou představovat příkazy pro manipulaci s lokálními proměnnými bloku vytvářejícího paměťový obvod (jde o proměnné, z nichž syntézou nevzniká registr, například řídící proměnné ve smyčkách). V příkazech vytvářejících paměťové prvky se pak používá neblokující přiřazení, u příkazů pro lokální proměnné přiřazení blokující. Jinak je však lepší se tomu vyhnout, je-li to možné.

Dodržení posledního z uvedených pravidel ve Verilogu může být nesnadné zejména ve velkých modelech. V SystemVerilogu jeho porušení při použití nově zavedených blokových příkazů **always_comb**, **always_ff**, **always_latch**, nebo je-li vícenásobné přiřazení provedeno v kontinuálním příkazu **assign**, znamená syntaktickou chybu, což prakticky představuje automatickou kontrolu dodržení tohoto pravidla. Přiřazení nové hodnoty téže proměnné ve dvou či více blocích **always** ve Verilogu i v SystemVerilogu je přípustné a simulátory je akceptují, není však syntetizovatelné – viz např. [13]. Uvádí se, že Verilog tak dovoluje modelovat abstraktní signálové spoje, kde více zdrojů budí společnou linku; výsledný signál je pak stanoven pomocí rozhodovací funkce (viz odst. 2.3.1) s ohledem na úroveň intenzity (*drive strength*) budičů. V našich uvažovaných aplikacích se s tím však sotva setkáme.

Jako příklad nevhodného a vhodného použití přiřazovacích operátorů uvedeme popis dvoubitového posuvného registru (dosud jsme o popisu paměťových prvků neuvažovali, ale popis je vcelku pochopitelný; podobné konstrukce se často vyskytují například při optimalizaci časových parametrů pipeliningem). V prvním textu použijeme blokující přiřazení (oba texty jsou psány ve Verilogu; v SystemVerilogu bude typ **reg** nahrazen typem **logic**, a místo **always** bude **always_ff**, jak bylo uvedeno v poznámce na konci odst. 2.1, výsledek však bude v obou jazycích stejný):

```
module Blocking(input Clk,Ain,          // V                    // TXT2.12
                output reg Bout);
    reg Tmp;
  always @(posedge Clk) begin
    Tmp = Ain;
    Bout = Tmp;
  end
endmodule
```

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

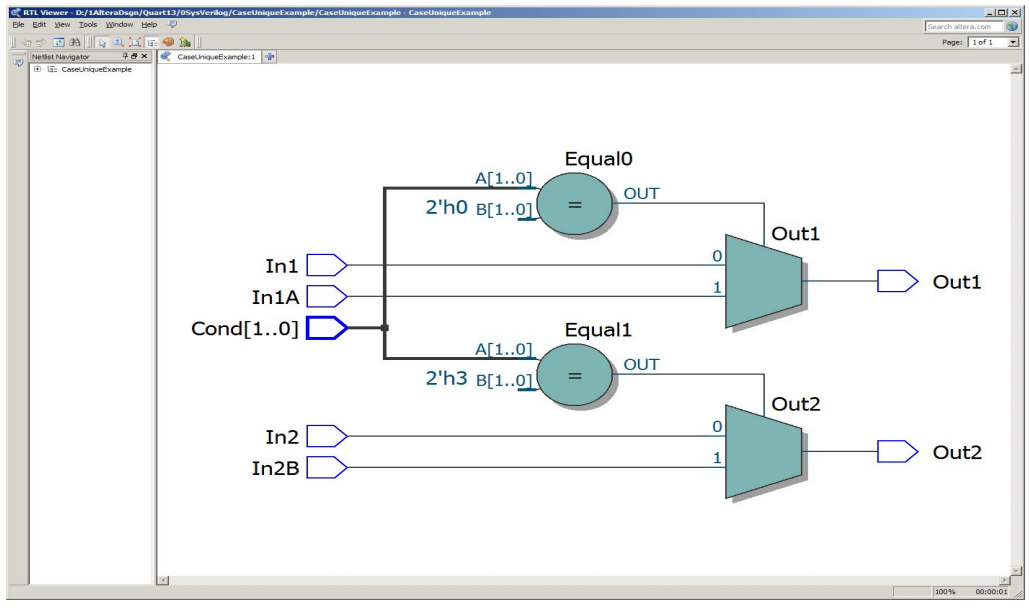


Obr. 2.2. *Výsledek zobrazení RTL Schematic u textu* TXT2.12

V druhém textu, jinak shodném, použijeme neblokující přiřazení:

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

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

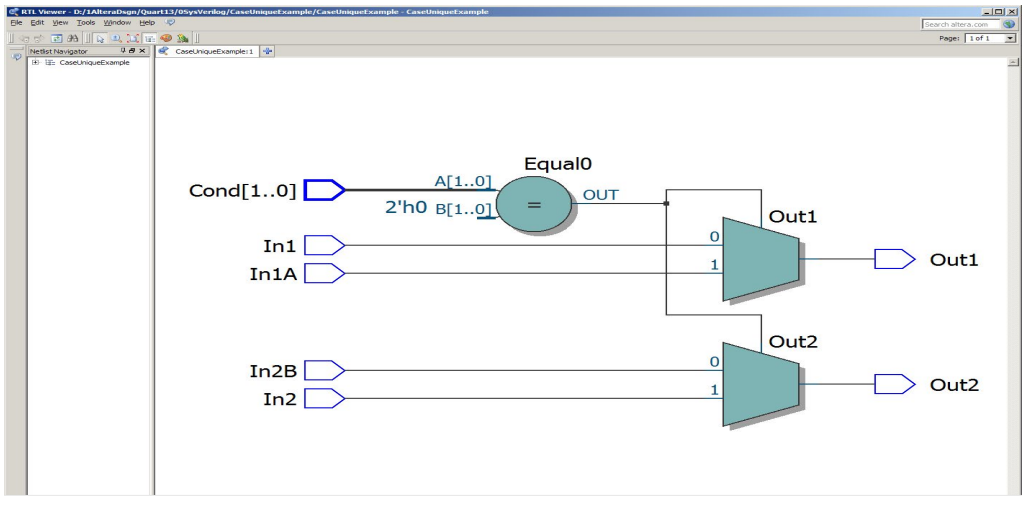


Obr. 2.4. *Výsledek zobrazení RTL Viewer u textu* `TXT2.15`

Pokud bychom však zde použili modifikátor **`unique`**, dostali bychom zcela jiný výsledek:

```
module CaseExample2(input In1,In2,In1A,In2B,      // V*,SV      // TXT2.16
                    input [1:0] Cond,
                    output logic Out1,Out2);
  parameter CondA = 2'b00, CondB = 2'b11;
  always_comb begin
    Out1 = In1;
    Out2 = In2;
    unique case (Cond)
      CondA: Out1 = In1A;
      CondB: Out2 = In2B;
    endcase
  end
endmodule
```

Zpracování textu `TXT2.16` procesem RTL Viewer v systému Quartus dává následující výsledek:



Obr. 2.5. *Výsledek zobrazení procesem RTL Viewer u textu* `TXT2.16`

Stejné schéma vznikne ve Verilogu s použitím pragmatu *`full_case`*. Jak vyplynulo z korespondence autora s aplikačními inženýry firmy Altera, je zřejmě nesprávný výsledek v případě textu s modifikátorem **`unique`** způsoben nesprávnou funkcí systému Quartus. Nicméně se např. v článku [31] tvrdí, že při použití výchozího přiřazení může modifikátor **`unique`** vést k chybným výsledkům syntézy, což potvrzuje i třetí z následujících doporučení.

Vodítkem pro rozhodnutí, kde modifikátory použít, mohou být doporučení uvedená v článku [23], která uvedeme dále (odst. 2.6.5.4). Tato doporučení jsou platná především pro syntézu, neuděláme však chybu, budeme-li je dodržovat i v textech určených pro verifikaci, tedy v benčích, kde mohou být užitečné funkce modifikátorů spojené s tvrzením (*assert*).

Na závěr ještě uvedeme příklad použití příkazů **`if-else`** a **`case`** k popisu multiplexoru, u něhož je výběrový signál `Sel` kódován kódem 1 ze 3.

V prvním příkladu je multiplexor popsán příkazem **`if-else`**:

```
module Mux3To1wLatch(input logic [2:0] Sel,         // V*,SV    // TXT2.17
                     input logic [1:0] In0,In1,In2,
                     output logic [1:0] MxOut);
  always_comb
    if      (Sel == 3'b001) MxOut = In0;
    else if (Sel == 3'b010) MxOut = In1;
    else if (Sel == 3'b100) MxOut = In2;
endmodule
```

Při pokusu o jeho syntézu v systému Quartus je inferován latch. Je-li v systému nastavena syntaxe SystemVerilogu, hlásí se chyba (příkaz

3 Pokročilé prvky jazyka SystemVerilog

Jazyk SystemVerilog je koncipován tak, aby se jeho konstrukty pokud možno blížily konstrukům jazyka C. Motivace k tomu je mimo jiné dána skutečností, že se předpokládá vývoj programů pro automatický převod mezi těmito jazyky. V této kapitole si povšimneme některých konstruktů jazyka SystemVerilog, které mají význam pro syntézu a blíží se podobným konstruktům jazyka C. Tyto konstrukty mají umožnit popis na vyšší úrovni než je úroveň většiny konstruktů, s nimiž jsme se setkali v předcházejících kapitolách. To dovoluje stručně a přehledně popsat zejména rozsáhlé systémy s rozvinutou hierarchickou strukturou, jejíž dílčí bloky mají velký počet vstupních a výstupních signálů. Nicméně se tyto konstrukty často mohou výhodně uplatnit i při popisu poměrně malých systémů.

3.1 Struktury

Struktury jsou konstrukty podobné polím. Dovolují sdružovat datové objekty – **členy** nebo také **položky** struktury (v angličtině označované slovem *field* [6], někdy také *member* [11]) – do jednoho celku, s nímž pak lze pracovat jako s jedním objektem. Na rozdíl od pole, které je složeno z prvků stejného datového typu a stejné bitové šířky, mohou mít členy struktury různé datové typy a různou šířku. Členy struktury mohou být proměnné a/nebo konstanty či parametry jakéhokoliv typu (a také to mohou být sbalená i nesbalená pole nebo i jiné struktury – tedy substruktury). Podle [11] nemohou být členy struktury objekty typu propojení (*net*) – pro ně je určen konstrukt zvaný rozhraní (*interface*), viz odst. 3.3; manuál [6] však takové členy výslovně nevylučuje, naopak se tam uvádí, že nesbalené struktury mohou obsahovat členy jakýchkoliv typů. Podle [11] je struktura bez explicitní deklarace typu považována za proměnnou. Může však jako celek být deklarována i jako propojení, jak bude vysvětleno dále.

3.1.1 Definice a vložení struktury

Struktura se **definuje** klíčovým slovem `struct`. Syntaxe definice struktury je podobná jako v jazyku C, s tím rozdílem, že jazyk C dovoluje deklarovat „tag“ za klíčovým slovem `struct`, před výčtem členů struktury, což v SystemVerilogu není dovoleno.

Příklad jednoduché definice struktury (předpokládá se, že uživatelský typ `OpCode_t` byl dříve definován):

```
struct {shortint     a,b;     // 16bitove promenne
        OpCode_t     OpCode;  // promenna s uzivatelsky definovanym typem
        logic [15:0] Addr;    // 16bitova promenna - adresa
       } InstWord_st;
```

Výčet členů struktury je zapsán ve složených závorkách. Připomeňme, že ve shodě s poznámkou z odst. 2.1.2 budeme identifikátory struktur zakončovat koncovkou `_st`.

Se strukturou je možno pracovat jako s celkem, nebo se také lze odvolávat hierarchicky na jednotlivé její členy (se syntaxí podobnou syntaxi v jazyku C):

jméno_struktury`.`*jméno_proměnné*

Například adrese ze struktury `InstWord_st` lze přiřadit hodnotu příkazem:

```
InstWord_st.Addr = 16'hF01B;
```

Struktura s definicí uvedenou výše se označuje za strukturu s **anonymním typem** (v anglické literatuře se jí také říká anonymní struktura – *anonymous structure*). Je však možno také nejprve definovat typ struktury klíčovým slovem `typedef`, a následně deklarovat strukturu (nebo i více struktur) tohoto typu – pak jde o strukturu s **pojmenovaným typem** (*named*, nebo také *typed*, [11]) – podle následujícího příkladu:

```
typedef struct {                  // definice typu InstWord_t
  shortint     a,b;
  OpCode_t     OpCode;
  logic [15:0] Addr;
  } InstWord_t;

InstWord_t InstW1_st,InstW2_st,InstW3_st;
                          // deklarace 3 struktur typu InstWord_t
```

Identifikátory `InstW1_st`, `InstW2_st`, `InstW3_st` zde označují proměnné – struktury s uživatelsky definovaným typem `InstWord_t`. Těm lze nyní přiřadit hodnoty a dále s nimi pracovat. O takové deklaraci struktury (s pojmenovaným typem) se mluví také jako o **vložení struktury** (*structure instantiation*).

Podle [11] může být struktura také explicitně definována jako propojení (*net*) užitím příslušného dodatečného klíčového slova (`wire`, `tri` a podobně); v tomto případě musí být typy všech jejích členů čtyřhodnotové. Příklad takové definice:

```
wire struct {integer      a,b;
             OpCode_t     OpCode; // ctyrhodnotovy typ drive definovany
             logic [15:0] Addr;
            } InstWordWire_st;
```

Členy struktury definované jako propojení musí však podle [11] být proměnné nebo konstanty, nemohou to být propojení (jak jsme se již zmínili, manuál [6] to výslovně nevyžaduje). Struktura explicitně nedefinovaná jako propojení se automaticky považuje za proměnnou.

Struktura může být definována v konstrukčním prvku – v modulu nebo v rozhraní, kde se používá, pak jde o lokální definici. Lze však také definovat typ struktury ve sloze a tuto definici importovat do konstrukčních prvků v projektu nebo do kompilační jednotky (viz odst. 2.2.2) – pak tento typ může být používán k vložení struktur ve více prvcích, a typ může být použit pro deklaraci bran modulů či rozhraní nebo argumentů funkcí a úloh, jak uvidíme v odst. 3.1.4. Typ struktury lze také definovat přímo v kompilační jednotce, což však není doporučovaná praxe.

Zajímavý a zřejmě prakticky užitečný příklad struktury, jejímiž členy jsou substruktury, je uveden v literatuře [60]. Uvažuje se tam o číslech

```
                                          // typu Instr_t
                                          // s nazvem Instr_st
                    output logic [7:0] DaOut);
  logic WaitCyc;
  logic [7:0] DataVar;
  always @(posedge RamIf.Clk, posedge RamIf.Rst)
    if (RamIf.Rst) begin
      DaOut <= ,0; WaitCyc <= 1'b1; DataVar <= ,Z;
    end
    else begin
          WaitCyc <= !WaitCyc;
          if (!WaitCyc)
            unique case (Instr_st.OpCode)
              2'b01: begin RamIf.Addr <= Instr_st.DaAddr;
                           RamIf.WrEn <= 1'b1; DataVar <= Instr_st.DaIn;
                     end
              2'b10: begin RamIf.Addr <= Instr_st.DaAddr;
                           RamIf.WrEn <= 1'b0; DataVar <= ,Z;
                     end
              default: begin RamIf.WrEn <= 1'b0; DataVar <= ,Z; end
            endcase
          else begin RamIf.WrEn <= 1'b0; DaOut <= RamIf.Data;
                     DataVar <= ,Z;
          end
    end
  assign RamIf.Data = DataVar;
endmodule

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

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

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

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

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

Do vrcholového modulu `Top` je vloženo rozhraní `Intf1_if` jako vnitřní signál s identifikátorem `TopIf`. Tento identifikátor může být libovolný, a byl zvolen tak, aby odpovídal funkci signálu. Není však žádný zvláštní důvod, proč by zde nemohl být použit jiný identifikátor, například `RamIf`. To by však mohlo být poněkud matoucí, protože jde o signál poněkud jiného druhu, než jsou signály bran obou modulů, které mají tento název, nikoli však syntakticky nebo sémanticky nesprávné. Kdybychom tento identifikátor pro vnitřní signál modulu `Top` použili, bylo by možno popis tohoto modulu zapsat stručněji důslednějším použitím zápisu formou *dot-star* (zkušení uživatelé SystemVerilogu by pravděpodobně tak postupovali):

```
module Top (input Clk,Rst,
            input Instr_t Instr_st, // struktura jako brana typu Instr_t
                                    // s nazvem Instr_st
```

Syntetizéry většinou také dokáží zdrojové texty zapsané v syntaxi Verilogu-2001 lépe optimalizovat. Nebudeme se zde proto zabývat přístupem odpovídajícím Verilogu-95, zájemce odkazujeme na bohatou literaturu.

Poněkud složitější situace však i při zápisu v syntaxi Verilogu-2001 nastává, je-li část operandů typu **unsigned**. Uvedeme zde dva příklady.

Při sčítání dvou vektorů musí mít výsledný vektor aspoň o jedničku větší šířku než širší ze sčítanců, má-li být zachycen přenos – například stačí přiřadit součet dostatečně širokému signálu. Jsou-li oba operandy typu **signed**, doplní se pak operandy před provedením operace sčítání automaticky znaménkovým bitem zleva na šířku výsledku. Je-li však kterýkoliv operand (i při součtu více operandů) typu **unsigned**, provádí se sčítání podle pravidel aritmetiky čísel bez znaménka, s doplněním zleva nulou. Má-li tedy být sčítání provedeno podle pravidel aritmetiky čísel se znaménkem, musí být všechny operandy typu **signed**. Operandy typu **unsigned** je nutno nejprve na typ **signed** převést. Tyto operandy se musí před převodem doplnit zleva aspoň jednou nulou, a teprve pak se provede převod – tím se zajistí, že po převodu na typ **signed** budou mít znaménkový bit nulový (jsou to kladná čísla).

Sčítačka dvou čtyřbitových vektorů typu **signed** se vstupem přenosového bitu tedy bude popsána následujícím textem (význam funkce *$signed* viz odst. 2.3.5):

```
module AddSignedWithCarry(input logic signed [3:0] Ain,Bin, // TXT5.1.8a
                          input logic CyIn,         // unsigned  // V,SV
                          output logic signed [4:0] Sum);
  assign Sum = Ain + Bin + $signed({1'b0,CyIn});
endmodule
```

V SystemVerilogu může být přiřazovací příkaz zapsán také poněkud stručněji s použitím převodu (změny) šířky a atributu **signed**/**unsigned** (odst. 2.3.8):

```
assign Sum = Ain + Bin + signed'(2'(CyIn));
```

Připomeňme, že skupina znaků †EF je operátor převodu typu, kterým se zde jednobitový signál `ófå převádí na dvoubitový – doplňuje se zleva nulou. Vytvoření benče pro sčítačku (např. podle vzoru v TXT5.1.3b) přenecháme čtenáři.

Pro násobičku připomeňme, že podle pravidel aritmetiky Verilogu je bitová šířka součinu rovna šířce širšího z operandů (odst. 2.4.9); aby nedošlo ke ztrátě vyšších bitů výsledku, je zde potřebné přiřadit výsledek násobení dostatečně širokému signálu – tedy signálu s počtem bitů nejméně rovnému součtu počtu bitů obou operandů – a na tuto šířku jsou pak před provedením operace součinu automaticky rozšířeny oba operandy. Popis násobičky čtyřbitových operandů s odlišnými typy a s výsledkem typu **signed** tedy může vypadat takto:

```
module MultSignedUnsigned(input logic signed [3:0] Ain,     // TXT5.1.8b
                          input logic [3:0] Bin,   // unsigned  // V,SV
                          output logic signed [7:0] Prod);
  assign Prod = Ain * $signed({1'b0,Bin});
endmodule
```

Odpovídající benč pro násobičku je:

```
module MultSignedUnsigned_TF;          // V*,SV             // TXT5.1.8c
  logic signed [3:0] Ain;    // vstup uut, ve Verilogu typ reg signed
  logic [3:0] Bin;           // vstup uut, ve Verilogu typ reg
  logic signed [7:0] Prod;   // vystup uut, ve Verilogu typ wire signed
// Instantiate the Unit Under Test (UUT)
  MultSignedUnsigned uut (.*);  // syntaxe ve Verilogu - viz TXT5.1.2c
  initial begin: TB
// Add stimuli here
    for (int i=0; i<=15; i++) begin
      for (int j=0; j<=15; j++) begin
        Ain = i; Bin = j;
        #10;
      end
    end
    #10; $stop;
  end
endmodule
```

5.2 Subsystémy se zvláštními typy bran

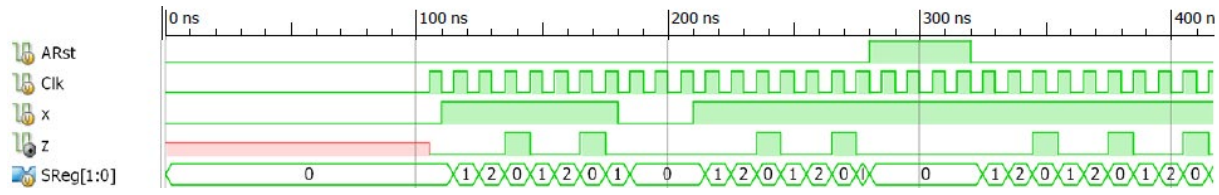
V této kapitole probereme modelování subsystémů s třístavovými výstupy, s otevřeným kolektorem a s obousměrnými branami. Podobně, jako jsme to poznali u modelování paměťových prvků (odst. 2.6.9), i pro syntézu subsystémů s takovými branami je potřebné u behaviorálních modelů používat standardní způsob popisu – šablonu, kterou systém rozpozná a vytvoří strukturu s odpovídajícím uspořádáním výstupů nebo obousměrných bran.

5.2.1 Subsystémy s třístavovými výstupy a s otevřeným kolektorem

Třístavové výstupy se inferují nejlépe z behaviorálního popisu. Můžeme zde použít kontinuální příkaz, kde signál třístavové brány s názvem `OutPort` je ve Verilogu propojení typu **wire** nebo **tri** (ve Verilogu se kontinuálním příkazem může přiřazovat hodnota jen propojením), v SystemVerilogu to může být i proměnná (obvykle datového typu **logic**):

```
assign OutPort = OutEnable ? OutSig : ,bz;
```

Třístavový výstup lze také modelovat blokem **always**. V takovém bloku však signál `OutPort` musí být proměnná (obvykle ve Verilogu typu **reg**, v SystemVerilogu **logic**), protože signálu typu propojení nelze přiřazovat hodnotu procedurálním příkazem (viz poznámku na konci odst. 2.1):



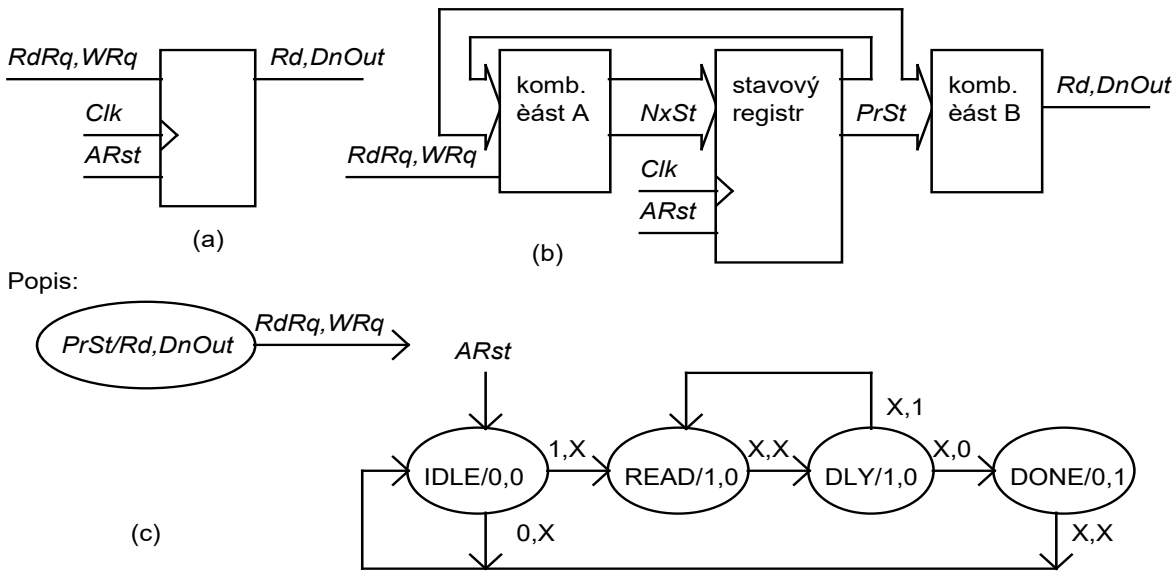
Obr. 5.13. *Výsledek funkční simulace modelu* `Det3OnesMea`

Popis tohoto detektoru v SystemVerilogu bude:

```
module Det3OnesMeaSV (input logic Clk,x,ARst,     // jen SV  // TXT5.5.2b
                      output logic z);
  logic Znxt;
  enum logic [1:0] {S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11} SReg;
  initial SReg = S0;                // SReg - alias PrSt - stavovy registr
  always_ff @(posedge Clk, posedge ARst)
    if (ARst) SReg <= S0;
    else
      unique case (SReg)                    // stavove prechody
        S0: if (x) SReg <= S1;
            else SReg <= S0;
        S1: if (x) SReg <= S2;
            else SReg <= S0;
        S2: SReg <= S0;
      endcase
  always_comb
    unique case (SReg)                      // vystupni prirazeni
      S0: Znxt = 0;
      S1: Znxt = 0;
      S2: if (x) Znxt = 1;
          else Znxt = 0;
      default: Znxt = 0;
    endcase
  always_ff @(posedge Clk, posedge ARst) // vystupni registr
    if (ARst) z <= 0;
    else z <= Znxt;
endmodule
```

5.5.3 Příklad 3: Řídící obvod čtení z paměťového média

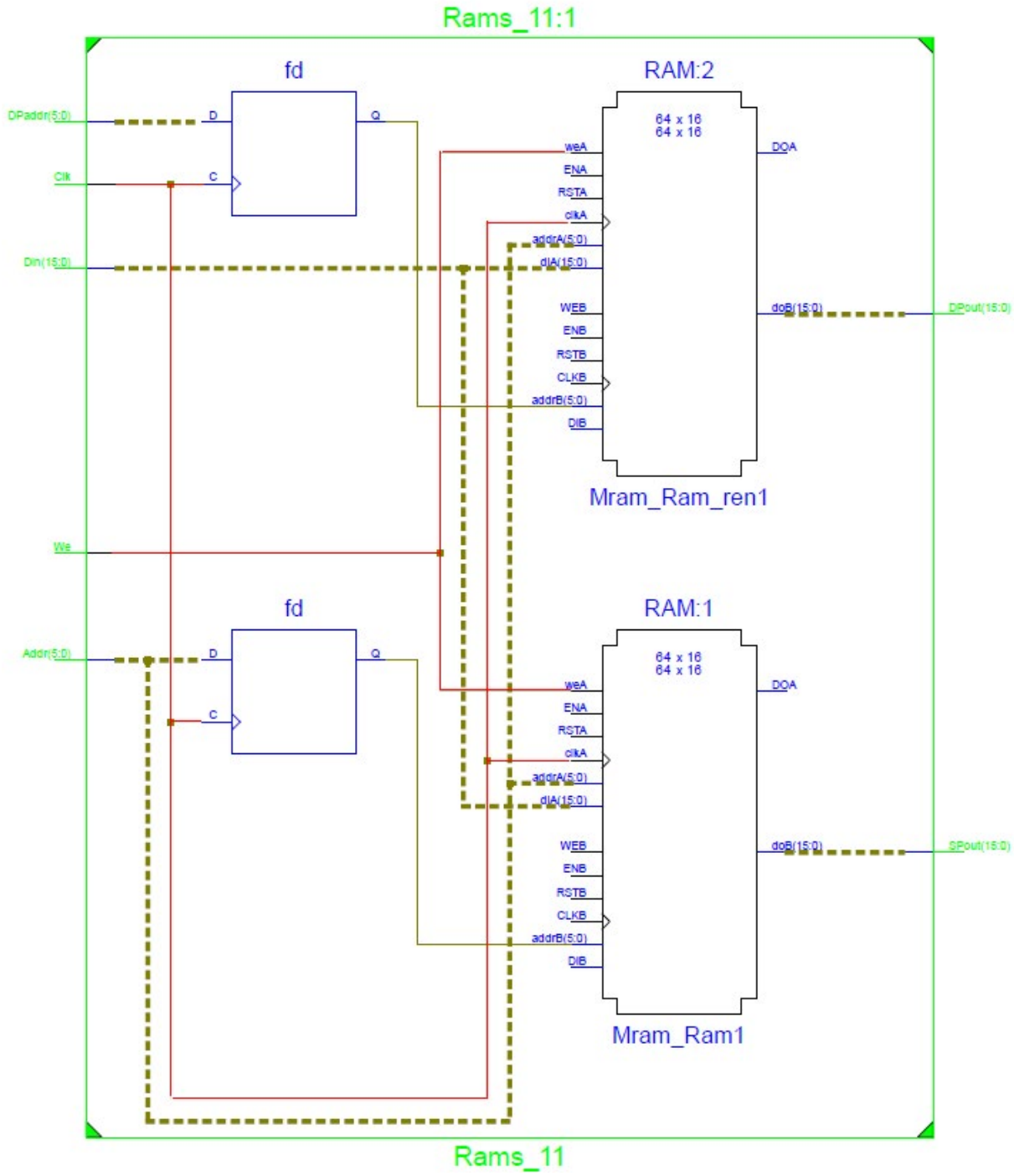
Jako příklad způsobu popisu stavového automatu, u něhož je v SystemVerilogu použit zvláštní blok **`always_ff`** pro popis stavového registru a další blok **`always_comb`** pro popis kombinačních částí, uvedeme jednoduchý automat určený pro řízení čtení z paměťového média. Blokový symbol, schéma a stavový diagram jsou na ***Obr. 5.14***. V klidu je obvod ve stavu `IDLE`. Po aktivaci vstupního signálu `RdRq` (Read Request) přejde do stavu `READ` a v následujícím taktu do stavu `DLY` (Delay), přičemž v obou těchto stavech aktivuje výstupní signál `Rd` (Read). Je-li ve stavu `DLY` aktivní vstupní signál `WRq` (Wait Request), vrací se obvod do stavu `READ`. Při neaktivním signálu `WRq` přechází do stavu `DONE`, přičemž aktivuje po dobu jednoho taktu výstupní signál `DnOut`, načež se vrací do klidového stavu `IDLE`.



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

Automat sestavíme v Moorově verzi. Popis ve Verilogu bude:

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



Obr. 5.22. *Schéma RTL syntetizovaného modelu dvoubránové paměti RAM se synchronním čtením* R~ãë|NN

Schéma RTL na ***Obr. 5.22*** obsahuje dva paměťové prvky. Při implementaci však systém ISE sloučí oba tyto prvky do jednoho bloku, což můžeme vidět z technologického schématu (není zde nakresleno, zájemce si je zobrazí v ISE). V technologickém schématu nejsou registry v adresových vstupech, synchronní čtení je zajištěno paměťovými bloky.

Podle [48] může být tato paměťová konfigurace implementována i do distribuované paměti.

5.6.7 Jednobránová bloková paměť ROM se synchronním čtením

Mnoho typů obvodů FPGA dovoluje specifikovat počáteční hodnotu paměťových prvků (hodnotu, kterou mají po připojení napájecího napětí) – platí to například o všech paměťových prvcích v obvodech FPGA řad Spartan i Virtex. Jestliže se do paměti dále nezapisuje, funguje takové paměť jako paměť typu ROM. Způsoby stanovení počátečního obsahu paměti mohou být odlišné v různých návrhových systémech. V systému ISE lze použít příkaz **case**, jak vidíme v textu TXT5.6.7, z něhož systém inferuje paměť ROM, viz [48]. Schéma RTL je zachyceno na ***Obr. 5.23***. Blok paměti je zde následován výstupním registrem, jehož funkci v technologickém schématu opět přebírá strukturní prvek paměťového bloku. V textu TXT5.6.7 je příkaz **case** zapsán jen částečně, pro tři hodnoty adresy.

```
module R~ãë|ON~  (input Clk,En,                  // V          // TXT5.6.7
                  input[5:0] Addr,
                  output reg [19:0] Data);
  always @(posedge Clk)
    if (En)
      case(Addr)
        6'b000000: Data <= 20'h0200A;
        6'b000001: Data <= 20'h00300;
              ...
        6'b111111: Data <= 20'h0400D;
      endcase
endmodule
```