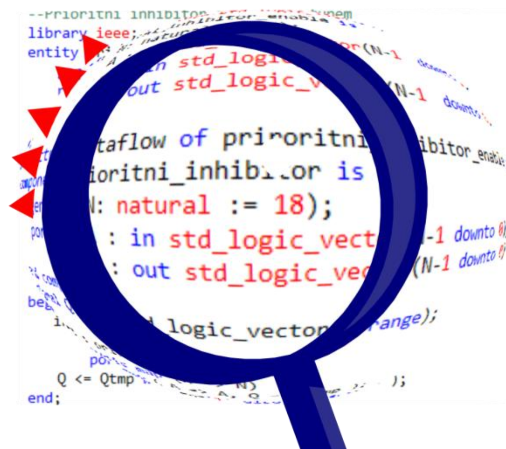


Návrh obvodů ve VHDL modelovacími styly "dataflow" a "structural"

Richard Šusta

Katedra řídicí techniky
ČVUT-FEL v Praze

Verze 1.1 - září 2019

Obsah

1 Úvod 4

2 Základy syntaxe VHDL a příkaz souběžného přiřazení <= 6
2.1 Základy syntaxe a identifikátory 6
2.2 Datový typ std_logic 7
2.3 Signal - vodič a souběžné přiřazení <= 7
2.4 Operace s typy std_logic a boolean 8
2.5 @Příklad I. - VHDL kód majority 10
2.6 VHDL šablona 11
2.7 @Příklad II. - VHDL kód 4-vstupového XOR 14
2.8 @Příklad III. - Majorita2 s indikací dvou v '1' 15
2.9 *** Cvičná úloha 1: - Majorita123 16
2.10 Datový typ std_logic_vector 17

3 @Příklad IV. - Dekodér příkazem with…select 20
3.1.b @Příklad V. - Logická ALU 23
3.2 @Příklad VI. - Dekodér pro velký ukazatel 24
3.3 Tvorba vlastních typů a atributy VHDL 24
3.4 @Příklad VII. - Dekodér pro velký ukazatel s polem 26
3.5 *** Cvičná úloha 2: Dekodér pro 7segmentový displej 27

4 @Příklad VIII. - VHDL příkaz when…else 28
4.1.a @Příklad IX. - Prioritní dekodér s 15-vstupy 30
4.2 Podmíněné přiřazení 30
4.3 *** Cvičná úloha 3: 8-vstupový prioritní inhibitor 31

5 @Příklad X. - VHDL příkazy generic a for-generate 32
5.1 @Příklad XI. - Deklarace generic 33
5.2 @Příklad XII. - Kontrola parametrů generic 34
5.2.a Co ještě chybí? 36
5.3 *** Cvičná úloha 4: Univerzální prioritní inhibitor 37
5.4 Inicializace vektorů proměnných délek asociací others=> 37
5.5 Převod čísel integer na unsigned a signed 38
5.6 *** Cvičná úloha 5: Rozšiřte prioritu přerušení pomocí generic 38

6 VHDL stylem "Structural" 39
6.1 @Příklad XIII. - Použití prioritního inhibitoru ve VHDL 39
6.1.a Stejné názvy generic parametrů 43
6.1.b @Příklad XIV. - Prioritní inhibitor s funkcí test a blank 43
6.2 *** Cvičná úloha 6: 7segmentový dekodér s potlačením úvodních 0 45
6.3 @Příklad XV. - Propojení několika obvodů 46

6.4 *** Cvičná úloha 7: Přidání 7segmentového displeje 47
6.5 @Příklad XVI. - Vícenásobné instance ve vektoru majority 47
6.6 @Příklad XVII. - Vektor majority s for generate 48
6.6.a Špatné užití pomocné definice 49
6.6.b Fatální chyba: Opakované přiřazení signálu ve for generate 51
6.7 @Příklad XVIII. - Příkaz for generate pro oddělené vstupy 52
6.8 *** Cvičná úloha 8: Zobrazení 16-bitového čísla na 7segmentovém displeji 53
6.9 *** Cvičná úloha 9: Prioritní inhibitor pro tři trojice 53

7 @Příklad XIX. - Vytvoření vlastní knihovny ("package") 54
7.2 @Příklad XX. - Distribuce - vše v jednom 57

8 Příloha A: Řešení cvičných úloh 58
8.1 Cvičná úloha 1: Majorita123 58
8.2 Cvičná úloha 2: Dekodér pro 7segmentový displej 59
8.3 Cvičná úloha 3: Prioritní inhibitor 61
8.4 Cvičná úloha 4: Univerzální prioritní inhibitor 62
8.5 Cvičná úloha 5: Univerzální prioritní kodér 64
8.6 Cvičná úloha 6: 7segmentový dekodér s potlačením úvodních 0 65
8.7 Cvičná úloha 7: Přidání 7segmentového displeje 66
8.8 Cvičná úloha 8: Zobrazení 16-bitového čísla na 7segmentovém displeji 67
8.9 Cvičná úloha 9: Prioritní inhibitor pro tři šestice 69

9 Příloha B: Definice operací s std_logic - Resolution tables 71

10 Příloha C: Vytvoření a překlad VHDL souboru majorita.vhd 73
10.1 Uložení VHDL souboru 74
10.2 Překlad souboru 75
10.3 Chyby a varování při překladu 76
10.4 Zobrazení RTL Map a Technology Map 78
10.5 Vytvoření schematické značky pro symbolický editor 79
10.6 Simulace 80
10.7 Vložení indexovaných signálů 86

11 Závěr 87
11.1 Historie verzí dokumentu 87

1 Úvod

Historie textu

V roce 2013 jsem vytvořil Příkladný úvod do VHDL, který jsem v létě 2019 inovoval na základě svých zkušeností s výukou. Rozšířil jsem výklad v částech obtížnějších pro studenty a doplnil nové příklady.

Zaměření textu

VHDL patří mezi HDL jazyky ("**H**ardware **D**escription **L**anguages") určené pro návrhy obvodů. Všimněte si, že v jeho názvu není slovo „programming" ale „description". Občas se jím píše i programový kód, zejména když se tvoří simulační testy, avšak VHDL primárně cílí na fyzickou syntézu obvodů. Dobří návrháři při ní "nebuší" mechanicky jakési příkazy, ale soustředí se na strukturu vytvářeného zapojení.

VHDL obsahuje široké spektrum konstrukcí, ale jen jejich část se hodí k fyzické realizaci obvodů:

1. jeho základem je fyzická syntéza obvodů,
2. lze jím popsat i propojení dílčích modulů
3. a pomocí simulací se návrh dá i vyzkoušet a sledovat jeho odezvy na generované vstupy.

Učební text se zaměří na fyzickou syntézu "dataflow" návrhovým stylem, který tvoří základ VHDL, a "structural" stylem, jenž představuje neoddělitelnou součást větších návrhů.

- **dataflow** styl sestavuje obvody **concurrent** příkazy, které popisují tok dat mezi vstupy a výstupy. Dá se přímo syntetizovat, a proto se právem označuje za základní styl. I obvody vytvořené dalšími styly totiž obsahují dataflow části, a tak se mu budeme primárně věnovat.
- **behavioral** styl popisuje obvody **sequential** (sekvenčními) příkazy, a to kódem, který se blíží klasickému programování. Usnadňuje návrhy a řada výukových materiálů se věnuje pouze jemu. Zde však nutno vzít na vědomí, že behavioral styl nelze přímo syntetizovat. Překladač musí napřed celý kód překlopit na concurrent příkazy a teprve pak z nich sestaví obvod.
  Pokud někdo nerozumí dataflow, těžko si dokáže představit, co vlastně vznikne z jeho kódu. Styl behavioral sice usnadňuje návrh, ale musí se velmi rozumě používat, jinak se jím vygenerují odstrašující obvodové paskvily. Bude vysvětlený až na přednáškách.
- **structural** styl specifikuje propojení obvodů vzniklých na základě dataflow či behavioral popisů. Pro velké návrhy se lépe hodí než složité změti propojek v obřích symbolických schématech.
  U menších návrhů, kde zapojení zůstává pořád ještě přehledné, se schémata používají i v profesionální praxi. V Quartusu se kreslí v **B**lock **D**iagram/Schematic **F**ile (soubory *.bdf).

Popis stylem dataflow se nejlépe hodí pro části popsané kombinačními obvody, kde závislost výstupů na vstupech lze vyjádřit logickými operacemi. Dovoluje psát i sekvenční obvody, avšak tam již není výhodný, takže se pro ně volí styl behavioral. Budeme se tedy věnovat jen kombinačním obvodům, ale ty tvoří i součást návrhů se sekvenčními obvody, které si necháme na přednášky, a v nich se rovněž nevyhnete dataflow částem, protože některé příkazy existují jen v něm, například vytváření instancí obvodů, téma celé kapitoly 6.

Prerekvizity

Budeme přepokládat, že čtenáři:

- znají celou prerekvizitu **APOLOS**, v níž se popisují základy logických obvodů a formáty čísel integer. Najdete ji na http://dcenet.felk.cvut.cz/edu/fpga/navody.aspx .
- instalovali si bezplatnou verzi vývojového prostředí **Altera Quartus II**. Ta nabízí stejnou funkčnost jako komerční verze, oproti ní překládá jen trochu pomaleji, protože využívá pouze jedno jádro procesoru a nedovoluje dílčí kompilace, což vůbec nevadí u menších návrhů, s nimiž budeme

v předmětu pracovat, protože jejich překlad se tím zpomalí jen o několik sekund. Potřebné soubory se nachází na http://dcenet.felk.cvut.cz/edu/fpga/install.aspx.

- používají **Quartus verze 13.0.1.232**, poslední podporující FPGA obvody řady Cyclone II, které máme ve vývojových deskách DE2 (http://dcenet.felk.cvut.cz/edu/fpga/de2.aspx), na nichž provádíme laboratorní cvičení. Nemá význam instalovat si vyšší verzi Quartusu.
- umí si **vytvořit projekt Quartusu II pro desku DE2,** a to včetně přiřazení "Pin Assignments" desky DE2 ze souboru "DE2_pin_assignments.csv". Návod se nachází na: http://dcenet.felk.cvut.cz/edu/fpga/navody.aspx. Postup si ukážeme hned na úvodním cvičení.

Práci usnadní i komerční editor **Sigasi** (http://www.sigasi.com), který zvýrazňuje chyby při psaní VHDL kódu. Jeho bezplatná verze nabízí vše pro menší soubory a pro větší se přepne na omezené možnosti, ale stále detekuje chyby přímo při psaní, což Quartus neumí.

VHDL editory prostředí Quartus a Sigasi nezamykají přístup k souborům, ale načítají si je celé do paměti, a tak lze externí aplikací přepsat soubor na disku, který mají právě otevřený. Detekují si automaticky jeho změnu a nabídnou aktualizaci. Lze tedy paralelně používat více editorů.

Sigasi se vám přepne v budově FEL na Karlově náměstí na plnou verzi, zadáte-li si náš licenční server. Na něj lze přistupovat výhradně z lokální sítě felk.cvut.cz — podmínky naší školní licence Sigasi.

@Příklady

Text obsahuje 20 řešených příkladů, označených římskými číslicemi @Příklad I. až @Příklad XX., které postupně uvádí do problematiky. Čtenáři si porozumění textu otestují na 9 úlohách, které najdou ve vložených kapitolách ***Cvičná úloha 1 až ***Cvičná úloha 9. Po jejich samostatném vyřešení si mohou své kódy srovnat s výsledky uvedenými v kapitole 8 - Příloha A: Řešení cvičných úloh.

Proč zrovna VHDL?

V USA se převážně používá HDL jazyk Verilog, se syntaxí podobnou jazyku C, zatímco v Evropě se více ujal **VHDL** ("**V**ery-high-speed integrated circuits **H**ardware **D**escription **L**anguage"), který se vzdáleně podobá PASCALu, kdysi hojně užívanému jazyku, ale dnes již ojedinělému. Přesněji — VHDL si vypůjčilo svou syntaxi z jazyka ADA určeného pro vojenské systémy, a totéž provedl i PASCAL.

Když jsme se v roce 2009 rozhodovali, zda studenty učit VHDL či Verilog, profesionální návrháři, kteří léta používají oba jazyky, se klonili k VHDL. Jde o jazyk sice upovídaný a typově striktnější než silně typové programovací jazyky jako Java nebo C++, ale přehršle jeho limitů, které začátečníkům komplikuje život, výrazně zvyšuje šanci, že sestavený obvod bude fungovat dle našich představ.

Jazyk Verilog vznikl tři roky před VHDL jako programovací jazyk pro simulaci obvodů. Časem se do něho přidala i fyzická syntéza. Nabízí částečnou podobnost s jazykem C včetně širších konverzí typů. Obé však představuje líbivou pastičku na začátečníky, protože postupy osvojené z C programů se nehodí pro obvody. Profesionální návrháři tvrdí, že lidé, kteří se napřed naučili VHDL, si lépe osvojili správný styl syntézy. A rychle se přeškolili na Verilog, pokud ho později potřebovali, avšak opačně to neplatí.

Zvýraznění syntaxe a nepřesnosti

S omluvou upozorňuji, že barevné zvýraznění syntaxe v kódech VHDL není jednotné. Quartus žel kopíruje vše jako prostý text. Obarvení se provádělo ručně a používala se také různá externí prostředí. Sjednocení se odložilo až na budoucí korekce textu.

2 Základy syntaxe VHDL a příkaz souběžného přiřazení <=

Budeme psát jen ve VHDL‘93. Proč? Quartus umí navolit i inovaci VHDL 2008, která zavedla některé pokročilejší příkazy. Nicméně ModelSim, aplikace pro simulaci obvodů, s níž budeme později pracovat ve cvičení, spolehlivě interpretuje celou verzi 93, ale dovolí jen pouhé fragmenty z verze 2008, neboť její konstrukty se obtížně emulují. Kvůli tomu zůstaneme u verze VHDL‘93.

2.1 Základy syntaxe a identifikátory

Komentáře ve VHDL začínají dvojicí pomlček "--" a končí s koncem řádku. Neexistuje sice možnost jak komentovat celé bloky, musíte označit každý řádek zvlášť, ale Quartus II dovede vložit komentářové pomlčky na začátky všech vybraných řádek a případně je zase odebrat.

Nepoužívejte diakritiku, a to ani v komentářích! Překladače ji občas vezmou, jindy zas ne. Spolehlivě rozumí jen čistému ASCII.

Identifikátory začínají malým i velkým písmenem A-Z. Za prvním písmenem mohou následovat malá a velká písmena A-Z a číslice 0-9. Délka identifikátorů není nijak omezená.

Znaky podtržítek _ lze také použít, ale nesmí být na začátku nebo konci názvu a nedovolují se ani dvě podtržítka za sebou.

Povolené názvy jsou třeba: A1, A_1, a12_b7_Z

a nepovolené: A1_ (podtržítko na konci), A__1 (dvě podtržítka), end (klíčové slovo),
A$B (nepovolený znak $)

Pozn. VHDL‘93 sice dovoluje i rozšířené názvy ohraničené znaky \ \ — v těch lze použít i jinak zakázané znaky. Například dovolené rozšířené názvy jsou \123.45\ , \END\ či \A$B\. Jejich užití přináší často problémy, a tak se jim vyhněte. Používají se na speciální operace a automaticky generované kódy.

VHDL nerozlišuje malá a velká písmena. Nicméně bývá dobrým zvykem psát identifikátory a klíčová slova pořád stejně. Pokud napíšete název vstupního signálu jako "CLOCK_50", "clock_50", "Clock_50" i "ClocK_50", pak překladač sice vyhodnotí vše jako stejné označení, ale kód bude přehlednější, budeme-li signál psát pořád stejně, tedy například buď **CLOCK_50** nebo **Clock_50.**

Pozn. Některá vývojová prostředí, např. Sigasi, vypíší varování, pokud se zmíněné pravidlo nedodržuje.

Identifikátor musí být unikátní a nesmí být klíčovým slovem, a těch je hodně. Základní rezervovaná klíčová slova VHDL:

abs, access, after, alias, all, and, architecture, array, assert, attribute,
begin, block, body, buffer, bus, case, component, configuration, constant,
disconnect, downto, else, elsif, end, entity, exit, file, for, function,
generate, generic, group, guarded,
if, impure, in, inertial, inout, is, label, library, linkage, literal, loop, map, mod,
nand, new, next, nor, not, null, of, on, open, or, others, out,
package, port, postponed, procedure, process, pure,
range, record, register, reject, rem, report, return, rol, ror,
select, severity, shared, signal, sla, sll, sra, srl, subtype, then, to, transport, type,
unaffected, units, until, use, variable, wait, when, while, with, xnor, xor

Další vyhrazené názvy mohou přidat i použité knihovny. V předmětu LSP bude pracovat s dvěma nejčastějšími: ieee.std_logic_1164 a ieee.numeric_std.

2.2 Datový typ std_logic

Při návrhu obvodů nevystačíme s logickou '0' a '1'. Výsledný obvod může mít na výstupu vysokou impedanci, popřípadě chceme zadat, že nám na hodnotě nezáleží, apod.

Typ std_logic je výčtový a obsahuje 9 hodnot: '1', '0', 'X', 'Z','U', '-', 'L', 'H', 'W' vícehodnotové logiky zvané MVL-9 (Multi Value Logic 9):

Forcing 1	´1´	Logická 1, v symbolických schématech vytvořených v Quartusu totožná s napájením **Vcc** (Voltage at Common Collector)
Forcing 0	´0´	Logická 0, v symbolických schématech vytvořených Quartusem totožná s napájením označeném jako **GND** (Ground)
Unutilized	´U´	Simulace obvodu hlásí, že není ještě známá hodnota výstupu, jelikož se dosud neprovedla jeho inicializace.
Don’t Care	´-´	Značí nezadanou hodnota, na níž nám nezáleží.
Forcing Unknown	´X´	Neznámá hodnota, simulace ji nedokáže jednoznačně stanovit.
High Impedance	´Z´	Odpojení výstupu, tzv. třetí stav vysoké impedance.
Weak 1, Weak 0, Weak unknown	*´H´,´L´, ´W´*	*Výstupy speciálních obvodů, např. s otevřeným kolektorem. FPGA řady Cyclone je neobsahují, a tak u nich nelze použít.*

Tabulka 1 - Typ std_logic

Datové typy odvozené od std_logic se při VHDL syntéze volí pro veličiny spojené se vstupy a výstupy, neboť překladače je lépe realizují. Logické operace s nimi se definují tabulkami, viz kapitola 9 str. 71.

VHDL obsahuje i datové typy "boolean" a "bit" — oba jsou opět výčtové.

- *Typ boolean s hodnotami TRUE a FALSE lze použít jedině pro podmínky, viz dále.*
- *Typ bit s hodnotami ´0´ a ´1´ se zase užívá jen v simulaci či pomocných výpočetních částech. Nehodí se k syntéze obvodů, protože překladač u něho nedetekuje některé obvodové chyby, jako například dva zdroje signálu ve zkratu.*

Pozor — datové typy std_logic a bit nejsou vzájemně převoditelné přetypováním, lze je konvertovat jedině podmínkou kombinovanou s přiřazením, bude v kapitole 4.2 na str. 30.

2.3 Signal - vodič a souběžné přiřazení <=

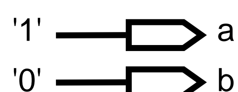
Definice signal vytvoří prvek ekvivalentní vodiči, a **není proměnnou**, tedy variable. Drát je pouhou spojkou! V jazyce Verilog se jeho analogie jmenuje přímo "wire". Přiřazení do prvku signal se provádí pomocí symbolu **<=** nazývaného "concurrent assignment", tedy souběžné, resp. paralelní přiřazení. Všechna taková přiřazení se provádějí současně. Budou-li někde definice signálů:

```
signal a, b : std_logic;
```

a později se na ně připojí '1' a '0' příkazy:

```
a <= '1'; b <= '0';   -- '1' a '0' jsou hodnoty vyctoveho typu std_logic - komentar bez diakritiky!
```

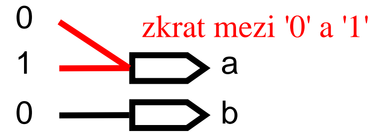
pak se obě přiřazení provedou najednou bez ohledu na jejich pořadí v kódu, neboť v obvodu, který jsme zápisem vytvořili, neexistuje žádná časová priorita, oba vodiče pracují souběžně!



Dále platí striktní pravidlo, že **do prvků signal lze přiřadit jen jednou!** Napíšu-li totiž:

```
a <= '1';
b <= '0';
a <= '0';   -- chyba
```

pak vytvořím obvod



Překladač Quartusu vypíše chybové hlášení "Error (10028): Can't resolve multiple constant drivers". V zapojení jsme totiž do stejného bodu zapojili jak logickou '0', tou obvykle bývá zem, tak logickou '1', ta se často ztotožňuje s napájecím napětím. Ojoj, navrhli jsme jiskřící zkrat mezi napájením a zemí:-(

Poznámka: Pokud v nějakém VHDL kódu uvidíte vícenásobné přiřazení, pak se jedná o speciální sekce (jako process, function a procedure) psané ve "VHDL behavioral modeling style", v němž se využívá vyšší úroveň abstrakce. Překladač si ho konvertuje na dataflow, a to zavedením pomocných signálů tak, aby vyhověl striktní podmínce jediného přiřazení. Teprve pak může sestavit obvod.

2.4 Operace s typy std_logic a boolean

VHDL dovoluje řadu operací. S výčtovým typem std_logic můžeme provádět následující operace.

Priorita		Datový typ výsledku	Operátor
Nejvyšší	⬇	stejný jako operandy [1]	**not**
		boolean (true, false)	**= /=**
Nejnižší		stejný jako operandy [1]	**and or nand nor xor xnor** [2]

[1] Logické operátory jsou v ieee.std_logic_1164 definované jak pro typ std_logic , kdy vracejí výsledek typu **std_logic**, a také i pro typ Boolean; u něhož jejich výsledkem je opět Boolean.

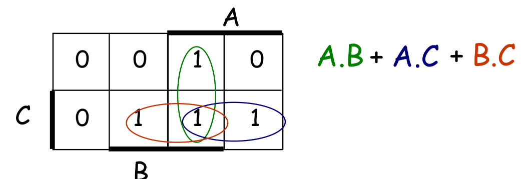
[2] Všimněte si, že logické operace nemají vůči sobě prioritu!

Tabulka 2 - Operace s std_logic

Vezměme si logickou funkci majority se třemi vstupy A, B a C, která má svůj výstup Y v logické ´1´ jen tehdy, pokud jsou v logické ´1´ alespoň dva její vstupy.

Funkci majorita navrhneme buď z Karnaughovy mapy nebo následují logickou úvahou:

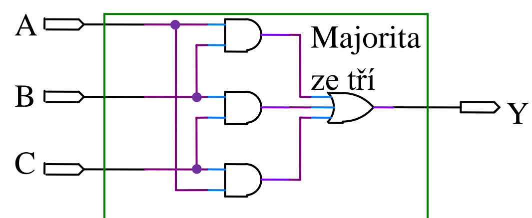
Ze tří vstupů lze vytvořit tři dvojice A-B, A-C a B-C. Bude-li nějaká z nich mít oba členy v '1', pak výstup Y je vždy '1'. Spojíme tedy členy dvojic AND operátorem, který dává na výstupu ´1´ jen tehdy, pokud jsou oba vstupy v '1'.



Tři AND pak svážeme OR operacemi, jejichž výsledek bude ´1´, pokud alespoň jeden AND bude v '1':

Y = (A and B) or (A and C) or (B and C)

Navržený obvod lze sice v Quartusu vytvořit grafickým editorem jako *.bdf (**B**lock **D**iagram/Schematic **F**ile) soubor majorita.bdf, ale my si napíšeme ve VHDL.



Obrázek 1 - Majorita ze tří

Předpokládejme, že máme někde definované signály **a**, **b**, **c** a **y**, pak VHDL rovnice majority bude:

```
y <= (a AND b) OR (a AND c) OR (b AND c);          --Eq2.4-1
```

Závorkami jsme specifikovali pořadí operací.

Poznámka: V jazyce C i Java se pro pohodlnost zavedla priorita operátoru && nad || , ač ji Booleova algebra nezná. Ve VHDL musíte jasně napsat, v jakém pořadí se mají logické operace provést.

Pokud bychom v předešlém výrazu vynechali závorky a napsali ho jako

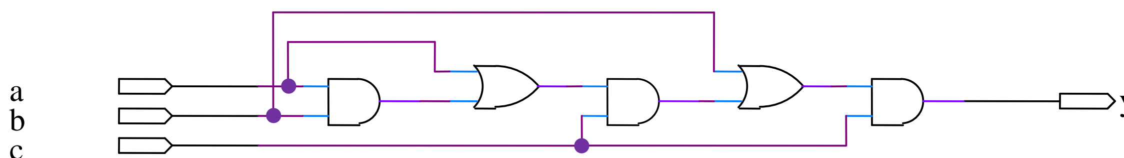
y <= a AND b OR a AND c OR b AND c; --Eq2.4-2

pak překladač vypíše chybu: *Error (10500): VHDL syntax error near text "OR"; expecting ";"*, protože nekompromisně vyžaduje oddělení různých typů logických operací závorkami.

Můžeme několikrát za sebou použít jeden stejný binární operand, aniž vkládáme do zápisu závorky, třeba psát "a XOR b XOR c", ale přidání odlišného operandu, např. OR, již vyžaduje závorkování.

Starší VHDL překladače by naši nezávorkovanou rovnici (Eq2.4-2) nekontrolovaly a předpokládaly, že výraz podle pořadí operandů, čímž by vznikl obvod s odlišnou funkcí od majority:

y <= (((((a AND b) OR a) AND c) OR b) AND c); --Eq2.4-3



Obrázek 2 - Schéma odpovídající přiřazení `y <= a AND b OR a AND c OR b AND c;`

Kvůli tomu se doplnila přísná závorková kontrola do novějších verzí Quartusu, i do námi používané 13.

Typové std_logic pastičky

Kdyby někdo výše rovnici Eq2.4-1 napsal velekrkolomně takto:

y <= (a='1' AND b='1') OR (a='1' AND c='1') OR (b='1' AND c='1'); --Eq2.4-4

dočkal by se záhadné chyby *Error (10327): VHDL error can't determine definition of operator ""="" -- found 0 possible definitions.*

Zavedeme-li si však nový vodič v sekci VHDL kódu vyhrazené pro definice:

signal x : boolean;

a upravíme rovnici Eq2.4-4 na

x <= (a='1' AND b='1') OR (a='1' AND c='1') OR (b='1' AND c='1'); --Eq2.4-5

pak se mu bez problémů přeloží.

Typy std_logic můžeme sice porovnávat i pomocí operátorů rovnosti = a nerovnosti /= , ale ty dávají výsledek výčtového typu boolean. Ten má pouze dvě hodnoty TRUE a FALSE, které nejde přetypováním konvertovat na 9-hodnotový typ std_logic.

Poznámka: Převod lze provést podmíněným přiřazením, s nímž se seznámíme v kapitole 4.2 na str. 30.

V rovnici majority nemají nadbytečné porovnávací = operátory smysl. Vždyť vše lze vyjádřit std_logic!

Výsledek boolean	Ekvivalent dávající výsledek typu std_logic	
a='1'	**a**	
a='0'	NOT a	
a \= b	**a** XOR **b**	XOR je non-ekvivalencí ≠
a=b	**a** XNOR **b**	negované XOR je ekvivalencí **=**

Pozor, následující rovnice:

y <= (a='1' AND b) OR (a AND c) OR (b AND c); --Eq2.4-6

je zcela špatně a generuje další tajuplnou chybu *Error (10500): VHDL syntax error near text ";"; expecting "end", or "(", or an identifier, or a concurrent statement.*

Výsledkem prvního porovnání a='1' je totiž typ boolean, a ten nelze kombinovat s typem std_logic. Na prohřešek se přijde až kdesi v pozdější fázi překladu, a tak se objeví krajně podivné hlášení.

Quartus nepřekládá na instrukce assembleru ve stylu kompilátorů klasických programovacích jazyků, ale v prvním průchodu si vytváří meta-schéma obvodu (RTL Map), které dále optimalizuje. Pokud se při jeho tvorbě neuspěje, vypíše někdy mnohoznačné hlášení. Chyba může být i kdekoli před místem, kde došlo k zádrhelu.

Na druhou stranu buďme rádi, že VHDL závisí extrémně na typech, protože jejich přesným používáním se vychytá řada chyb. Pro ladění sestaveného obvodu nelze použít debugger, krokovat jím dráty a číst si hodnoty vodičů. Vidíme či měříme jen signály, které jsme si sami vyvedli na výstupy. Ostatní zůstávají skryté, a proto pomůže pečlivý návrh, jelikož jeho výsledek se ladí obtížněji než klasický program.

Souhrn: Typ std_logic se realizuje jako výčtový typ, jehož hodnoty byly uvedené na straně 7. Budeme-li mít definici:

```
signal t, x, y, z : std_logic;
```

pak do vodičů můžeme přiřadit jedině typ std_logic, tedy buď výsledek logické operace s typy std_logic nebo konstanty jeho výčtového typu, např.:

```
t<=´Z´; x<=´1´;  y<=´0´; z <= x AND y;
```

Konstanty ´1´ a ´0´ jsou v uvozovkách, ale zde nejde o znaky (VHDL typ character), ale o hodnoty výčtového typu std_logic definovaného v knihovně ieee.std_logic_1164 seznamem hodnot, ve stylu analogickém typům enum v Java či C#, zjednodušeně takto (*přesná definice je v kapitole 9 na str. 71*):

```
type std_logic is ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

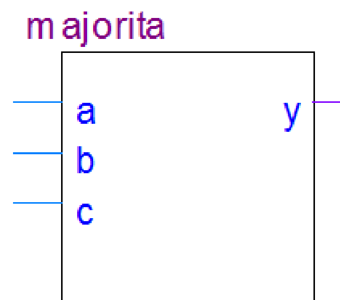
2.5 @Příklad I. - VHDL kód majority

VHDL kód majority není složitý. Vyjdeme z univerzální šablony, která bude vysvětlena hned v další části. Budeme ji používat i k dalším návrhům. Zvýraznění ukazuje námi doplněné části:

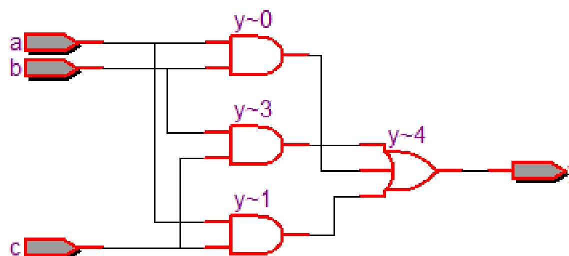
VHDL kód majorita.vhd

```
-- Majority function 2/3
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity majorita is
	port (	a, b, c : in std_logic;
		y : out std_logic );
end;
architecture dataflow of majorita is
begin
	y <= (a AND b) OR (a AND c) OR (b AND c);
end;
```

Schematická značka obvodu



Quartus z VHDL kódu sestaví i schematickou značku pro editor schémat. Obsahuje také prohlížeč interního zapojení obvodu, který vytvořil z našeho kódu během prvního průchodu (RTL Map Viewer):



Obrázek 3 - Vnitřní schéma majority vytvořené překladačem

Pomocné identifikátory, které si překladač vygeneroval, obsahují znak ~ . Ten nesmíme v našich identifikátorech napsat. Quartus se ho rezervoval sám pro sebe.

V příloze C na str. 73 najdete úplný postup jak vytvořit VHDL soubor, přeložit ho, vygenerovat jeho schematickou značku, zobrazit si interní RTL logické schéma a otestovat si návrh simulací.

2.6 VHDL šablona

Zde si vysvětlíme šablonu VHDL kódu, kterou jsme použili k napsání majority. Jakmile pochopíte její strukturu, rychle vytvoříte další VHDL kódy — nakopírujete šablonu a opravíte vyznačená místa.

Odkazy na řádky L01 až L15 se zde zavedly jen za účelem výkladu a str. je stránka s jejich vysvětlením.

VHDL kód	Řádka	str.	
-- comment	L01	12	*nepovinný, ale velmi vhodný komentář*
library ieee;	L02	12	*aktivace knihovny*
use ieee.std_logic_1164.all;	L03		*typy odvozené od std_logic*
use ieee.numeric_std.all;	L04		*číselné typy signed a unsigned*
entity our_name is	L05	12	*začátek entity - vnější vzhled obvodu*
port (	L06	13	*definice I/O*
-- inputs/outputs	L07		*sem napíšeme definice vstupů a výstupů*
);	L08		
end;	L09		*konec entity*
architecture dataflow of our_name is	L10	14	*popis implementace obvodu*
--definitions	L11		*zde vložíme definice pomocných elementů*
begin	L12		
--implementation	L13		*sem napíšeme popis činnosti obvodu*
end;	L14		

Trojice pojmenování VHDL souboru - our_name

Pro soubor musíme zvolit platný VHDL identifikátor unikátní v rámci našeho návrhu (tj. Quartus projektu). Námi vybraný identifikátor musíme bezpodmínečně napsat na tři místa!

1. Když soubor ukládáme, bude jeho jménem, k němuž přidáme jen příponu *.vhd
2. Stejně tak pojmenujeme entitu
3. a napíšeme ho do hlavičky architektury mezi klíčová slova of a is.

Zvolíme-li například majorita, pak soubor uložíme jako majorita.vhd, a v šabloně opravíme:

```
entity majorita is   -- L05
```

a dále ještě

```
architecture dataflow of majorita is        -- L11
```

Pravidlo trojího pojmenování nesmíte porušit. Jinak překladač návrh buď nenajde, nebo ho nedokáže zapracovat do vytvářeného zapojení. Podobný prohřešek bývá častou chybou, a proto pokaždé zkontrolujte, zde jste splnili trojí pojmenování.

Soubor musíme uložit do adresáře našeho Quartus projektu a celá cesta k němu musí být linuxového typu, tedy žádné mezery v cestě, diakritika nebo Windows speciální znaky.

Poznámka: Quartus je nativní linuxová aplikace, která ve Windows běží pod Cygwin .

Ve výukových laboratořích ČVUT-FEL se Quartus projekty nesmí umístit na síťové disky. Ty mají znak $ na své skutečné cestě, tedy na té, kterou aplikace dostanou od OS. Nenechte se mýlit tím, že znak $ nevidíte ve Windows průzkumníku, ten často ukazuje jen zástupný odkaz.

Nehodí se ani USB disky. Quartus si během překladu vytváří záplavu pomocných souborů v místě, kde se nachází projekt. Začne-li zapisovat na USB, překlad se protáhne z desítek sekund na desítky minut.

L01 šablony - komentář

Jde o doporučený řádek, či skupinu řádků, popisující účel kódu, aby se na to časem nezapomnělo☺.

L02 až L04 šablony - knihovny

Knihovny představují výhodu VHDL oproti Verilogu, který je neumí. Pod knihovnami si můžete představit vzdálené analogie příkazů using jazyka C# či import v Javě. Nejde o přesný ekvivalent #include jazyka C, to vkládá celé soubory, zatímco ve VHDL se zavedou jen definice, ale výsledný efekt je podobný.

Příkaz "library" (řádek **L02** šablony) aktivuje knihovnu, zde standardní ieee navrženou v "Institute of Electrical and Electronics Engineers (IEEE)". Pozn. Zavedená česká výslovnost IEEE je "aj-tripple-í".

Na **L03** se z ní vybírá balíček ("package") deklarující standardní logiku "std_logic_1164" a z něho vezmeme vše (all). Stejně tak se připojí i numeric_std. *Pozn. Volba "all" bývá nejčastější, ale z knihovny lze rovněž vybrat i jeden prvek, což se někdy hodí pro řešení konfliktů u složitějších projektů.*

- **ieee.std_logic_1164** - obsahuje definice a operace s typy založenými na std_logic.
- **ieee.numeric_std** - obsahující doporučené definice operací s typy integer, signed a unsigned. Knihovna nahrazuje starší ieee.std_logic_arith, u níž existovala řada různých verzí napsaných jednotlivými firmami, což ztěžovalo přenositelnost kódů. IEEE výbor kvůli tomu vytvořil mezinárodní standard ieee.numeric_std. V nových kódech se dnes používá pouze numeric_std.
  V našich úvodních VHDL kódech zatím nemáme operace s čísly, ty použijeme až později, ale knihovna se i tak vložila, aby šablona zůstala univerzální. Quartus má všechny nejčastější knihovny předkomplikované, takže jejím přidáním nijak se překlad nijak nezpomalí. (Testováno:-)

L05 a L09 - blok entita

Návrh obvodu obsahuje povinnou deklaraci bloku entity, který v naší šabloně začíná na **L05**:

```
entity our_name is
```

kde entity je klíčové slovo a "our_name" představuje námi přiřazený název obvodu, ten musí být shodný se jménem VHDL souboru a v architektuře. Blok entity může obsahovat i několik definic (*ale nemusí mít ani jednu, u testbench testů je prázdný*). Náš příklad má jen definici "port" (**L06**) vstupů a výstupů.

Na L09 blok entity končí klíčovým slovem end;

Blok entity určuje vnější vzhled obvodu, čili jeho vstupy a výstupy viditelné uživatelům. VHDL blok entity se obecně chová jinak než třídy známé z C++, Javy a C#, nicméně můžeme uvést nepatrnou analogii. Název uvedený v úvodu bloku entity koresponduje se jménem třídy — a od třídy, zde od entity, vytváříme instance, jimiž jsou v našem případě obvody sestavené podle předpisu pospaném v architektuře. Ty vznikají analogií konstruktoru, kterou je ve VHDL příkaz map, jemuž věnujeme mu kapitolu 6.

Zakončení bloku entity lze napsat zkráceně, tak jako v našem příkladu, nebo případně zopakovat za end a i klíčové slovo entity nebo také název, což se povoluje od verze VHDL'93 (používané v kurzu LSP). Náš blok entity může mít tedy tvar:

entity our_name is -- deklarace end;	entity our_name is -- deklarace end entity;	entity our_name is -- deklarace end our_name;	entity our_name is -- deklarace end entity our_name;

Tabulka 3 - Možnosti zakončení deklarace entity ve VHDL'93 a vyšším

Zkracování ukončujících "end" se povoluje i u některých dalších deklarací, např. u bloku architecture na rádcích **L10** až **L14**, kde možno uplatnit stejný princip. Volba zakončení závisí na návrháři.

U kratších bloků se volí jen end, ale u delších částí bývá přehlednější zopakovat klíčové slovo úvodní deklarace bloku, zde entity, nebo přidat i název bloku či obojí, aby se vědělo, k čemu se end vztahuje.

L6 až L08 - port - vstupy a výstupy

Definice port se může objevit jen uvnitř deklarace entity a nejvýše jedenkrát. Specifikuje vstupy a výstupy typu **signal**, které lze vidět zvnějšku, tedy jakési analogie prvků "public" třídy.

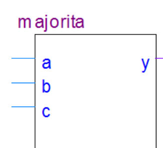
Definuje seznamy signálů (***signal-names***), jejich datové typy (***signal-type***) a módy (***mode)*** určujícími druh vstupu či výstupu.

Všimněte si, že za posledním signal-type chybí středník — jeho napsání bývá častou syntaktickou chybou.

```
entity entity-name is
  port (signal-names : mode signal-type;
        signal-names : mode signal-type;
        ...

        signal-names : mode signal-type);
end entity-name;
```

Náš kód majorita.vhd definoval tři vstupy a, b, c a jeden výstup y, a to popisem:

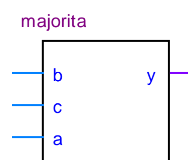
```
entity majorita is
port ( a, b, c : in std_logic;
       y : out std_logic );
end;
```



V definicích port lze použít buď seznam signálů oddělených čárkami, nebo každý signál psát na samostatný řádek, což umižní na zbytek řádky doplnit komentář jeho významu.

Pořadí definic v bloku port je sice libovolné, ale určujeme tím současně pořadí vstupů a výstupů na vygenerované schematické značce určené k použití v symbolickém schématu. Změníme-li se jejich pořadí či názvy, značka se musí znovu vygenerovat, viz v příloha C, kapitola 10.5 str. 79. Nedošlo-li k žádné změně v port sekci, předchozí značka zůstává pořád platná.

```
entity majorita is
 port ( b : in std_logic;
        y : out std_logic;
        c : in std_logic;
        a : in std_logic );
end;
```

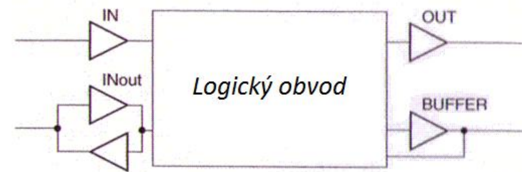


Mód vstupu a výstupu:

- Vodič portu vstupu módu **in** smíme pouze číst, protože bude buzený ze zdroje signálu přivedeného zvnějšku, tedy ze zapojení, které se připojí k našemu obvodu. Do vstupu nelze zapisovat, aby se nevytvořil zkrat. Mód in má tedy charakter read-only.
- Naproti tomu mód **out** specifikuje náš výstup, jemuž můžeme zvnitřku obvodu přiřadit hodnotu, ale nelze ho číst. Mód out je tedy write-only. A pozor, jelikož jde o vodič, tedy typ signal, smíme do něj přiřadit pouze jednou.

Pokud musíme hodnotu výstupu nutně i číst, lze použít mód **buffer**, pro který překladač vyhradí logický element, jelikož z výstupu vede drát zpět do vnitřku obvodu.

Posledním módem je typ **inout,** který dovoluje data jak číst tak do nich zapisovat. Vyžaduje ovšem složitý prvek obousměrného budiče, a tak se používá jedině výjimečně, například pro sběrnice. Na desce DE2 ho má třeba audio výstup, jehož interobvodová I2Csběrnice je obousměrná.



Obrázek 4 - Realizace módů v obvodu

U návrhů určených k fyzické syntéze obvodů používejte jen módy **in** a **out**, protože jsou nejjednodušší na implementaci. Dávají jim přednost všichni profesionální návrháři.

L10 a L14 šablony Architektura

Blok architektury popisuje operace vlastního obvodu. V kódu majorita začíná na **L10**, a to klíčovým slovem architecture, za nímž následuje jméno vytvářené architektury a název entity, pro niž je architektura určená. Poté následuje příkazový blok begin end;

V řádku **L11** bychom mohli definovat i vnitřní pomocné signály, ale naše majorita je nepotřebovala.

Pro zakončení bloku architektury klíčovým slovem end (**L14**) platí podobné možnosti jako pro zakončení bloku entity. Za klíčovým slovem end možno zopakovat i klíčové slovo architecture a případně i její název dataflow, podobně jako u zakončení bloku entity, viz Tabulka 3 na straně 12.

Poznámka: Bloky entity *a* architekture *si můžeme představit jako části elektronického zařízení, například digitálního budíku. **Entita** popisuje prvky vyvedené na povrch jeho krabičky, jako třeba tlačítka, bzučák či zobrazovací elementy. **Architektura** pak koresponduje s vnitřním zapojením budíku, které je skryté před vnějším pozorovatelem uvnitř krabičky.*

Budeme-li budík inovovat, můžeme třeba změnit jen ovládací prvky na jeho povrchu, avšak vnitřní zapojení necháme stejné — tedy provádíme jedině úpravy entity, čili vnějšího vzhledu. Stejně tak lze předělat vnitřní zapojení, tj. architekturu, při zachování stejného ovládání, nebo upravovat oboje.

Název architektury dataflow byl zvolený podle VHDL stylu, který jsme použili. Není však žádným pravidlem. Další hodně častý název bývá synth a obvodům s hodinovým signálem se zpravidla dává pojmenování rtl (register-transfer level).

Název architektury, zde dataflow, není globálním identifikátorem a patří pouze své entitě. Jinými slovy — ve všech entitách v dalších souborech můžeme jejich architektury opět nazvat dataflow.

Poznámka: Povinný název architektury nabízí možnost vytvořit několik různých architektur pro jednu entitu, tj. více bloků architecture*, což se hodí například tehdy, potřebujeme-li jednu funkci obvodu pro jeho samotnou realizaci a další k urychlení během simulací. Používání více architektur, samozřejmě odlišených různými názvy, představuje už pokročilejší téma, protože musíme do projektu přidat VHDL soubor s příkazem* configuration*. V kurzu LSP vystačíte u každé entity s jedinou architekturou.*

2.7 @Příklad II. - VHDL kód 4-vstupového XOR

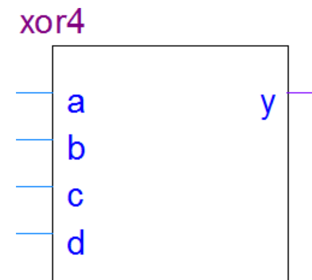
Operace XOR je asociativní a komutativní, viz https://en.wikipedia.org/wiki/Exclusive_or, a tak u ní nezáleží na pořadí operací. Následující VHDL výrazy vedou na zcela identické logické funkce:

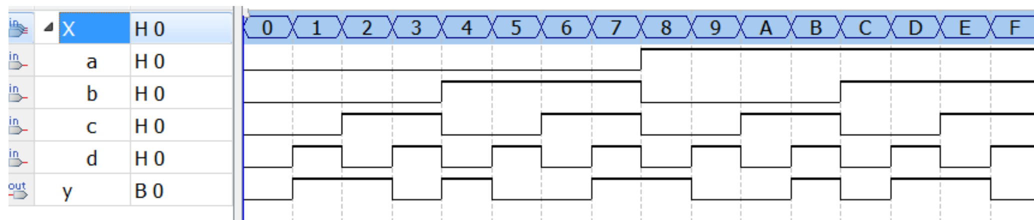
```
y1 <= (a XOR b) XOR (c XOR d);   y2 <= (((a XOR b) XOR c) XOR d);
y <= a XOR b XOR c XOR d;
```

Zvolme tedy poslední jednoduchý zápis a pomocí naší šablony sestavíme obvod 4-vstupového xor.

VHDL kód - xor4.vhd | **Schematická značka obvodu**

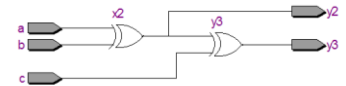
```
-- 4-input XOR
library ieee;   use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity xor4 is
        port (   a, b, c, d : in std_logic;
        y : out std_logic );
end;
architecture dataflow of xor4 is
begin
        y <= a XOR b XOR  c XOR d;
end;
```





Pokud si xor4 vyzkoušíme v simulátoru, pak uvidíme, že dává výstup '1', pokud má lichý počet vstupů v logické '1' — jinými slovy, 4-vstupové XOR testuje, zda vstup má lichou paritu.

Poznámka: Každé vícevstupové XOR dává '1' na výstupu jen při lichém počtu svých vstupů v '1'. Tvrzení lze dokázat matematickou indukcí. Platí pro 2-vstupové XOR a zůstane platné po přidání dalšího XOR za něj.



2.8 @Příklad III. - Majorita2 s indikací dvou v '1'

Přidejte k majoritě výstup, který hlásí, že majorita má právě dva své vstupy v '1'.

a	**b**	**c**	**y** -majorita	**y2** -dva v 1
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	**1**	**1**
1	0	0	0	0
1	0	1	**1**	**1**
1	1	0	**1**	**1**
1	1	1	**1**	0

Funkci nového výstupu y2 může opět odvodit z Karnaughovy mapy, ale rychleji ji dostaneme logickou úvahou. Výstup y2 je shodný s y, až na stav všech vstupů v '1'.

Nemůžeme však využít již vypočtenou hodnotu y majority, jelikož do y smíme jen jedenkrát zapsat hodnotu a tu již nelze číst!

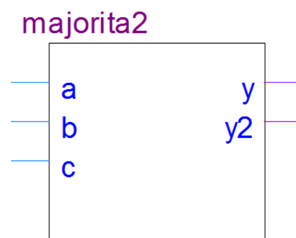
```
y2 <= y AND NOT (a AND b AND c); -- chyba          --Eq2.7-1
```

Lze sice u y změnit I/O mód na buffer, ale existuje elegantnější řešení, a to vytvoření pomocného signálu, který nazveme třeba tmp. Obvod, nazvaný třeba majorita2, bude realizovat funkce:

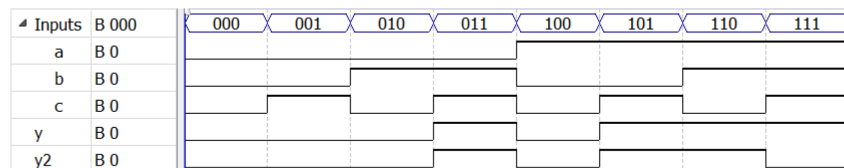
```
tmp <= (a AND b) OR (a AND c) OR (b AND c);          --Eq2.7-2
y<= tmp; y2 <= tmp AND NOT (a AND b AND c);          --Eq2.7-3
```

VHDL kód majorita2.vhd | **Schematická značka obvodu**

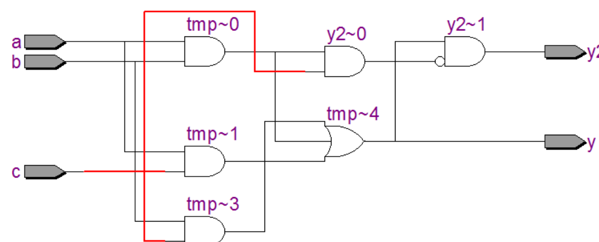
```
--Majority 2/3 plus the indication of 2 true inputs
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity majorita2 is
        port (  a, b, c : in std_logic;
                y, y2 : out std_logic );
end;
architecture dataflow of majorita2 is
signal tmp : std_logic;
begin
   tmp <= (a AND b) OR (a AND c) OR (b AND c);
   y<= tmp; y2 <= tmp AND NOT (a AND b AND c);
end;
```



Výstup simulátoru:



Vnitřní schéma obvodu sestavené Quartusem v úvodním kroku:



Všimněte si, že překladač šikovně využil člen (a AND b) ve výrazu pro tmp k realizaci 3-vstupového AND. Provádí tak zvanou skupinovou minimalizaci, při níž se snižuje celkový počet operací ve všech funkcích. Ta bude tématem přednášek LSP.

Výsledné schéma obvodu představuje pouze první krok překladu a bude dále redukované, avšak vyplatí se na něj vždy podívat, protože popisuje, jak Quartus porozuměl našemu VHDL kódu.

Signály u kombinačních obvodů znamenají jen pojmenované dráty, a tak **definicemi pomocných signálů nezvýšíme složitost obvodu**, protože vodiče nealokují paměť na rozdíl od proměnných v programech.

Pokročilá poznámka pro úplnost: Pokud bychom předepsali, že <= se nahrazuje vodičem s registrem, pak i pomocné signály mohou zvýšit složitost. Něco takového však vyžaduje vložení detekce hrany hodinového signálu, což lze sice provést i dataflow *stylem, ale s komplikacemi, a tak se podobné operace píší VHDL* behavioral *stylem, který si necháme až na přednášky. Bez detekce hrany hodinového signálu zůstávají všechny signály pouhými dráty opatřenými jmenovkami!*

2.9 *** Cvičná úloha 1: - Majorita123

Rozšiřte si majoritu2 o další dva výstupy. Prvním bude výstup y3 signalizující 3 vstupy v '1' a dalším výstup y1, který hlásí, že právě jeden vstup je v '1'. Až si celý návrh uděláte sami, můžete si ho porovnat s možným řešením uvedeným na str. 58. Napíšete-li kód trochu jinak, ale se stejnou funkčností, pak jste také navrhli správný obvod.

2.10 Datový typ std_logic_vector

Někdy potřebujeme manipulovat se skupinami vodičů jako s jedním celkem, což ve VHDL dovoluje datový typ std_logic_vector, který představuje jednorozměrné polem složené z prvků std_logic. Jde o mírně obtížnější téma, protože k němu již neexistuje rozumná analogie v C programech.

- Typ std_logic_vector nemá žádnou číselnou hodnotu — jde o pouhý svazek číslovaných vodičů. Počet vodičů může být od jednoho prvku výše.
- Jedná se sice o jednorozměrné pole, jehož indexy jsou nezáporná čísla (VHDL datový typ natural), ale ty mohou běžet vzestupně (to) nebo sestupně (downto) a začínat libovolnou natural hodnotou.
- **Upřednostňuje se sestupné číslování**, ačkoli to se definuje delším klíčovým slovem downto ☺, jelikož pole pak začíná vyšším prvkem a řazení jeho prvků odpovídá tak uspořádání binárního čísla, u něhož leží bit s nejvyšší vahou vlevo.
- S typem std_logic_vector lze provádět stejné operace jako s std_logic, viz Tabulka 2 na str. 8, provedou se po jeho jednotlivých členech.
- **A nemáme zde žádnou paměťovou buňku**! Typ std_logic_vector není přesnou analogií vektoru. Když se mu přiřadí nová hodnota, vytvoří se tím jen propojka v obvodu! Nejde o žádné uložení dat, ale o obyčejný drát. Změní-li se hodnota zdroje, okamžitě se přenese vodičem na cíl přiřazení.

Typ std_logic_vector si lze nejlépe představit jako řadu vodičů s koncovkami, která mají pořadová čísla, avšak jejich číslování začalo od nějaké nezáporné hodnoty a postupovalo sestupně či vzestupně.

signal A, B, C: std_logic_vector (7 downto 0);

začátek->	7	7	7
	6	6	6
	5	5	5
	4	4	4
	3	3	3
	2	2	2
	1	1	1
konec ->	0	0	0

signal Z: std_logic_vector (1 to 16);

začátek->	1
	2
	3
	4
	5
	6
	7
	8
	9
	10
	11
	12
	13
	14
	15
konec->	16

Konstanty typu std_logic se ohraničují ' ' apostrofy a u typu std_logic_vector naopak " " **uvozovkami**, a to i v případě, když vektor má délku jen jeden člen. Vektorům totiž odpovídají řetězce.

> Obecně je prvek '0' typu std_logic, ale "0" je typu std_logic_vector (0 downto 0), tedy vektor s jedním členem. Skupinu vodičů s jedním členem lze samozřejmě definovat třeba i jako signal xv : std_logic_vector (100 to 100); U jednoprvkového vektoru nezáleží na směru a jeho počáteční/konečný index může být libovolné kladné číslo. Do celého výše xv pak zapisujeme logické konstanty 0 a 1 s uvozovkami, tedy například jako xv<="1";

Indexy se píší v kulatých závorkách! Například, na jediný prvek výše uvedeného xv přistupujeme jako xv(100), tedy dle indexu uvedeného v jeho definičním rozsahu.

Sdružuje-li vektor více vodičů, lze vybrat jen jeden z nich nebo jejich skupinu, tedy podvektor. Pro výše uvedené definice lze použít třeba:

Operace	*Způsob realizace*
B(5) <= A(0);	Vodič B(5) se spojí s vodičem A(0). Oba jsou typu std_logic.
A(1) <= '0';	Na vodič typu std_logic ve vektoru A na indexu 1 se připojí '0'.
C <= "01010110";	Vodiče v C se v pořadí svého číslování připojí na uvedené hodnoty: C(7)<='0'; C(6)<='1'; C(5)<='0'; C(4)<='1'; C(3)<='0'; C(2)<='1'; C(1)<='1'; C(0)<='0';
B(7 downto 4) <= A(5 downto 2);	B(7) <= A(5); B(6) <= A(4); B(5) <= A(3); B(4) <= A(2);

Při výběru podskupiny se musí **zachovat směr číslování** zadaný v definici, tj. downto nebo to. Směr se nesmí obrátit. Napíšeme-li Z(15 downto 14), pak překladač ohlásí chybu: "*…range direction of object slice must be same as range direction of object…*".

Pozn. Editor schémat (.bdf) v Quartusu specifikuje rozsahy polí zhuštěnými zápisy ve stylu Pascalu. Schématu se signály A[2..0], Z[1..7] a A[1] odpovídají ve VHDL zápisy: A(2 downto 0), Z(1 to 7) a A(1).*

Přiřazení konstanty do std_logic_vector o počtu prvků **dělitelném 4** lze zkráceně zapsat hexadecimální notací se znakem X před první uvozovkou, tedy X"5F" je totožné s "01011111".

Logické operace s typy std_logic_vector se provádí po jednotlivých prvcích. Zápis:

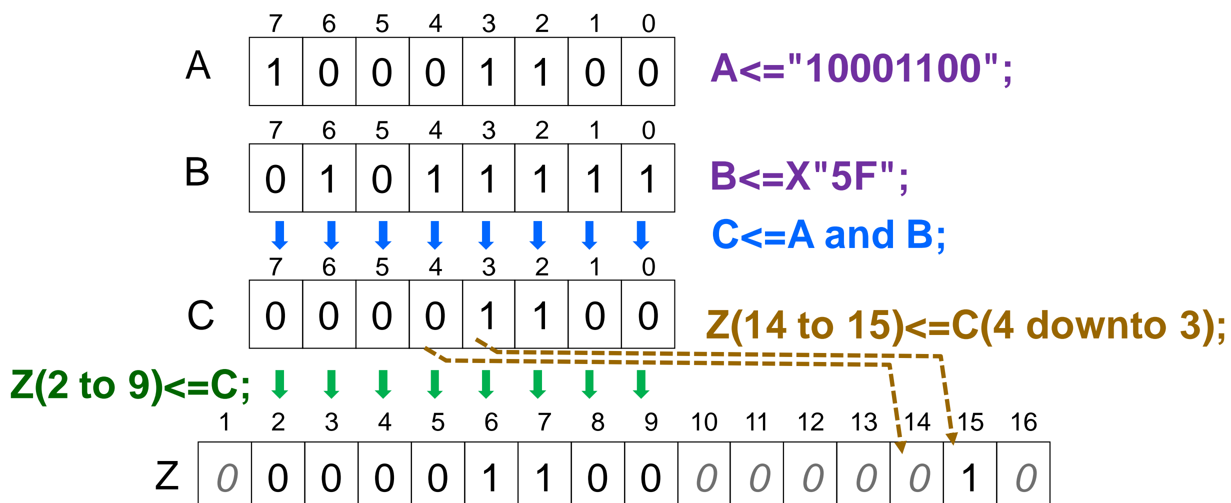
```
C <= A AND B;
```

se vykoná stejně, jako kdybychom zdlouhavě napsali:

```
C(7) <=A(7) AND B(7); C(6) <=A(6) AND B(6); C(5) <=A(5) AND B(5); C(4) <=A(4) AND B(4);
C(3) <=A(3) AND B(3); C(2) <=A(2) AND B(2); C(1) <=A(1) AND B(1); C(0) <=A(0) AND B(0);
```

Vše nejlépe přiblíží příklad použití výše uvedených definic. V obrázcích se z prostorových důvodů kreslila pole naležato a levá strana odpovídá počátečnímu (hornímu, počátečnímu) prvku:

```
A<="10101100";  B<=X"5F"; C<=A AND B; Z(2 to 9)<=C;  Z(14 to 15)<= C(4 downto 3);
```



Obrázek 5 - Operace s std_logic_vector

Operátor &

Budeme-li mít dvě definice:

```
signal nv: std_logic_vector (1 downto 0);
signal n1, n0: std_logic;
```

pak můžeme n1 a n0 s nv propojit dvěma způsoby. Zaprvé po členech:

```
nv(1)<= n1;  nv(0)<= n0;
```

Zadruhé můžeme výše uvedený řádek zapsat stručně pomocí operátoru & pro spojení do vektoru.

```
nv <= n1 & n0;
```

Pozn. Operátor & představuje analogii řetězcového (concatenation) operátoru **.** *webového jazyka PHP.*

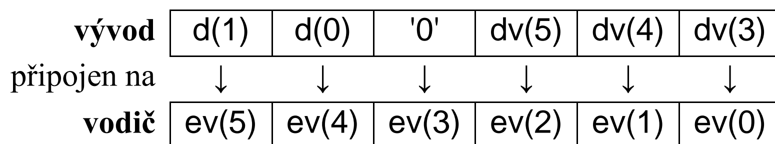
Pomocí operátoru & se dají vytvářet i složitá spojení. Máme-li někde definice:

```
signal dv, ev: std_logic_vector (5 downto 0);
```

pak následující zápis

```
ev <= dv(1 downto 0) & '0' & dv(5 downto 3);
```

vytvoří propojky



vývod	d(1)	d(0)	'0'	dv(5)	dv(4)	dv(3)
připojen na	↓	↓	↓	↓	↓	↓
vodič	ev(5)	ev(4)	ev(3)	ev(2)	ev(1)	ev(0)

Ve výrazu lze ohraničit 0 i dvojitými uvozovkami

```
ev <= dv(1 downto 0) & "0" & dv(5 downto 3);
```

Operátor & má totiž definovanou řadu přetížených operací a umí i std_logic spojit s std_logic_vector typem, a tak lze napsat jak '0' tak "0".

3 @Příklad IV. - Dekodér příkazem with...select

Zadání: Navrhněte dekodér pro lineární ukazatel hodnoty (angl. bar indicator).

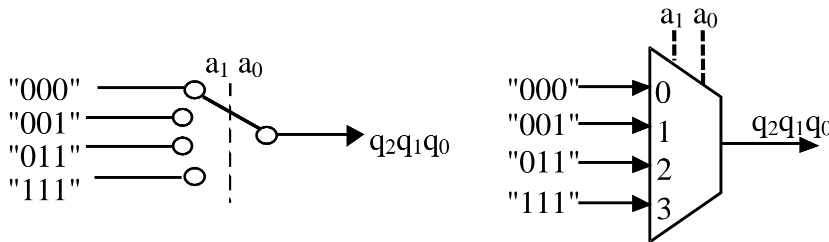
V jeho tříbitovém výstupu bude tolik bitů v logické ´1´, kolik má adresa hodnotu jako číslo bez znaménka, jak ukazuje pravdivostní tabulka vpravo.

Vstup a0 je nižším bitem a a1 vyšším bitem.

Pravdivostní tabulka dekodéru					
Vstup adresy			**Výstup**		
Hodnota	a_1	a_0	Q_2	Q_1	Q_0
0	0	0	0	0	0
1	0	1	0	0	**1**
2	1	0	0	**1**	**1**
3	1	1	**1**	**1**	**1**

Řešení: Můžeme snadno napsat tři logické rovnice, pro každý výstup jednu. Podobný postup není ani univerzální a ani přehledný. Nehodil se pro delší ukazatele hodnoty, např. u dekodéru se čtyřbitovou adresou by potřeboval patnáct rovnic pro výstupy. Najdete tedy jiné, mnohem přehlednější řešení.

Zde se přímo nabízí přepínač. Poloha jeho jezdce je určena adresou a na kontakty jsou připojené požadované výstupní kombinace hodnot bitů. K přepínači existuje analogie v podobě multiplexoru propouštějícímu na výstup jen hodnotu ze vstupu vybraného adresou.



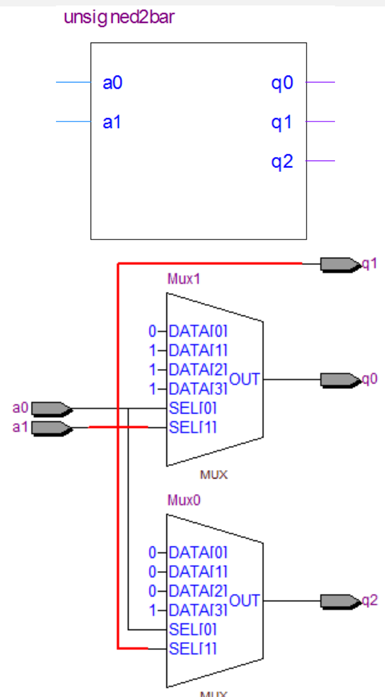
Poznámka: V obvodech lze multiplexor efektivně vytvořit přepínači na bázi CMOS tranzistorů.

Ve VHDL se multiplexor definuje příkazovou konstrukcí with...select.

VHDL kód unsigned2bar .vhd

```
--Bar indicator / cz  linearni ukazatel
library ieee;  use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity unsigned2bar is
        port (   a0, a1 : in std_logic;
                 q0, q1, q2 : out std_logic );
end;
architecture dataflow of unsigned2bar is
signal av: std_logic_vector(1 downto 0);
signal qv: std_logic_vector(2 downto 0);
begin
        av <= a1 & a0;
        with av select
        qv <=   "000" when "00",
                "001" when "01",
                "011" when "10",
                "111" when "11";
        q2<=qv(2); q1<=qv(1); q0 <= qv(0);
end;
```

Značka a schéma obvodu

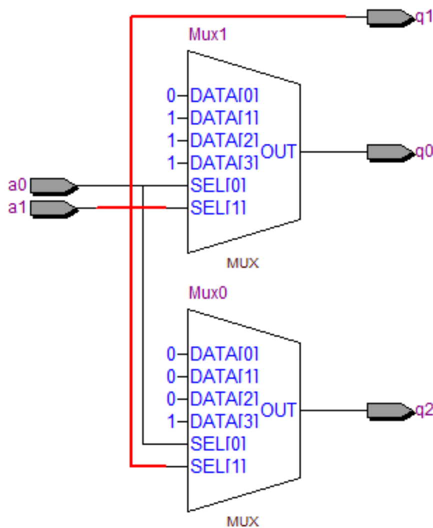
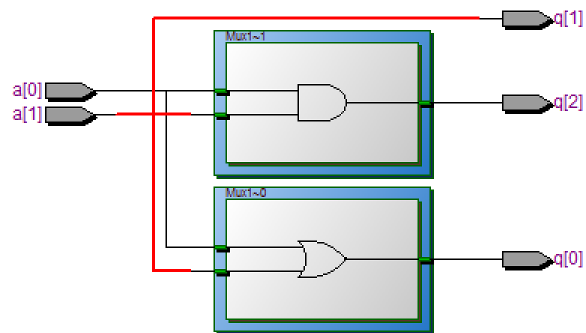


Z interního sestaveného po úvodním průchodu vidíme, že Quartus detekoval, že výstup q1 bude v jedné vždy, když vstup a1 má hodnotu '1', a tak ho vyvedl přímo. Zbylé dva vstupy zapojil multiplexory.

Obrázek ukazuje pouze výsledek úvodním průchodu VHDL, kdy se z kódu sestavuje vhodné schéma (RTL Map). Poté se provádí jeho minimalizace podle možností cílového FPGA obvodu (Technology Map). Výsledkem budou logické rovnice s AND a OR, viz Tabulka 4.

```
q0<=  a1 AND a0;  q1<= a1;  q2<=a1 OR a0;                    --Eq3-1
```

Rovnice Eq3-1 jsme si mohli po menší úvaze navrhnout i sami z logické tabulky. Nicméně jsme si napsali multiplexor coby demonstraci, že ve VHDL nezáleží tolik na co nejúspornějším kódu ve smyslu minima příkazů, ale na jeho správné struktuře. Je-li dobře zvolená a jasně specifikuje strukturu obvodu, pak Quartus si potřebné minimalizace už provede sám.

Úvodní průchod - RTL Map	Minimalizace - Technology Map
	

Pozn. Pro účely minimalizace si Quartus vytvořil skupiny z typů std_logic.

Tabulka 4 - Lineární ukazatel - prvotní schéma a jeho minimalizace

<u>**Otázka:**</u> **Proč jsme definovali av a nenapsali přímo with a & b select?** No, nenapsali jsme, protože jsme nechtěli provést příliš dlouhý žabí skok, abychom si při něm nepolámali VHDL studijní náladu. Pokud se totiž vynecháme definice av a napíšeme multiplexor přímo jako:

```
with a1 & a0 select
qv <=  "000" when "00",
       "001" when "01",
       "011" when "10",
       "111" when "11";
```

pak překladač ohlásí chybu: *Error (10327): VHDL error: can't determine definition of operator ""&"" -- found 2 possible definitions.*

Operátor & je přetížený a používá se pro více typů. V původním příkazu av <= a1 & a0; se dal správný typ výsledku odvodit z levé strany příkazu, na níž stál signál typu std_logic_vector . V příkazu multiplexoru se adresy specifikují konstantami nemajícími jednoznačný typ.

Musíme tedy vybrat návratový typ ručně pomocí typové nápovědy, kterou napíšeme uzavřením výrazu do závorek, před něž napíšeme jméno požadované typu a apostrofem ', tedy jako:

```
with std_logic_vector'(a1 & a0) select
qv <=  "000" when "00",
       "001" when "01",
       "011" when "10",
       "111" when "11";
```

Pozor, **zde nejde o přetypování**! Specifikujeme pouze přání, aby se v případě mnoha možných typů vybral právě tento. Bude-li to možné, překladač nám vyhoví.

Vlastnosti with select

Hodnoty adres za when musí být konstanty a **nutno uvést všechny možné adresy** — přepínač (multiplexor) musí pokaždé přepnout na nějaký vstup. Jinak to neumí. Můžeme také využít klíčové slovo **others**, které má význam všechny dosud neuvedené hodnoty. Příkaz by šel napsat i takto:

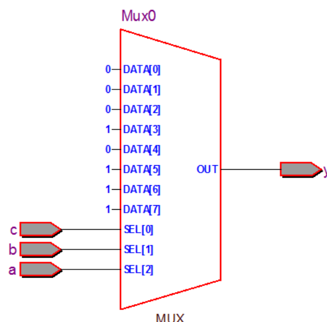
```
with std_logic_vector'(a1 & a0) select
        qv <= "000" when "00",  "001" when "01",  "011" when "10",
                "111" when others;
```

Dále lze pomocí znaku | **sdružit několik podmínek.** Kdybychom chtěli předchozí majoritu ze str. 10 definovat multiplexorem, můžeme to učinit takto:

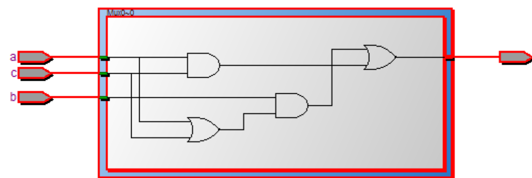
```
library ieee;     use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity    majoritaMux is
          port (   a, b, c : in std_logic;   y : out std_logic );
end;
architecture dataflow of majoritaMux is
begin
   with std_logic_vector'(a & b & c) select
          y <=    '1' when "011" | "101" | "110" | "111",
                  '0' when others;
end;
```

Pokud se podíváme na výsledek překladu majoritaMux , pak její úvodní krok obsahuje multiplexor, který se minimalizuje na logickou rovnici Eq2.4-1, kterou jsme předtím použili pro úvodní majoritu.

Úvodní průchod - RTL Map



Minimalizace - Technology Map



Rovnice: (a AND c) OR((a OR c) AND b)
= (a AND c) OR (a AND b) OR (c AND b)

Obrázek 6 - Překlad MajoritaMux

Pokročilá poznámka: Norma VHDL povoluje použít i rozsah, pokud typ adresy má definované uspořádání, což svádí k tomu, abychom v příkazu nahoře první when *řádek:*

```
when "011" | "101" | "110" | "111",
```

napsali zhuštěně jako when "011" | "101" to "111",

Quartus ho i správně přeloží, avšak u std_logic_vector *není uspořádání definované, takže jde o nestandardní konstrukci, která nemusí fungovat v jiných překladačích.*

Přenositelný kód by mohl vypadat takto:

```
with unsigned(std_logic_vector'(a & b & c)) select
 y <=     '1' when "011" | "101" to "111",
          '0' when others;
```

Tady jsme napřed specifikovali, že chceme skupinu std_logic_vector'(a & b & c) *chápat jako číslo bez znaménka, které má již definované uspořádání, a tak můžeme nahradit tři za sebou jdoucí* unsigned *binární hodnoty* "101" | "110" | "111" *rozsahem* "101" to "111".

Typ unsigned *je definovaný v* ieee.numeric_std *a jde opět o skupinu vodičů, avšak zavedly pro něj aritmetické operace. Věnujeme se mu v odstavci na str. 27.*

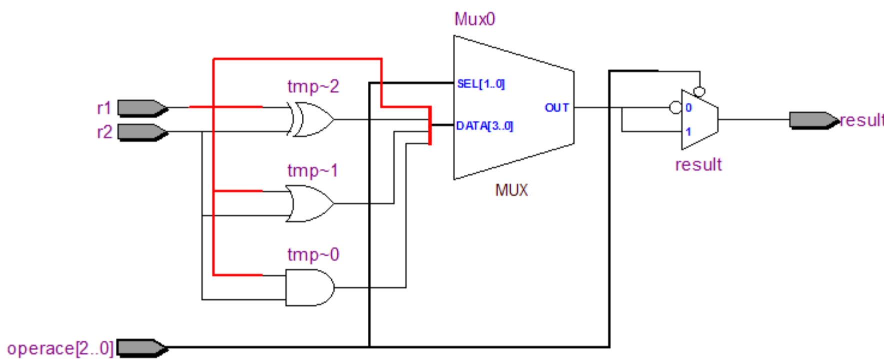
3.1.b @Příklad V. - Logická ALU

Před klíčovým slovem when **může být i logický výraz**, protože se jedná o vstup multiplexoru. Demonstrujeme si to na logické ALU, která dokáže provádět základní logické operace se vstupy r1 a r2.

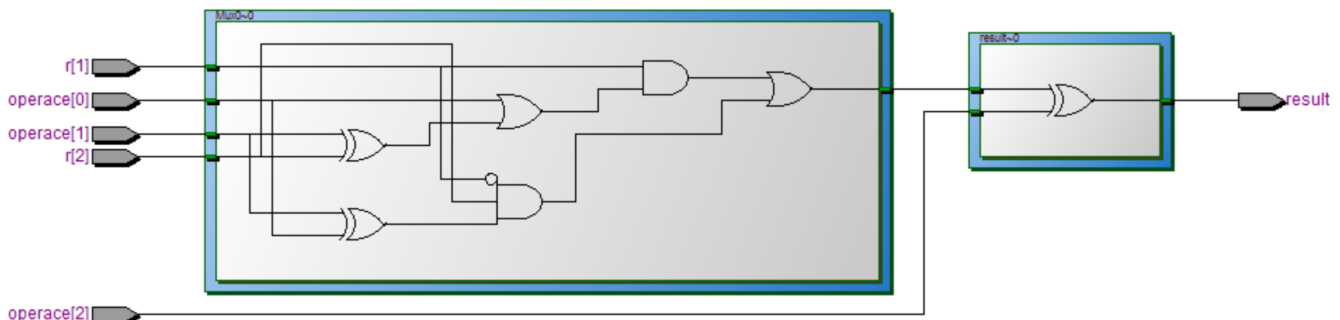
```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity LogALU is
port (  operace : in std_logic_vector(2 downto 0);  -- select required operation
        r1, r2 : in std_logic;                      -- inputs into ALU
        result : out std_logic);                    -- result of ALU
end;
architecture dataflow of LogALU is
signal tmp : std_logic;
begin
  with  operace(1 downto 0) select
        tmp <= r1 AND r2 when "00",
               r1 OR r2 when "01",
               r1 XOR r2 when "10",
               r1 when others;
  with  operace(2) select   result <= tmp when '0', NOT tmp when '1';
end;
```

Zvolili jsme jednobitové vstupní r1 a r2 typu std_logic, aby nám vyšlo jednoduché schéma. Můžete sice r1 a r2 změnit na std_logic_vector libovolné délky, ale pak by RTL Map a Technology Map obsahovala stejné elementy, jako vidíte dole, jen ve větším počtu — každý bit vstupu by měl svůj vlastní multiplexor.

Úvodní průchod - RTL Map



Minimalizace - Technology Map



Quartus konvertoval první multiplexor na 4-vstupový logický obvod, ale druhý multiplexor, který volí negaci výsledku, nahradil pouhým XOR hradlem.

Vyzkoušejte si, že příkaz:

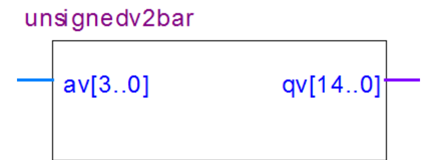
```
with operace(2) select result <= tmp when '0', NOT tmp when '1';
```

je opravdu plně ekvivalentní s jednodušším zápisem:

```
result <=  operace(2) XOR tmp;
```

3.2 @Příklad VI. - Dekodér pro velký ukazatel

Předchozí lineární ukazatel lze rozšířit na vícebitový výstup. Opět využijeme naši šablonu:




```
--Long bar indicator / cz velky linearni ukazatel
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity unsignedv2bar is
port    ( av : in std_logic_vector(3 downto 0) ;
          qv : out std_logic_vector(14 downto 0));
end;

architecture dataflow of unsignedv2bar is
signal tmp : std_logic_vector(15 downto 0);
begin
   with  av select
      tmp <= X"0000" when X"0",   X"0001" when X"1",   X"0003" when X"2",   X"0007" when X"3",
             X"000F" when X"4",   X"001F" when X"5",   X"003F" when X"6",   X"007F" when X"7",
             X"00FF" when X"8",   X"01FF" when X"9",   X"03FF" when X"A",   X"07FF" when X"B",
             X"0FFF" when X"C",   X"1FFF" when X"D",   X"3FFF" when X"E",   X"7FFF" when X"F";
   qv <= tmp(14 downto 0);
end;
```

Kvůli hexadecimálnímu zápisu X".." jsme si definovali pomocné pole, protože ten lze uplatnit jen u délek dělitelných 4, což qv nesplňuje. Napřed jsme hexadecimální hodnoty zapojily na tmp a z něho jsme si potřebné signály vyvedli na výstup. Nijak jsme tím nezvýšili složitost, protože signál tmp(15), který se nepoužívá ve výsledném qv, se odstranění během minimalizace RTL Map na Technology Map.

I tak náš kód má daleko do elegance. Podíváme-li se za klíčová slova when, kde píšeme volby, vidíme, že jsme vkládali čísla 0 až 15 psaná hexadecimálně. Vždyť jde vlastně o výběr z pole pomocí indexu! Kód lze přepsat mnohem elegantněji, avšak napřed se musíme podívat na tvorbu polí.

3.3 Tvorba vlastních typů a atributy VHDL

Už standardní knihovna ieee.std_logic_1164 definuje std_logic_vector jako pole:

```
type std_logic_vector is array (natural range<>) of std_logic;
```

kde

- type je klíčové slovo a
- std_logic_vector udává název námi vytvářeného typu, zde je jím std_logic_vector;
- natural znamená číslo typu integer omezené na rozsah 0 až maximální číslo integer použité implementace překladače (v Quartusu do 2^{31}-1).
- range je klíčové slovo intervalu. Ve spojení s natural leží někde mezi 0 a maximálním číslem.
- range <> znamená nespecifikovaný rozsah, jeho hodnota bude doplněna při použití typu.

Definujeme-li si dva vlastní typy se shodnou strukturou, budou z hlediska překladače různé. Například:

```
type std_logic_vector_A is array (7 downto 0) of std_logic;
type std_logic_vector_B is array (7 downto 0) of std_logic;
signal slvA1, slvA2 : std_logic_vector_A;
signal slvB1, slvB2  : std_logic_vector_B;
```

Připojíme-li nyní k signálům konstantní hodnoty příkazy <=, pak vše proběhne v pořádku; pracujeme s členy vektorů a ty jsou shodného datového typu std_logic.

```
slvA1 <= X"12"; slvB1 <= X"34";  -- OK, lze
```

Pokud se pokusíme vzájemně přiřadit vektory obou skupin, nepůjde to:

```
slvB2<=slvA1;  slvA2<=slvB1;     -- Chyba  - typy nesouhlasi
```

U silně typových jazyků, mezi které patří i VHDL, se shoda vnitřní struktury typu netestuje, důležitá jsou výhradně jména typů, a ta se liší.

Existuje ale možnost definovat si **subtype** zužující originální typ. Přepíšeme-li jimi uvedené definice:

```
subtype std_logic_vector_A is std_logic_vector (7 downto 0);
subtype std_logic_vector_B is std_logic_vector (7 downto 0);
signal slvA1, slvA2 : std_logic_vector_A;
signal slvB1, slvB2  : std_logic_vector_B;
```

pak se předchozí propojovací příkazy přeloží bez chyby, protože subtypy se pokládají za plně kompatibilní s výchozím datovým typem a všemi jeho subtypy:

```
slvA1 <= X"12";  slvB1 <= X"34";  slvB2<=slvA1;  slvA2<=slvB1;
```

Souhrn: Definicemi **type** vytváříme izolované datové typy, zatímco **subtype** umožňuje, aby se nový datový typ zařadil do členské skupiny již existujícího typu.

VHDL atributy dovolují dotazovat se na parametry definic. Jde o vzdálenou analogii "property" známé z tříd v Java či v C#, jen se neoddělují tečkou, ale apostrofem. Přidáme si ještě definice:

```
subtype std_logic_vector_C is std_logic_vector (5 to 27);
signal slvC : std_logic_vector_C;   --tedy signal slvC : std_logic_vector (5 to 27);
```

pak typy založené na array lze použít následující atributy s významem:

Atribut použitý u	slvA1, slvB1	slvC	
LEFT	7	5	levá hodnota rozsahu
RIGHT	0	27	pravá hodnota rozsahu
LOW	0	5	nižší číselná hodnota rozsahu
HIGH	7	27	vyšší číselná hodnota rozsahu
LENGTH	8	23	počet prvků
ASCENDING	*FALSE*	*TRUE*	*Boolean TRUE při definici s **to***
RANGE	7 **downto** 0	5 **to** 27	definiční rozsah
REVERSE_RANGE	0 **to** 7	27 **downto** 5	obrácený definiční rozsah

Tabulka 5 - Atributy typů založených na array

Atributy rozsahu vrací VHDL datový typ range, který se hodí pro definice signálů stejného rozsahu. Lze se jím odvolávat na předchozí definici, což umožní rychlejší změny, je-li nutné původní rozsah upravit.

```
signal X : std_logic_vector(31 downto 0);
signal novy1 : std_logic_vector(X'RANGE);          -- (31 downto 0)
signal novy2 : std_logic_vector(X'REVERSE_RANGE);        -- (0 to 31)
```

Lze rovněž definovat pole polí, takže můžeme pro náš velký ukazatel definovat vlastní typ, opět o délce 16, aby se daly využít X" " hexadecimální konstanty.

Pole se ve VHDL musí vždy napřed definovat jako typ a až od něho lze vytvářet instance.

```
type BarArray_t is array(0 to 15) of std_logic_vector(15 downto 0);
```

Teprve nyní lze vytvořit inicializované pole:

```
constant barArray : BarArray_t := (  X"0000", X"0001", X"0003", X"0007", X"000F",X"001F",X"003F", X"007F",
                                     X"00FF", X"01FF",X"03FF", X"07FF", X"0FFF", X"1FFF", X"3FFF", X"7FFF");
```

Definice barArray můžeme založit i na signal:

```
signal barArray : BarArray_t := (    X"0000", X"0001", X"0003", X"0007", X"000F",X"001F",X"003F", X"007F",
                                     X"00FF", X"01FF", X"03FF", X"07FF", X"0FFF", X"1FFF", X"3FFF", X"7FFF");
```

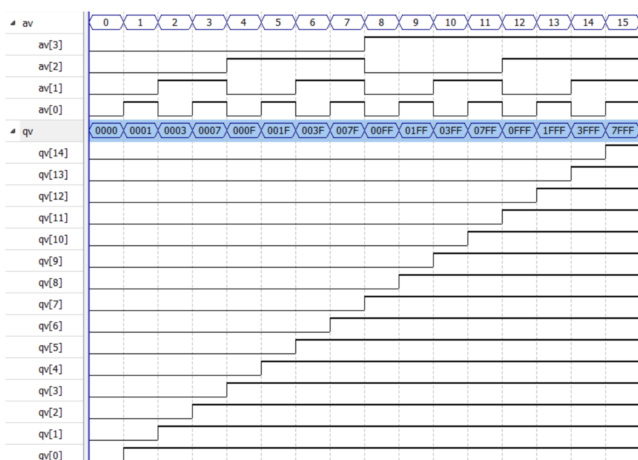
Když ale použijeme zápis s constant, upřesníme, že na hodnoty se nesmí připojit výstup.

3.4 @Příklad VII. - Dekodér pro velký ukazatel s polem

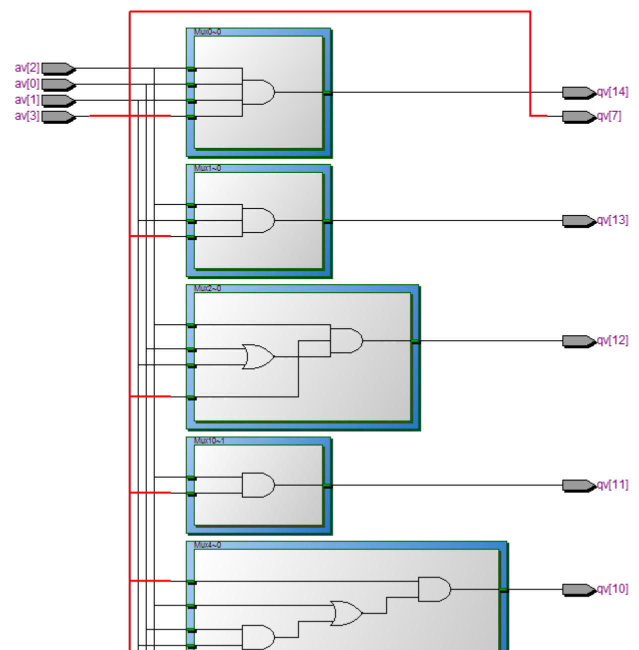
Využijeme-li atributů a možností definovat pole, pak se kód velkého ukazatele zpřehlední.

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity unsignedv2bar1 is
   port (  av : in std_logic_vector(3 downto 0) ;
           qv : out std_logic_vector(14 downto 0));
end;

architecture dataflow of unsignedv2bar1 is
type BarArray_t is array(0 to 15) of std_logic_vector(15 downto 0);
constant barArray : BarArray_t := (X"0000", X"0001", X"0003", X"0007", X"000F", X"001F",X"003F", X"007F",
                                   X"00FF", X"01FF",X"03FF", X"07FF", X"0FFF", X"1FFF", X"3FFF", X"7FFF");
signal tmp : std_logic_vector(barArray(0)'RANGE);

begin
  tmp <= barArray( to_integer(unsigned(av)) );
  qv <= tmp(qv'RANGE);
end;
```

Výsledek simulace



Kousek Technology Map



Indexy pole musí být čísly typu integer, a tak náš nový kód obsahuje operaci to_integer(unsigned(**av**)), která převede std_logic_vector na skalární typ integer. Převod se provádí dvoufázově. Napřed se upřesní, že chceme chápat skupinu vodičů v av jako číslo bez znaménka, a teprve se konvertuje na integer.

Poznámka: Pokud někomu nevyhovuje dvoustupňový převod std_logic_vector na integer, může si definovat vlastní konverzní funkci. VHDL nabízí široké možnosti přetěžování, které lze uplatnit i na operátory. Bude na přednáškách.

Poslední řádky **tmp** <= **barArray**(to_integer(unsigned(**av**))); **qv** <= **tmp**(qv'RANGE);
by se rovněž daly napsat zhuštěně jako

```
qv <= barArray( to_integer(unsigned(av)) )(qv'RANGE);
```

tedy bez nutnosti zavést si pomocný vodič tmp. Můžemee časem používat takové zkrácené zápisy, pokud se v nich dobře orientujeme, ale nemusíme. Zhuštěným zápisem se ušetří jen text kódu. Obvod zůstane stejný. A i ve VHDL platí zásada, že kód má zůstat přehledný pro naše oči.

Číslo integer ve VHDL

VHDL chápe integer jako bezrozměrný skalár.

Řekne-li se integer, u běžných počítačů víme podle typu prostředí, jakou má bitovou délku, ale ve VHDL tohle neplatí. Na rozdíl od procesorů můžeme číslo v obvodu uložit v libovolné délce dle momentální potřeby, a to od 1 bitu až do 32 bitů, maximum implementace překladače v Quartusu. Norma VHDL sama délku integer nijak neomezuje. Kvůli tomu se typ integer nedoporučuje používat jako vstup nebo výstup v entitě. V některých implementacích se to i přímo zakazuje. Snadno se poruší kompatibilita.

Jako porty entity lze použít signed a unsigned typy, které se mají přesně definované rozsahy ve stejném stylu jako std_logic_vector:

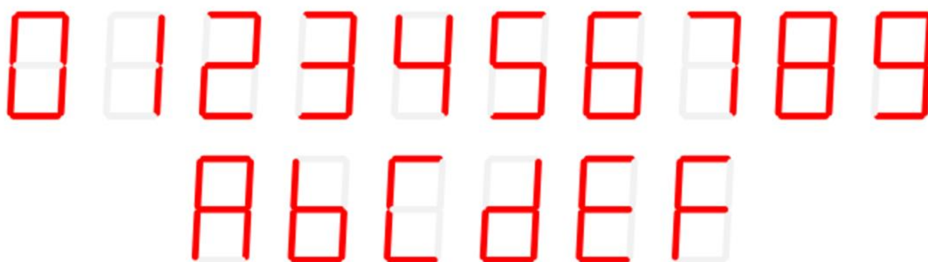
```
type std_logic_vector is array (natural range<>) of std_logic;
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

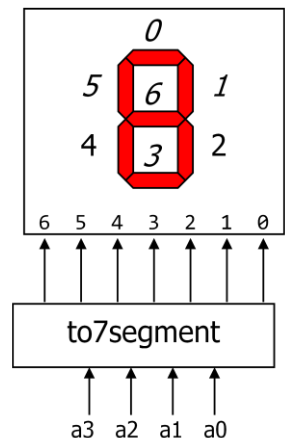
Definice se liší jen tím, že k unsigned a signed typům existují v knihovně i přetížené aritmetické operace.

Otázka: Proč se aritmetika nedoplnila i k typu std_logic_vector? Nedefinovala se úmyslně, aby se u něho musel napřed jasně definovat binární formát čísla! Špatně navržené obvody mohou způsobit značné škody, a tak při jejich popisu raději nadměrně typuje a izolují se definice, aby nedošlo k jejich záměně a nechtěné funkci. Již v úvodu jsme se zmínili, že VHDL vyšlo z jazyka ADA určeného pro návrhy vojenských systémů, kde opravdu záleží na bezchybnosti.

3.5 *** Cvičná úloha 2: Dekodér pro 7segmentový displej

Sestavte obvod se čtyřmi vstupy a3, a2, a1 a a0, který zobrazí hexadecimální číslici na 7segmentovém displej. Běžné číslování segmentů vidíte vpravo. Vstup a3 má nejvyšší váhu a a0 nejnižší.





Píšeme pro desku DE2 a na té svítí segment, pokud se na něj přivede '0'. Chceme-li třeba rozsvítit třeba číslici 1, tak na výstup hex pošleme vektor "1111001".

Kdybychom obvod vytvářeli logickými funkcemi, jejich návrh by nám zabral několik hodin. Využijeme-li předchozí kód s polem, pak dekodér pro 7segmentový displej sestavíme za několik minut.

Zkuste si obvod sami navrhnout, podívejte se na jeho RTL a Technology Map. Jedno možné řešení najdete v příloze na str. 59.

4 @Příklad VIII. - VHDL příkaz when…else

Zadání:

Pro tři vstupy žádostí o přerušení procesoru Int3 až Int1 navrhněte prioritní dekodér, který pošle číslo periférie s nejvyšší prioritou žádající právě o přerušení.

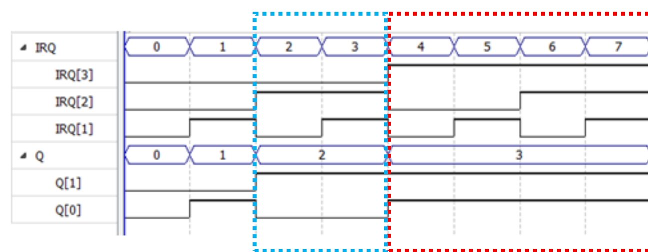
Je-li vstup Int3 s nejvyšší prioritou v '1' bude na výstupu číslo 3, ostatní vstupy se ignorují. Int1 má nejnižší prioritu a posílá číslo 1, jsou-li zbývající vstupy v ´0´. Nežádá-li nikdo se o přerušení, výstup bude číslo 0, viz zkrácená pravdivostní tabulka.

Prioritní dekodér					
Vstupy			Výstupy		
IRQ3	**IRQ2**	**IRQ1**	Q1	Q0	Číslo
1	**-**	**-**	**1**	**1**	3
0	**1**	**-**	**1**	**0**	2
0	**0**	**1**	**0**	**1**	1
0	**0**	**0**	**0**	**0**	0

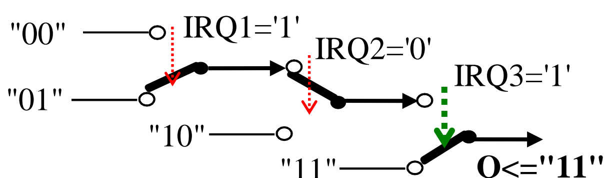
Mohli bychom samozřejmě sestavit logické rovnice nebo vytvořit multiplexor příkazem with .. select

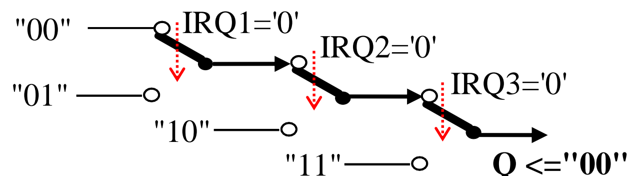
```
with unsigned(IRQ) select              --by unsigned type conversion, we define ordering
     q <=   "11" when "100" to "111",  -- we utilize ordering to short the choice list by range
            "10" when "010" | "011",
            "01" when "001",
            "00" when others;
```

Podobný postup lze rozšířit i na více žádostí o přerušení, ale budeme zdlouhavě vypisovat rozsahy vstupů. Využijeme raději toho, že při aktivaci přerušení vyšší úrovně se ignorují hodnoty nižších vstupů.



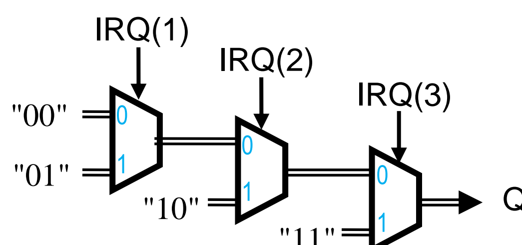
Podobnou vlastnost nabízí kaskáda dvouvstupových multiplexorů, kde jednotlivé žádosti o přerušení tvoří podmínky jejich přepnutí.





Bude-li sepnutý první přepínač IRQ3, pak na výstup Q_1 Q_0 bude přivedena kombinace "11", tj. číslo 3, bez ohledu na pozice ostatních přepínačů. Nebude-li sepnutý IRQ3, uplatní se vliv dalších přepínačů dle pořadí (priority) podmínek.

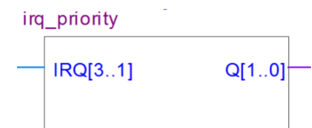
Sdružíme žádosti o přerušení do vektoru IRQ, a to v pořadí jejich požadované priority, aby se s nimi lépe manipulovalo. Přepínače nakreslíme schematickou značkou multiplexoru:



Podobné kaskádě odpovídá ve VHDL konstrukce `when …else:`

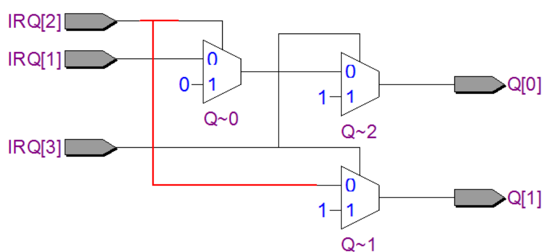
VHDL kód irq_priority.vhd	Značka obvodu

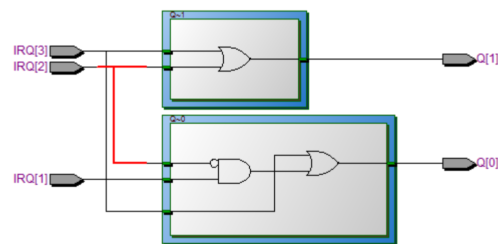
```
--Interrupt request priority
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity irq_priority is
   port (  IRQ : in std_logic_vector(3 downto 1);
           Q :    out std_logic_vector(1 downto 0) );
end;

architecture dataflow of irq_priority is
begin
   Q<= "11" when IRQ(3)='1' else
       "10" when IRQ(2)='1' else
       "01" when IRQ(1)='1' else
       "00";
end;
```



Podmínka za when **vyžaduje typ** boolean a kvůli tomu jsme nemohli vložit pouze logickou hodnotu, např. užití *when IRQ(3) else* by ohlásilo chybu. Potřebujeme výsledek boolean, a tak se napsalo IRQ(3)='1'.

Úvodní RTL Map	Technology Map





V úvodu se Quartus sestavil kaskádu multiplexorů pro každý bit zvlášť a zmenšil její úroveň. Poté zapojení minimalizoval. A se zmenšením má pravdu:

```
Q(0)<=  '1' when IRQ(3)='1' else     ←zkráceno     Q(0) <= '1' when IRQ(3)='1' else
        '0' when IRQ(2)='1' else       v RTL               '0' when IRQ(2)='1' else
        IRQ(1);                                            '1' when IRQ(1)='1' else
                                                           '0';
Q(1)<=  '1' when IRQ(3)='1' else     ← zkráceno    Q(1) <= '1' when IRQ(3)='1' else
        IRQ(2);                        v RTL               '1' when IRQ(2)='1' else
                                                           '0' when IRQ(1)='1' else
                                                           '0';
```

Příkaz when else dovoluje použít výrazy jak v podmínkách, tak na vstupech, zatímco dříve probraný příkaz with select dovolí výrazy jen na svých vstupech.

Všimněte si, že v příkazu when .. else **neopakujeme nesplnění předchozích podmínek**. Bylo by zbytečné psát něco ve stylu:

```
Q<=  "11" when IRQ(3)='1' else
     "10" when IRQ(3)='0' and IRQ(2)='1' else
     "01" when IRQ(3)='0' and IRQ(2)='0' and IRQ(1)='1' else
     "00";
```

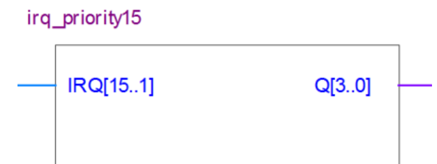
Zápis není sice chybný, ale zvýrazněné části implicitně vyplývají z předchozích podmínek. Hodnoty "10", "01","00" se mohou připojit na výstup jedině tehdy, když INT(3)='0'. Nadbytečné testy by akorát kód znepřehlednily. "*Kvůli čemu je tedy vkládat? — leda jako mňamky pro klávesu delete!*"☺

A jak by vypadal kód, kdybychom potřebovali víc vstupů přerušení? Stačí nám jen přidat víc řádků pomocí oblíbené metody "*copy-paste-edit*" .

4.1.a @Příklad IX. - Prioritní dekodér s 15-vstupy

VHDL kód irq_priority15.vhd	Značka obvodu

```
--Interrupt request priority
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity irq_priority15 is
        port (   IRQ : in std_logic_vector(15 downto 1);
                 Q : out std_logic_vector(3 downto 0) );
end;
architecture dataflow of irq_priority15 is
begin
   Q<=  X"F" when IRQ(15)='1' else
        X"E" when IRQ(14)='1' else
        X"D" when IRQ(13)='1' else
        X"C" when IRQ(12)='1' else
        X"B" when IRQ(11)='1' else
        X"A" when IRQ(10)='1' else
        X"9" when IRQ(9)='1' else
        X"8" when IRQ(8)='1' else
        X"7" when IRQ(7)='1' else
        X"6" when IRQ(6)='1' else
        X"5" when IRQ(5)='1' else
        X"4" when IRQ(4)='1' else
        X"3" when IRQ(3)='1' else
        X"2" when IRQ(2)='1' else
        X"1" when IRQ(1)='1' else X"0";
end;
```



Tabulka 6 - Prioritní dekodér

Zkusíme-li ho jeho test v simulátoru, dostaneme opravdu dlouhý grafický výstup. Složitější obvody se v HDL jazycích testují už pomocí automatických simulací, tzv. testbench, které budou na přednáškách.

4.2 Podmíněné přiřazení

V jazyce C existuje operátor **? :** podmíněného přiřazení, jehož obvodovou analogii představuje dvou-vstupový multiplexor, který můžeme vytvořit buď příkazem with...select, nebo when..else.

Příkaz with...select vytváří vždy jediný multiplexor a u podmíněného přiřazení bude mít jen dva vstupy. Kaskáda multiplexorů when..else bude zase pro něj obsahovat pouze jediný dvouvstupový multiplexor, čili obě konstrukce vedou v případě podmíněného přiřazení na totožný výsledek.

Máme-li třeba definované signály:

```
signal A, B, C: std_logic_vector (7 downto 0);
signal sel, x, y, z: std_logic;
```

pak lze podmíněné přiřazení napsat dvěma způsoby:

when ... else		with ... select
A<= **B** when sel='1' else **C**;	←*ekvivalent*→	with sel select **A**<= **B** when '1', **C** when '0';
x<=**y** when sel='1' else **z**;	←*ekvivalent*→	with sel select **x**<= **y** when '1', **z** when '0';

V praxi se častěji dává přednost kratšímu zápisu when ... else. Nezapomínejte však, že u něho se za when píše výraz vracející typ boolean.

Ve with select příkazu se naopak za when dává konstanta s hodnotou známou během překladu a typem shodným se signálem použitým za with.

4.3 *** Cvičná úloha 3: 8-vstupový prioritní inhibitor

Zadání: Navrhněte obvod s 8 vstupy av(7 downto 0) a 8 výstupy qv(7 downto 0) propouštějící pouze nejvyšší vstup ve stavu ´1´ na jemu číselně odpovídající výstup, což znamená, že nejvýše jen jeden výstup bude tedy v ´1´ — kvůli tomu jsme obvod nazvali prioritní inhibitor.

Pokud bychom měli jen tři vstupy av(2 downto 0) a 3 výstupy qv(2 downto 0), pak by logická tabulka vypadala takto:

Úplná pravdivostní tabulka					
Vstupy			Výstupy		
av(2)	**av(1)**	**av(0)**	**qv(2)**	**qv(1)**	**qv(0)**
0	0	0	0	0	0
0	0	1	0	0	1
0	1	*0*	0	1	0
0	1	*1*	0	1	0
1	*0*	*0*	1	0	0
1	*0*	*1*	1	0	0
1	*1*	*0*	1	0	0
1	*1*	*1*	1	0	0

Zkrácená pravdivostní tabulka					
Vstupy			Výstupy		
av(2)	**av(1)**	**av(0)**	**qv(2)**	**qv(1)**	**qv(0)**
0	0	0	0	0	0
0	0	1	0	0	1
0	1	-	0	1	0
1	-	-	1	0	0

Tabulka 7 - Pravdivostní tabulka prioritního inhibitoru se 3 vstupy a výstupy

Kde bychom takový obvod použili? Představte si, že máte 8 přepínačů ovládajících nějaké funkce, přičemž se žádá, aby se najednou zapnul jen jeden přepínač. Uživatel může nechtěně, či úmyslně ("pokušitel jeden"), provést několik voleb najednou. Pokud se využije náš prioritní inhibitor, duplicitní zapnutí se potlačí podle priorit přepínačů, takže máme jistotu volby maximálně jedné funkce.

Sestavte si VHDL kód sami.

Opět se podívejte na RTL Map a Technology map. Své řešení si vyzkoušejte i simulací.

Nápověda: Kód lze rychle sestavit edicí předchozí úlohy a Tabulka 7 napovídá jak příkaz napsat.

Možné řešení najdete v příloze na str. 61.

5 @Příklad X. - VHDL příkazy generic a for-generate

V kapitole 3 na str. 20 jsme navrhli lineární ukazatel, jehož logickou tabulku vidíme vpravo, který jsme si později rozšířili na více bitů.

Naše řešení vycházelo z předdefinovaného pole a museli jsme provádět jeho změny pro každou požadovanou bitovou délku výstupu.

Pravdivostní tabulka dekodéru				
Vstup adresy		Výstup		
Hodnota	av	qv(2)	qv(1)	qv(2)0
0	00	0	0	0
1	01	0	0	1
2	10	0	1	1
3	11	1	1	1

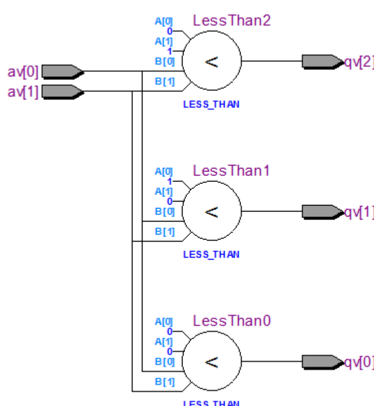
V této části si ukážeme univerzální kód, tedy lineární ukazatel libovolné délky, v němž již nebude potřebné předdefinované pole.

Z logické tabulky vyplývá, že kód lineární ukazatele lze napsat i pomocí podmíněného přiřazení. Využijeme toho, že k typu unsigned existují běžné aritmetické operace včetně operátorů porovnávání, které mají stejný zápis jako v jazyce C, až na operátory *rovná se* a *nerovná se*. Ty se ve VHDL píší trochu jinak, jako = (nikoli C operátor ==) a /= (nikoli C operátor !=).

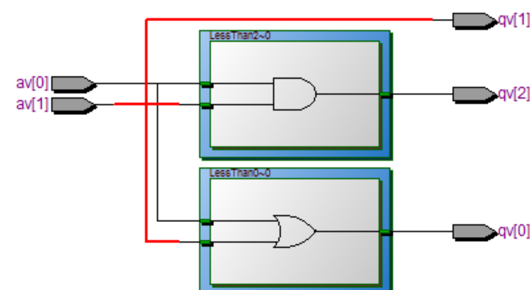
```
--Bar indicator / cz linearni ukazatel
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity unsignedx2bar is
    port (  av : in std_logic_vector(1 downto 0) ;
            qv : out std_logic_vector(2 downto 0));
end;
architecture dataflow of unsignedx2bar is
signal x : unsigned(av'RANGE);
begin
    x <= unsigned(av);
    qv(0) <= '1' when x>0 else '0';
    qv(1) <= '1' when x>1 else '0';
    qv(2) <= '1' when x>2 else '0';
end;
```

Překladač si v úvodním kroku vytvoří odlišné RTL Map meta-schéma, v němž využije komparátory, ale následnou minimalizací vznikne logický obvod totožný s výsledkem návrhu with select příkazem, viz Tabulka 4, na str. 21.

Úvodní RTL Map



Technology Map



Podmínky pro jednotlivé bity by se daly lehce namnožit kopírováním s následnou edicí, neboť se vzájemně liší pouze číslem v indexu a porovnávání. Podobný postup není příliš univerzální.

VHDL dovoluje zapsat i analogie for cyklu, avšak v něm nemůže opakovaně vykonávat stejný příkaz s různými hodnotami, protože v obvodu tohle nejde. Cyklus for se překládá stylem, který je z programování znám jako inline expansion, kdy se nahradí několikanásobným vložením stejného příkazu s různými parametry. V našem příkladě bude mít následující tvar a nahradí se příkazy vpravo:

Zápis cyklu		Překladač vygeneruje příkazy
rep: for i in 0 to 2 generate qv(i) <= '1' when x>i else '0'; end generate;	→	qv(0) <= '1' when x>0 else '0'; qv(1) <= '1' when x>1 else '0'; qv(2) <= '1' when x>2 else '0';

Generační cyklus má dvojí syntaxi, buď pro jednu podmíněnou generaci:

```
label: if <expression> generate  <concurrent-statements>
       end generate [label];
```

nebo pro opakovanou generaci:

```
label: for <parameter> in  <range>  generate   <concurrent-statements>
       end generate [label] ;
```

kde

- pokud se použije první tvar, pak **<expression>** dávající typ boolean musí být globálně statické, tedy s konstantní hodnotou známou už v době kompilace.
- **label** je povinné návěští oddělené dvojtečkou. Musíme ho zadat, i když ho nikde nepoužijeme. Nahoře jsme zvolili název "rep", ale mohli jsme zadat i jiný, třeba "cyklus1", apod.
- **<parameter>** představuje unikátní parametr cyklu, který bude nabývat hodnot v rozsahu <range>. Nesmí jít o již definovaný signál. Cyklus ho vytvoří uvnitř sebe, a mimo něj není platný. Hodnotu parametru lze jen číst, nelze ji přepisovat.
- **<range>** udává rozsah hodnot; ten má stejný tvar jako rozsahy polí, je buď s downto nebo to
- end generate [label] ; — cyklus se musí zakončit buď end generate nebo end generate label; Klíčové slovo generate je za end povinné, zde ho nelze vynechat;
- **<concurrent-statements>** označuje blok jednoho či více příkazů řazených mezi "concurrent". Sem patří přiřazení **<=** a konstrukce with..select , when..else, dále také for-generate (cykly lze vnořit) a lze vložit i příkaz port map; ten probereme v kapitole 6.
- **Příkazy musí být vždy úplné**. Ač by se nám to mnohdy hodilo, žel nelze generovat jen dílčí části, jako třeba jednotlivé řádky podmínek ve "when..else" či přidávat členy do výrazů.
- Příkazy generate můžete použít jedině v architektuře, v entitě ne. Mohou mít i své vlastní lokální definice. V tom případě se za generate vloží ještě klíčové slovo begin. Lokální definice se pak píší před begin (stejné uspořádání jako architecture - begin - end;). Zájemci najdou podrobnější informace ve VHDL referenčních příručkách. Měli by se však napřed podívat na kapitolu 6.6 na straně 48.
- Nezaměňujte souběžný příkaz for generate dataflow stylu se sekvenčním příkazem for loop , který patří už do behavioral stylu. Překladač transformuje for loop během kompilace na for generate.

5.1 @Příklad XI. - Deklarace generic

Změnou rozsahu ve for-generate dokážeme již vytvořit libovolně dlouhé pole, avšak pořád nevzniká univerzální obvod. Potřebujeme ještě parametrizovat náš návrh, tedy zadat v něm jistou analogii parametrů v konstruktorech tříd.

Slouží k tomu příkaz generic, který se vkládá do entity a definuje prvky read-only, které se však dají měnit při vytváření instancí obvodů příkazem map, který bude dále.

```
--Configurable bar indicator / cz Univerzalni dekoder pro linearni ukazatel
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity uint2bar0 is
    generic (  AV_LENGTH : integer := 4;  -- bit length of av input
               QV_LENGTH : integer := 16); -- bit length of qv output
    port (  av : in std_logic_vector(AV_LENGTH-1 downto 0) ;
            qv : out std_logic_vector(QV_LENGTH-1 downto 0));
end;

architecture dataflow of uint2bar0 is
signal x : unsigned(av'RANGE);
begin
    x <= unsigned(av);
  rep: for i in 0 to QV_LENGTH-1 generate
       qv(i) <= '1' when x>i else '0';
   end generate;
end;
```

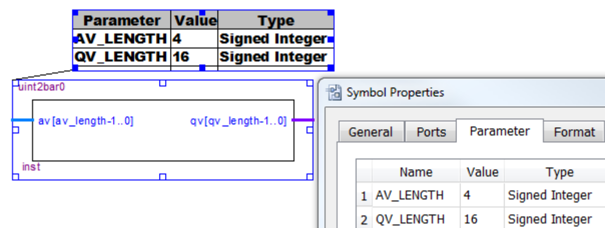
Definice generic , která se nově objevila v sekci entity, může obsahovat několik různých parametrů. Většina implementací VHDL překladačů povoluje u parametrů pouze typy integer:

```
generic(   name1: type1 := default_value1;   name2: type2 := default_value2;
           . . .
           nameN: typeN := default_valueN  );
```

Všimněte si, že stejně jako u bloku port se ani v generic nepíše ; za poslední prvkem seznamu.

Zvnějšku zadané parametry se uvnitř VHDL kódu chovají vždy jako konstanty a musí mít výchozí hodnotu, která se přiřazuje pomocí **:=** , nikoliv <= "concurrent assignment", protože nejde o propojení; všechny parametry se v době překladu nahradí hodnotami uvedenými v jejich instancích entity.

V editoru symbolických schémat lze parametry změnit vyvoláním kontextového menu properties vybrané schematické značky, která obsahuje nyní navíc i vstup pro zadání parametrů s individuálními hodnotami u každé instance obvodu.



Obrázek 7 - Ukázka použití lineárního ukazatele

Poznámka: V editoru schémat lze zapnout/vypnout zobrazování hodnot generic parametrů u instancí v kontextovém menu editoru: pravou myší kamkoliv na volnou plochu a "Show->Show Parameter Assignments".

5.2 @Příklad XII. - Kontrola parametrů generic

Pokud někdo použije obvod uint2bar a zadá špatné parametry, nulu nebo záporná čísla, pak překladač vypíše chybu. Uživatel bude však jen vědět, že něco napsal špatně, ale bude postrádat podrobnější informaci.

Ke kontrole hodnot parametrů můžeme do sekce entity vložit příkazy tzv. pasivního procesu, který se vypočítá pouze v době kompilace a nekonvertuje se na obvod. Slouží výhradně ke kontrole konzistence parametrů.

```vhdl
--Configurable bar indicator with the checking of generic parameters
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity uint2bar is
     generic (   AV_LENGTH : integer := 4;   -- bit length of av input
                 QV_LENGTH : integer := 16); -- bit length of qv output
     port (  av : in std_logic_vector(AV_LENGTH-1 downto 0) ;
             qv : out std_logic_vector(QV_LENGTH-1 downto 0));
begin -- passive process - its statements are executed in compile time only
   assert AV_LENGTH >0 AND QV_LENGTH >0
   report "uint2bar requires AV_LENGTH >0 and QV_LENGTH >0 " severity failure;
   assert  2**AV_LENGTH-1 <= QV_LENGTH
   report "uint2bar displays only range 0 to " & integer'Image(QV_LENGTH-1) & " of av input"
   severity warning;
end;
architecture dataflow of uint2bar is
signal x : unsigned(av'RANGE);
begin
      assert false report "AV max value ="& integer'Image(2**av'LENGTH-1) severity note;
       x <= unsigned(av);
  rep: for i in 0 to QV_LENGTH-1 generate  qv(i) <= '1' when x>i else '0';
       end generate;
end;
```

Příkaz assert není obdobou výjimky throw programovacích jazyků, protože se nevyvolává během činnosti obvodu; to by ani nešlo. Vyhodnocuje už v době překladu a odpovídá stejnojmenným rozšířením v Java, C# a C++. Například do C programu se jeho funkční analogie přidá pomocí #include <assert.h>

Ve VHDL má příkaz assert má tři části:

```
assert <condition>  report <string> severity <severity_level>;
```

jeho sekce report a severity jsou nepovinné, lze vynechat kteroukoli z nich či obě.

- <condition> označuje výraz vracející datový typ boolean. Jeho hodnota se počítá během překladu, a tak můžeme použít i náročnější konstrukce. Operátor ** ve výrazu 2**AV_LENGTH-1 počítá vztah 2^{AV_LENGTH} -1, tedy nejvyšší hodnotu čísla unsigned dané bitové délky.
- <string> označuje řetězec vypsaný při podmínce s hodnotou false, tedy nesplněné;
- <severity_level> specifikuje požadovanou reakci překladače, ta je buď:
  - **note** — vypíše se text s příznakem pouhé informace;
  - **warning** — vypíše se text s příznakem varování;
  - **error** — udává chybu bránící v sestavení obvodu, nicméně překlad může pokračovat;
  - **failure** — znamená inkonsistenci či nesmyslné zadání a nelze pokračovat v překladu.

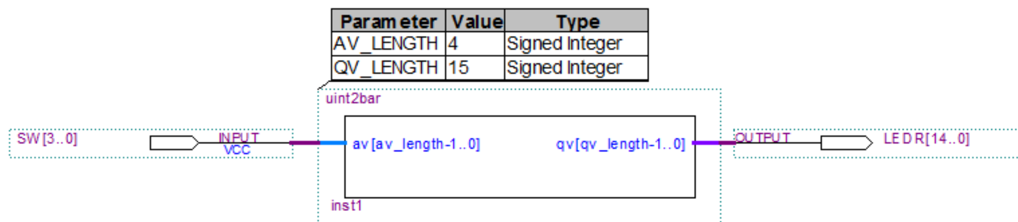
  *Pozn. Vynecháme-li část "*severity <severity_level>*", automaticky se použije výchozí hodnota* error.

Můžeme vložit několik příkazů assert za sebou a použít je i v sekci begin end bloku architektury. Také zde samozřejmě platí, že hodnota podmínky se musí znát už během kompilace.

Druhý a třetí assert ukazuje tvorbu textových výstupů. Čísla se převedou na řetězce atributem Image(), který patří k typu. V knihovnách je předdefinovaný pro většinu z nich. Texty se spojují operátorem &.

Poznámka: Výpisy textů se používají při simulaci obvodu nebo k hlášení chyb vzniklých při překladu. V obvodu se nedají realizovat. Tvorba textových výstupů není silnou stránkou VHDL. Existují však externí knihovny dovolující vytvářet texty skoro C stylem a lze je volně používat pro nekomerční účely. Jedna takovou najdete na https://www.easics.com/products/freesics .

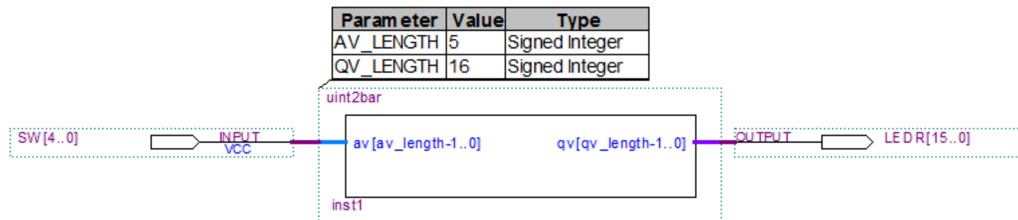
Příklad výpisu chyb:



Zapojení nahoře sdělí jen:

Info (10544): VHDL Assertion Statement at uint2bar.vhd(18): assertion is false - report "AV max value =15" (NOTE)

Pokud zadáme jiné parametry:



pak dostaneme:

Warning (10651): VHDL Assertion Statement at uint2bar.vhd(11): assertion is false - report "uint2bar displays only range 0 to 16 of av input" (WARNING)

Info (10544): VHDL Assertion Statement at uint2bar.vhd(18): assertion is false - report "AV max value =31" (NOTE)

Při chybném použití v symbolickém editoru schémat se překlad ani nedokončí.

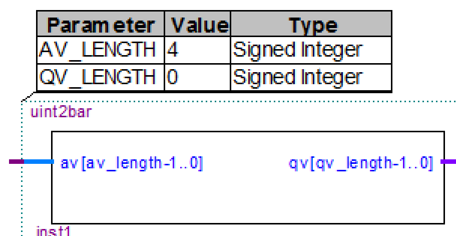


Schéma vytvořené v symbolickém editoru se totiž napřed konvertuje na VHDL kód, a teprve pak se překládá. Chyba vznikne už při konverzi do VHDL:

Error (275050): Bus range -1 is illegal in signal "port "qv[qv_length-1..0]" (ID uint2bar:inst1)"

Pokud bychom ale náš obvod použili přímo ve VHLD kódu, viz kapitola 6, pak by se zpráva vypsala:

Warning (10445): VHDL Subtype or Type Declaration warning at uint2bar.vhd: subtype or type has null range

Error (10652): VHDL Assertion Statement at uint2bar.vhd(9): assertion is false - report "uint2bar requires AV_LENGTH >0 and QV_LENGTH >0 " (FAILURE or ERROR)

5.2.a Co ještě chybí?

Poslední kód se již blíží skutečným VHDL popisům pro svou vysokou univerzalitu. Můžeme ho beze změny aplikovat na různé šířky sběrnic, a přesto není dokonalý. Chybí v něm jen:

- **Testbench** – profesionální návrh se považuje za neúplný, není-li u něho také soubor typu testbench pro automatické testování pomocí simulací psaný VHDL behavioral stylem. Bude na přednáškách.
- **Licence** – uvádí, jak je kód distribuován. Akademické kódy mívají "GNU General Public License".
- **Jazyk** – čeština může zůstat základem pro naše lokální obvody v kurzu LSP. Chceme-li VHDL kód více zpřístupnit, pak volíme angličtinu jak pro komentáře, tak pro identifikátory.

- **Komentáře** - uváděli jsme jen nejnutnější texty, abychom zkrátili řádky, a popisy jsme dávali do následujícího textu. Přidejte popisy aspoň do bloku entity a u hlavních veličin, aby se nemusel luštit jejich význam. Vložte i vysvětlivku, k čemu kód vlastně slouží, a případně i jméno autora. Neškodí ani vkládat poznámky před dílčí celky kódu specifikující, co se v nich vlastně provádí.
  Pozn. Hodně návrhářů časem zjistí, že komentáře psali hlavně kvůli sobě, aby později věděli, co kdysi dávno napsali a proč zrovna takhle. ☺

5.3 *** Cvičná úloha 4: Univerzální prioritní inhibitor

Zadání: Navrhněte obvod propouštějící pouze nejvyšší vstup ve stavu ´1´ na jemu číselně odpovídající výstup, takže nejvýše jen jeden jeho výstup je v ´1´. Obvod jste řešili pro 8 vstupů na str. 31. Zkuste vymyslet jeho kód pro libovolný počet vstupů od 2 do N, a to i s kontrolou hodnoty parametru.

Návod: Cyklus for generate nedovolí doplňovat řádky do when else. Napište inhibitor po signálech pomocí podmíněných příkazů. Nelze však mít univerzální flag, že se již propustil nějaký výstup, protože do signálu můžete zapojit jen jednou. Vytvořte si tedy pomocný vektor, například inh (*inhibit*), který bude svým rozsahem shodný s výstupem. Operací OR v něm tvořte rekurzivně podmínky, že nějaký vyšší výstup je již v '1'. Hodnoty inh(i) lze využít v dalších podmínkách. Výstup inh s nejvyšší váhou leží mimo cyklus generate, protože platí, že výstup N-1 se vždy rovná vstupu N-1. Až výstupy s nižšími indexy i budou blokované inh. Možné řešení najde v příloze na str. 62.

5.4 Inicializace vektorů proměnných délek asociací others=>

Klíčové slovo others znamená veškeré dosud nepoužité hodnoty. Hodí se k asociativnímu přiřazení hodnot vektorům, u nichž jsme délky specifikovali pomocí generic, tedy nevíme, jak budou dlouhé. Z našeho kódu musí vyplynout jasná velikost už v době překladu. Asociace se zapisuje => operátorem.

Je-li asociace použita v konstantách, pak na její levé straně stojí určení indexů v poli, a to s možnými směry jak to tak downto, a na pravé straně požadovaná hodnota jeho členu či členů.

Možnosti nejlépe přiblíží následující demo-příklad:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity asociace is
   generic(N:natural:=8);
   port  ( X1, X2, X3, X4, X5, X6, X7 : out std_logic_vector(N-1  downto 0)  );
begin
   assert N>=7 report "Example requires N>=7" severity failure;
end entity;
architecture structural of asociace is
begin                                                  --   for N = 8          for N = 12
   X1 <= (others=>'0');                                --   X1="00000000"      X1="000000000000"
   X2 <= (others=>'1');                                --   X2="11111111"      X2="111111111111"
   X3 <= (others=>'Z'); --                             --   X3="ZZZZZZZZ"      X3="ZZZZZZZZZZZZ"
   X4 <= (0=>'0', others=>'1');   --                   --   X4="11111110"      X4="111111111110"
   X5 <= (N-1=>'0', others=>'1');          --          --   X5="01111111"      X5="011111111111"
   X6 <= (N-1=>'0',  1=>'0', others=>'1');        --   --   X6="01111101"      X6="011111111101"
   X7 <= (N-1 downto N-3=>'1', 1 to 3=>'1', others=>'0'); --   X7="11101110"      X7="111000001110"
end;
```

Všimněte si, že hodnota u inicializační asociace se specifikuje shodně s datovým typem prvků pole, i když výsledek ovlivní více členů, protož jde o asociace na indexované prvky pole. Pole std_logic_vector má členy výčtového datového typu std_logic, a tak píšeme pravé strany asociací v ´ ´ apostrofech.

Při inicializace X7 jsme v rozsazích použili jak směr downto tak i to, což zde můžeme, protože definujeme asociaci pro vytvoření inicializační konstanty během překladu, a teprve ta se přiřadí vektoru.

Kdybychom chtěli například X7 dát stejnou hodnotu jako poslední asociativní list:

```
X7 <= (N-1 downto N-3=>'1', 1 to 3=>'1', others=>'0');
```

pak by zápis vyžadoval více <= příkazů a others bychom se stejně nevyhnuli:

```
X7(N-1 downto N-3) <= "111";
X7(3 downto 1) <="111"; -- Here, we select the part of X7, thus, we cannot reverse defined range
X7(N-4 downto 4) <=(others=>'0');  X7(0)<='0';
```

Prostřední část vektoru X7(N-4 downto 4) má proměnlivou délku a bez others to tady nejde. Při N=7 by se navíc ohlásila chyba, takže bychom museli trvat na assert N>=8.

V příkazech jsme již striktně dodržovali směr downto určený definicí vektoru. Pokus o použití směru to (*např. X7(1 to 3) <="111";*) by vyvolal chybové hlášení obsahující text: "*…range direction of object slice must be same as range direction of object*".

5.5 Převod čísel integer na unsigned a signed

Jelikož integer má ve VHDL proměnnou délku, při konverzi z něj musíme vždy přesně specifikovat šířku typu signed a unsigned konstantním výrazem s hodnotou známou už v době překladu. Máme-li například:

```
signal xs : signed(10 downto 0);  signal xu : unsigned(5 downto 0);
signal anyinteger : integer;
```

pak konverzi na zvolenou šířku lze provést funkcemi knihovny ieee.numeric_std, a to to_signed() a to_unsigned(), kde se šířka výhodně zadá atributem délky cílového signálu, ale lze užít i konstantu:

```
xs <= to_signed( -512, xs'LENGHT);  xu<=to_unsigned(anyinteger, 6);
```

5.6 *** Cvičná úloha 5: Rozšiřte prioritu přerušení pomocí generic

Rozšiřte prioritu přerušení ze str. 28 pomocí generic definic na univerzální modul.

Kód má již obtížnější strukturu, a tak uvádíme jeho fragment coby nápovědu. Dokončete vyznačené části. Své řešení si můžete ověřit na str. 64.

```
--Universal interrupt priority decoder
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity irq_priorityN is
   { write here correct definitions of generic parameters IRQ_MAX and Q_LENGTH  }
   port (  -- interrupt request, higher index has higher priority
           IRQ : in std_logic_vector(IRQ_MAX downto 1);
           -- unsigned number of the highest active interrupt input
           Q : out std_logic_vector(Q_LENGTH-1 downto 0) );
   { insert here tests of generic parameters consistency }
end;
architecture dataflow of irq_priorityN is
type qtmp_array_t is array { finish array type definition };
signal qtmp : qtmp _array_t;
begin
    qtmp (IRQ'LOW) <= (others=>'0') when IRQ(IRQ'LOW)='0'
                                      else to_unsigned(IRQ'LOW, Q_LENGTH);
     { insert here  for generate statement }
    Q <= std_logic_vector(qtmp(IRQ'HIGH));
end;
```

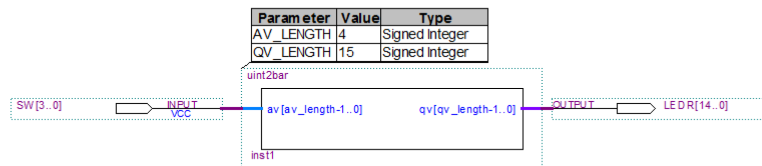
6 VHDL stylem "Structural"

VHDL umožňuje tvořit návrhy hierarchickou strukturou — můžeme obvod rozdělit na dílčí jednodušší entity, které samostatně odladíme. Nakonec definujeme jejich propojení stejně jako v editoru schémat, jen ho vytvoříme v textu. VHDL nenabízí takovou přehlednost u menších celků jako grafická schéma, ale zato propojení částí, které je v něm popsané, nabízí následující výhody:

- text se píše rychleji než kreslení schémat, a to dokonce i u jednoduchých obvodů;
- lze lépe porovnávání změny jednotlivých verzí;
- zjednoduší se složitější grafická schémata, která už ztrácí přehlednost pro množství propojek;
- VHDL obvod lze procházet ladicími nástroji, třeba jako je třeba ModelSim.

6.1 @Příklad XIII. - Použití prioritního inhibitoru ve VHDL

V předchozí části jsme si vytvořili univerzální lineární ukazatel, který můžeme testovat ve schématu. Každá změna generic parametrů ale vyžaduje edici rozsahů vstupů SW a výstupů LEDR.



Obrázek 8 - Užití prioritního inhibitoru ve schématu

Můžeme měnit rozsahy na jednom místě jen tehdy, pokud schéma přepíšeme do VHDL. Ukážeme si napřed celý kód a pak popíšeme jeho části. Za účelem výkladu jsme očíslovali řádky a k těm zvýrazněním existuje popis v dalším textu:

```
-- VHDL code with uint2bar  - downloadable into DE2 board                Řádka.   Strana.
library ieee;                                                             --L01
use ieee.std_logic_1164.all; use ieee.numeric_std.all;                    --L02
library work;                                                             --L03     40
entity demo_unit2bar is                                                   --L04
   generic (AV_LENGTH1 : integer:=4; QV_LENGTH1 : integer:=15);           --L05     40
   port (  SW :  in  std_logic_vector(AV_LENGTH1-1 downto 0);             --L06     40
           LEDR :  out  std_logic_vector(QV_LENGTH1-1 downto 0));         --L07
end;                                                                      --L08
architecture dataflow of demo_unit2bar is                                 --L09
component uint2bar                                                        --L10     40
      generic (   AV_LENGTH : integer:=4; -- bit length of av input       --L11
                  QV_LENGTH : integer:=15); -- bit length of qv output    --L12
      port( av : in std_logic_vector(AV_LENGTH-1 downto 0);               --L13
            qv : out std_logic_vector(QV_LENGTH-1 downto 0));             --L14
end component;                                                            --L15
begin                                                                     --L16
   inst1 : uint2bar                                                       --L17     41
     generic map(   AV_LENGTH => AV_LENGTH1,                              --L18     42
                    QV_LENGTH => QV_LENGTH1)   -- no ; after ) !          --L19
     port map(av => SW, qv => LEDR);                                      --L20
end;
```

L03 – library work

Příkazem library work; si zpřístupníme soubory, které máme v Quartus projektu vložené na záložce Files. Provedeme to příkazem library, i když work ve skutečnosti není knihovnou. Jeho význam se víc blíží klíčovému slovu this v objektových programovacích jazycích. Označuje právě otevřený projekt. Nemuseli jsme kvůli tomu ani vkládat samostatný řádek L03, ale identifikátor work by se dal přidat do řádku L01:

```
library ieee, work;
```

L04 - generic

Zavedli jsme si generic parametry s názvy AV_LENGTH1 a QV_LENGTH1 , tedy s odlišnými identifikátory od těch v uint2bar jedině kvůli přehlednosti. Mohli jsme použít i stejné, AV_LENGTH a QV_LENGTH, ale pak by bylo méně jasné, ke které definici se vztahují, jestli k uint2bar či k demo_uint2bar.

L05 a L06 – port

- Pokud soubor **je** **Top-Level entity** (tu v Quartusu vidíme v okně Project Navigator na jeho záložce Hierarchy), pak se vstupy a výstupy mapují podle přiřazení definovaných v Assignments. U desky DE2, jsme si je načetli z "DE2_pin_assignments.csv" v Quartus menu Assignments ->Import Assignments. SW poté definuje připojení k signálům, které vedou z přepínačů ke vstupům FPGA obvodu, zatímco LEDR označuje napojení na jeho výstupy ovládající k řadu červených led diod.
- Pokud soubor **není** **Top-Level entity**, pak se identifikátory nikdy nemapují podle Assignments na vývody FPGA a LEDR i SW by zůstaly v demo_unit2bar.vhd pouhými identifikátory. Použijeme-li však u nich název z Assignments, pak specifikujeme požadavek, aby se obvod připojil na stejnojmenný signál, bude-li využitý jako podřízená součást jiného zapojení.

L10 až L15 – component

VHDL vyžaduje z bezpečnostních důvodů, aby se vložila úplná hlavička použitého obvodu. Vytvoříme ji tak, že okopírujeme část entity z VHDL kódu obvodu, na který odkazujeme. Náš příklad ji měl v kapitole 5.1 na str. 33 a poté rozšířenou v kapitole 5.2 na str. 34:

```
entity uint2bar is
   generic (   AV_LENGTH : integer := 4;  -- bit length of av input
               QV_LENGTH : integer := 16); -- bit length of qv output
   port (av : in std_logic_vector(AV_LENGTH-1 downto 0) ;
         qv : out std_logic_vector(QV_LENGTH-1 downto 0));
begin -- passive process - its statements are executed in compile time only
   assert AV_LENGTH >0 AND QV_LENGTH >0
   report "uint2bar requires AV_LENGTH >0 and QV_LENGTH >0 " severity failure;
end;
```

Máme-li v entitě pasivní proces, tedy část začínající begin, v níž se kontrolují generic parametry, vymažeme ho, vkládáme jen informaci, ne kód. Klíčové slovo entity nahradíme component a místo end napíšeme end compoment; Nelze napsat jen end. Alternativně lze zakončit delším názvem end component uint2bar;

```
component uint2bar
   generic (  AV_LENGTH : integer:=4;  -- bit length of av input
              QV_LENGTH : integer:=15); -- bit length of qv output
   port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
           qv : out std_logic_vector(QV_LENGTH-1 downto 0));
end component;
```

Nutné vkládání komponent brání mylnému použití jiného obvodu s náhodně stejným názvem. VHDL vyžaduje nejen shodu jména obvodu a typů parametrů, ale současně i názvů všech vstupů a výstupů. Pokud nenajde nic s dokonalou shodou, ohlásí se chyba.

Poznámky:

1. Můžeme si vložit libovolné množství komponent bez ohledu na to, zda je použijeme nebo ne. Překladač nevyžaduje, aby se odkazovalo na jejich deklarace. Vložením komponenty nic nevzniká, protože se jedná pouze o informativní údaj. Není tedy nutné během vývoje kódu okamžitě mazat komponenty, na něž se momentálně nikde dále neodkazujeme.
2. Bloky component se dají sdružit do balíčku (package), tedy něčeho analogického *.h souborům jazyka C, a pak vkládat jen package, viz dále v kapitole 7 na straně 54, věnované tvorbě knihoven pomocí deklarace knihovních balíčků.
3. Na signály i parametry uvedené v komponentě lze odkazovat jedině v sekci map, viz dále. Z jiných částí kódu jsou nepřístupné.
4. *V bloku component lze uvést i neúplné deklarace ("incomplete type declarations") zavedené ve VHDL'93 pro složitější konstrukce. Můžeme tady vynechat default hodnoty u generic parametrů AV_LENGTH, QV_LENGTH nebo dále opomenout i rozsahy vektorů v port.* Povolená, ale nedoporučovaná zkrácená deklarace komponenty:

```
component uint2bar
        generic (    AV_LENGTH, QV_LENGTH : integer);
        port(  av : in std_logic_vector; qv : out std_logic_vector);
end component;
```

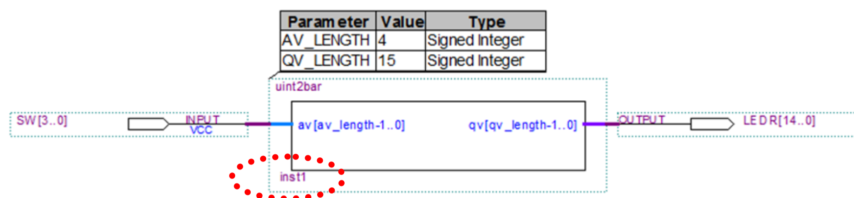
VHDL kódy nalezené na webu občas používají podobné zkratky, ale obecně se radí používat více úplné deklarace — ty mají menší problémy s přenositelností.

L17 - instance obvodu

Na řádce L17

inst1 : uint2bar

začínáme vytvářet instanci obvodu uint2bar, kde **inst1** je její jméno, jímž může být libovolný platný unikátní VHDL identifikátor. Ve schematickém diagramu vidíte jména instancí u každého obvodu vlevo dole, viz Obrázek 8, jehož kopii uvádíme níže:



Instance lze vytvářet jedině v bloku begin end architektury a každá musí mít unikátní identifikátor, před nímž nestojí žádné určení. Nejedná se totiž o analogii signálu či konstanty, ale o vložení kompletního obvodu, který se sestaví podle popisu v jeho VHDL souboru.

Chceme-li vytvořit další instance od uint2bar, můžeme. Stačí mít jedenkrát vloženou komponentu, pak lze od ní vytvářet tolik instancí, kolik právě potřebujeme, samozřejmě odlišených unikátními identifikátory. A každá jednotlivá instance obvodu bude naprosto samostatná a nebude sdílet žádnou svou část s jinou.

Poznámka: V prvním přiblížení lze instanci obvodu považovat za velmi vzdálenou analogii tvorby instance třídy v objektovém programování, kde se třída (class) chová jako jakási šablona, podle níž se příkazem new vytváří nový objekt zavoláním odpovídajícího konstruktoru.

Zde analogie ale končí! *Zásadní odlišnost obvodů oproti klasickým programům spočívá v úplném oddělení jejich instancí. Ty nesdílí žádné své části. Ve fyzické syntéze se každé použití obvodu konvertuje na jeho nové kompletní vložení. To se v případě syntéze provádí i u volání VHDL funkcí či procedur. Jejich každá evokace nesměřuje na sdílený kód jako v klasickém programování, ale pokaždé vede na samostatnou fyzickou realizaci celé sestavy příkazů, tedy na již zmíněnou inline expansion.*

L18 až L20 - příkazy map

Příkaz tvorby instance, který začal na řádku L17, pokračuje dvojicí příkazů map, které definují signály připojené na instanci obvodu, lze užít jmenné asociace => nebo poziční.

Jmenné asociace ("keyword notation") mají syntaxi:

```
generic map(  generic_name1 => value1,
              generic_name2 => value2,
              ...
              generic_nameN => valueN )
port map(  port_name1 => signal_name1,
           port_name2 => signal_name2,
           ...
           port_nameN => signal_nameN );
```

POZOR - všimněte si, že

- * členy jsou od sebe oddělené čárkami, nikoliv středníky — jde o seznam asociací.
- * za závorkou není ; — příkaz pokračuje.
- * píšeme opět seznam asociací s členy oddělenými od sebe čárkami.
- * Až za) se vyskytuje středník— konec příkazu.

kde

- Pokud komponenta nemá generic parametry, sekce generic map se neuvádí.
- Označení **generic_nameX** udává jméno parametru shodné s jeho deklarací v sekci generic komponenty a **valueX** specifikuje jemu přiřazenou hodnotu ve vytvářené instanci. Zapíšeme-li valueX výrazem, pak jeho výsledek musí být známý v době překladu — jde o konstantu.
- Označení udává název vstupu nebo výstupu, a to opět shodný s deklarací v sekci port komponenty, a signal_name1 určuje, k jakému signálu bude připojen.
  - Jedná-li o asociaci na **port_nameX**, který je vstupem komponenty, pak lze jeho hodnotu specifikovat i **výrazem** a ten nemusí mít konstantní hodnotu v době překladu, na rozdíl od generických parametrů, avšak nedoporučuje se to. Quartus dovolí **výrazy v** port map , žel překladač v ModelSimu je často odmítne z implementačních důvodů. Kvůli tomu bývá lepší napřed přiřadit výrazy do signálů a teprve ty mapovat na vstupy.
  - **Výstup** musí být samozřejmě vždy mapován na signál.
- Jde o asociativní seznamy, a tak lze jejich členy uvést v **libovolném pořadí**, ale pro přehlednost se doporučuje zachovávat ho.

Poziční asociace ("positional notation") mají zjednodušenou syntax:

```
generic map(  value1, value2,
              ...
              valueN )
port map (  signal_name1, signal_name2,
            ...
            signal_nameN );
```

Hodnoty se v pozičních asociacích musí pochopitelně objevit **přesně v pořadí deklarací** jednotlivých parametrů, tj. vstupů a výstupům v bloku komponenty. Pro případné použití výrazů zde platí pravidla uvedená u jmenných asociací.

Rozhodnutí zda použit jmenných či poziční asociace závisí na návrháři. Pro jednodušší komponenty lze volit méně upovídané poziční asociace, zejména v případech, kdy deklarace použitých komponent jsou vložené do entity, a tak si z nich lze snadno přečíst, čemu hodnoty vlastně přiřazujeme. Máme-li složitější komponenty nebo komponenty definované v externích knihovnách, pak se vyplatí napsat přehlednější jmenné asociace, které navíc nabízejí vyšší imunitu vůči změnám v originálních entitách.

6.1.a Stejné názvy generic parametrů

Předchozí kód by fungoval, i kdyby se identifikátory parametrů v sekci generic entity nazvaly stejně jako v komponentě uint2bar:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity demo_unit2bar is
   generic (AV_LENGTH : integer:=4; QV_LENGTH : integer:=15);
   port (  SW :  in  std_logic_vector(AV_LENGTH-1 downto 0);
           LEDR :  out  std_logic_vector(QV_LENGTH-1 downto 0));
end;

architecture dataflow of demo_unit2bar is

 component uint2bar
   generic (   AV_LENGTH : integer:=4; -- bit length of av input
               QV_LENGTH : integer:=15); -- bit length of qv output
   port( av : in std_logic_vector(AV_LENGTH-1 downto 0);
         qv : out std_logic_vector(QV_LENGTH-1 downto 0));
end component;

begin

  inst1 : uint2bar
       generic map(AV_LENGTH => AV_LENGTH,
                   QV_LENGTH => QV_LENGTH)   -- no semicolon ; after ) !

       port map(av => SW, qv => LEDR);

end;
```

Komponenta má sice své identifikátory parametrů a vstupů a výstupů, ale na ně lze odkázat jedině v asociacích sekce jejích příkazů map, kdy se identifikátor na levé straně hledá jen v deklaraci komponenty, zatímco pravá strana se vztahuje k definicím vytvářeného VHDL kódu. Identifikátory se tedy překladači nepopletou. Ve VHDL kódech se toho často využívá. Tvoříme propojky a ty mohou mít shodná jména.

Poznámka: I v jazyce C existuje oddělení prostorů, v nichž se hledají identifikátory, viz:

```
struct test { int v; };
int main()
{     test test;
      test.v = 1; test.v++; printf("%d", test.v);
      return 0;
}
```

C program vypíše správný výsledek 2. Ve žlutě označeném řádku se totiž levý identifikátor **test** *hledá v odlišném jmenném prostoru než vytvářená proměnná* **test**. *Výše uvedený C program slouží leda jako demonstrační ukázka jmenných prostorů, a není nápovědou k vhodné tvorbě identifikátorů!*

6.1.b @Příklad XIV. - Prioritní inhibitor s funkcí test a blank

Předchozí kód si nyní rozšíříme o dva servisní vstupy test a blank, jimiž mohou servisní pracovníci rychle zkontrolovat, a to nezávisle na momentálním stavu vstupu, zda není některá led dioda vadná nebo trvale svítící. Testovací signály si připojíme na tlačítka KEY desky DE2:

test	blank	
'1'	-	*všechny výstupy LEDR jsou v '1'*
'0'	'1'	*všechny výstupy LEDR jsou v '0'*
'0'	'0'	*obvod pracuje jako prioritní inhibitor*

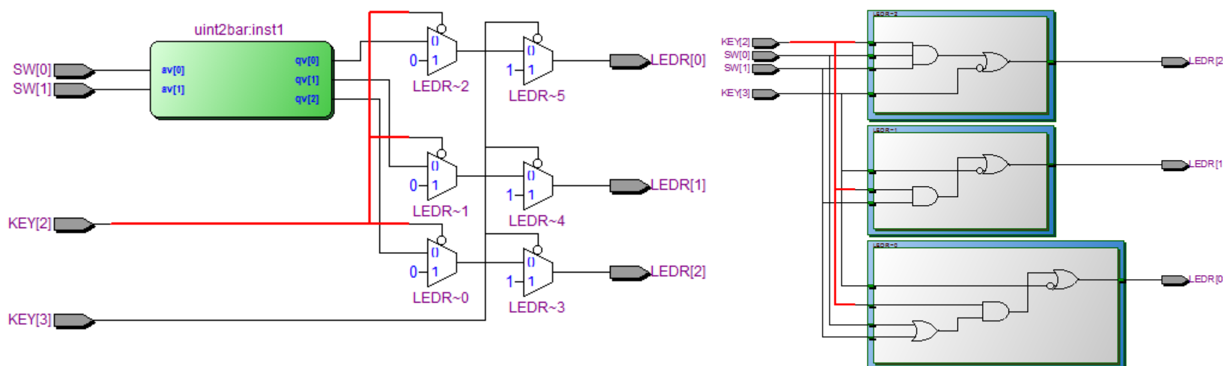
Podobné vstupy se často přidávají do průmyslových zařízení kvůli prověření výstupů.

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_unit2barTest is
  generic ( AV_LENGTH : integer:=2; QV_LENGTH : integer:=3);
  port (      SW :  in  std_logic_vector(AV_LENGTH-1 downto 0);
              KEY : in std_logic_vector(3 downto 2);
              LEDR :  out  std_logic_vector(QV_LENGTH-1 downto 0));
end;
architecture dataflow of demo_unit2barTest is
  component uint2bar
       generic (   AV_LENGTH : integer:=4; -- bit length of av input
                   QV_LENGTH : integer:=15); -- bit length of qv output
       port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
               qv : out std_logic_vector(QV_LENGTH-1 downto 0));
  end component;
signal tmp : std_logic_vector(LEDR'RANGE);
begin
    LEDR <=  (others=>'1') when KEY(3)='0' else
             (others=>'0') when KEY(2)='0' else tmp;
    inst1 : uint2bar generic map(AV_LENGTH, QV_LENGTH) port map(SW, tmp);
end;
```

Úpravy v kódu demo_unit2bar jsou následující:

- Uložili jsme soubor demo_unit2bar.vhd jako demo_unit2barTest.vhd a v tom jsme adekvátně změnili jméno v entitě i v architektuře.
- Snížili počet vstupů v sekci generic entity, abychom si ověřili, že nám to funguje, a také ukázali, že stačí změna jen tam a do komponenty nemusíme sahat. Její generic parametry se nastaví v sekci map.
- Přidali jsme tlačítka KEY desky DE2 do entity, KEY(3) bude funkce **test** a KEY(2) zase **blank**. Pozor, KEY tlačítka dávají '0' při stisknutí, v klidu '1'. Kvůli tomu je testujeme na '0', tj. stisknuto.
- Definovali jsme si pomocný signál tmp, který má rozsah výstupu LEDR, a v port map jsme na něj zapojili výstup komponenty qv.
- Do begin end architektury jsme přidali příkaz when else respektující prioritu funkce test nad blank. Vložili jsme ho úmyslně před tvorbu instance coby důkaz, že pořadí concurrent příkazů není důležité.
- Upravili jsme ještě map asociace na poziční jako ukázku stylu zápisu. U tak malého obvodu se dají používat bez rizika.

Úvodní schéma RTL Map ukazuje, že překladač zapojil kaskádu multiplexorů za náš uint2bar. Malý kroužek značí invertovaný vstup. Zapojení se dále minimalizovalo a jeho Technology Map již představuje jednoduchý logický obvod. Multiplexor ovládaný stisknutím tlačítka KEY(3), rozsvícení všeho, byl nahrazený hradly OR. Multiplexor tlačítka KEY(2), blank zhasnutí všech diod, se realizoval hradly AND.



Strukturální VHDL dovoluje provádět změny rychleji. Kdybychom nové funkce pracně doplňovali do symbolického schématu, kreslení by nám zabralo mnohem delší dobu.

6.2 *** Cvičná úloha 6: 7segmentový dekodér s potlačením úvodních 0

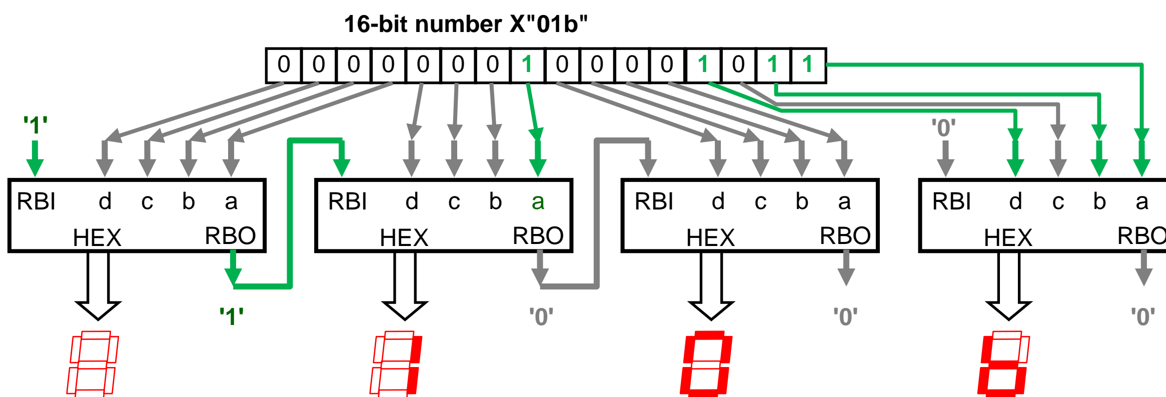
Pokud se zobrazuje číslo s větším počtem číslic, někdy se zhasínají úvodní 0. Například místo dlouhého čísla "0012" se vypíše jen " 12", tj. první dvě nuly se nerozsvítí.

Funkci lze realizovat pomocí jednoho přídavného vstupu, který označíme RBI, ripple blanking input, a výstupu RBO, ripple blanking output.

Číslice na d,c,b, a	d, c, b, a	RBI	RBO	Funkce
0	"0000"	1	1	zhasnutý segment
0	"0000"	0	0	zobrazena 0
X"1" až X"F"	/= "0000"	X, tj. 1 nebo 0	0	zobrazeno číslo 1 až F

Tabulka 8 - 7segmentový dekodér s potlačením 0

Nové dekodéry lze zřetězit a vytvořit jim libovolně dlouhý výstup. Na vstup dekodéru RBI nejvyšší číslice připojíme '1', protože před ním nic není. U dalších stupňů propojujeme RBO s RBI. Pokud chceme, aby vždy svítila alespoň jedna číslice, pak u pravého dekodéru připojíme RBI na '0'.



Obrázek 9 - Čtyři zřetězené 7segmentové dekodéry

Zadání úkolu:

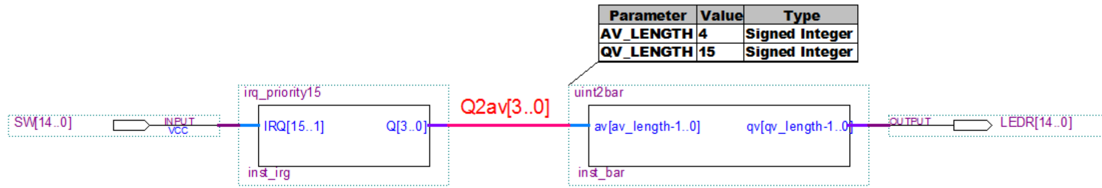
Předchozí 7segmentový dekodér, který jste již vyřešili ve cvičné úloze 3.5 na str. 27, si rozšiřte o dva nové vstupy RBI a RBO. Hotový původní obvod si vložte jako komponentu do nového kódu a vytvořte její instance.

Návod: Postupujte podobně, jako jsme řešili obvod demo_unit2barTest. Zatím si vytvořte pouze dekodér pro jednu číslici se vstupem a výstupem pro zřetězení. Zobrazení 16bitového čísla si uděláte až později v další samostatné úloze na str. 53. Před ní si probereme vícenásobné instance.

Možné řešení najdete v příloze na str. 65

6.3 @Příklad XV. - Propojení několika obvodů

Nyní si zkusíme strukturálním stylem provést propojení dvou obvodů. Využijeme náš obvod prioritního přerušení, jehož výstup si zobrazíme na lineárním ukazateli. Výsledné zapojení bude vypadat takto:

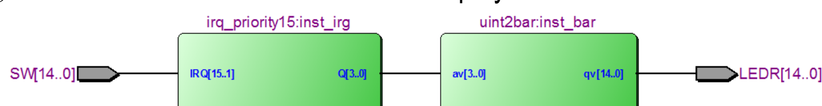


Ve schématu se objevil nový prvek, a to pojmenovaný vodič, zvýrazněný červeně, který jsme vytvořili v editoru schémat Quartusu. Vybrali jsme vodič a v jeho kontextovém menu jsme zvolili Properties a napsali náš **název spolu s rozsahem** v notaci. Pokud vodič ve schématu nepojmenujeme, překladač mu přidělí automaticky vygenerovaný název během kompilace. Zde jsme to provedli sami coby referenci, jelikož ve VHDL kódu musíme vždy definovat vodiče pro všechny vzájemné spojky mezi obvody.

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_irq_bar is  port ( SW :  in  std_logic_vector(14 downto 0);
                               LEDR :  out  std_logic_vector(14 downto 0));
end;
architecture dataflow of demo_irq_bar is
 component uint2bar
    generic ( AV_LENGTH : integer:=4; QV_LENGTH : integer:=15);
    port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
            qv : out std_logic_vector(QV_LENGTH-1 downto 0));
 end component;
 component irq_priority15 is
    port (  IRQ : in std_logic_vector(15 downto 1);
            Q : out std_logic_vector(3 downto 0) );
 end component;
 signal Q2av :  std_logic_vector(3 downto 0);
begin
 inst_bar : uint2bar generic map(Q2av'LENGTH, LEDR'LENGTH) port map( Q2av, LEDR );
 inst_irg : irq_priority15 port map( IRQ=>SW, Q=>Q2av );
end architecture;
```

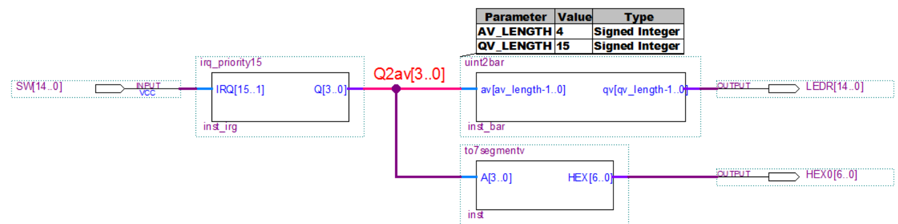
- V definici pomocného vodiče Q2av musíme napsat celý rozsah, protože nelze odkázat na výstup Q komponenty irq_priority15. Její členy lze použít výhradně na levé straně asociací příkazů map. Nesnáz by šla vyřešit definicí subtypu v balíčku, který probereme v kapitole 7 na str. 54.
- Při tvorbě instance inst_bar jsme mohli vynechat její sekci generic map, protože nastavujeme délky shodné s jejich výchozími hodnotami. Rozhodli jsme se však generic map ponechat a uvést v ní délky odvozené od vektorů, abychom zvýšili čitelnost kódu.

RTL Map nám ukáže zapojení, které jsme si přáli. Tentokrát na něm nezobrazujeme nitra obvodů, neboť jsme si je už ukázali v předchozích kapitolách. Chceme-li se na ně podívat i zde, vybreme obvody, které nás zajímají, a v kontextovém menu zadáme Display Content/Hide Content.



6.4 *** Cvičná úloha 7: Přidání 7segmentového displeje

Předchozí VHDL kód demo_irq_bar rozšiřte o výstup na 7segmentový displej, viz obrázek. Máte-li jen 7segmentový dekodér s oddělenými vstupy a0, a1, a2 a a3, tak si napište v port map asociace přímo na ně.



Možné řešení najdete v příloze na str. 66.

Poznámka:Můžete si kvůli zjednodušení chcete vyrobit i další 7segmentový dekodér s vektorovým vstupem av(3 downto 0), nazvaný třeba to7segmentv, a vložit si ten.
K jeho tvorbě ale rozhodně využijte původní obvod to7segment s oddělenými vstupy, který si do to7segmentv vložíte jako komponentu. Od ní si vytvořte instanci, jejíž vstupy namapujete na av vektor. Nedělejte to7segmentv.vhd tak, že si do něho nakopírujete celý kód to7segment s úmyslem editovat ho na vektorový vstup. Jde o špatný styl, jímž se duplikují i případné chyby. Vložíte-li instanci, pak odkazujete na její stávající kód, ten zůstává pořád na jednom místě a jeho případná oprava se automaticky promítne i do dalších částí.

6.5 @Příklad XVI. - Vícenásobné instance ve vektoru majority

Zadání: Předpokládejme, že máme vektor obsahující výstupy 6 trojic detektorů, třeba od světelných závor. Máme tedy 18 vstupů, přičemž vstupy s indexy 0 až 2, patří první trojici detektorů, indexy 3 až 5 druhé trojici, a tak dále, až po poslední šestou trojici s indexy 15 až 17.

Přejeme si každou trojici detektorů zpracovat funkcí majority, to znamená, nejméně dva detektory musí být najednou v '1'. Podobná funkce by se třeba mohla vyhodnotit situaci, když narušení jednoho paprsku zaviní drobné zvíře či hmyz, a pouze větší objekt přetne dva paprsky naráz.

Nenapíšeme 6 logických rovnic, ale vytvoříme si instance majority. Kód píšeme pro desku DE2. Spínače SW(17 downto 0) budou hodnoty čidel a výsledek zobrazíme na zelených led diodách LEDG(5 downto 0).

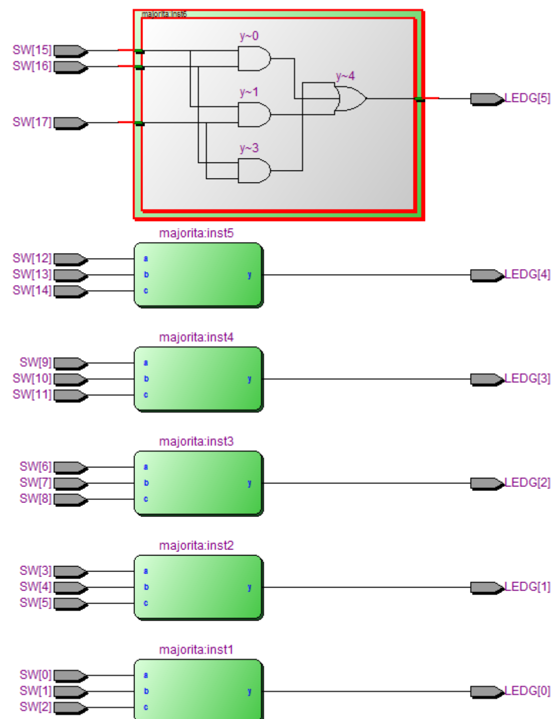
```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_majorita6g is
     port (  SW :  in  std_logic_vector(17 downto 0);
             LEDG :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of demo_majorita6g is
component majorita is port( a, b, c : in std_logic; y : out std_logic);
end component;
begin
     inst1 : majorita port map(a=>SW(0), b=>SW(1), c=>SW(2), y=>LEDG(0));
     inst2 : majorita port map(SW(3), SW(4), SW(5), LEDG(1));
     inst3 : majorita port map(SW(6), SW(7), SW(8), LEDG(2));
     inst4 : majorita port map(SW(9), SW(10), SW(11), LEDG(3));
     inst5 : majorita port map(SW(12), SW(13), SW(14), LEDG(4));
     inst6 : majorita port map(SW(15), SW(16), SW(17), LEDG(5));
end;
```

Obrázek 10 - Majorita 6 skupin po 3

Výsledný kód ukazuje několika násobné použití příkazu vytvoření instance. V prvním řádku jsme pro demonstraci použili jmenné asociace, zatímco v následujících jsme aplikovali poziční.

A vytvořili jsme obvod expresně, schéma bychom tak čiperně nenakreslili. Jeho sestavení jsme přenechali překladači.

Úvodní schéma RTL Map vidíme vpravo, přičemž u horní majority jsme nechali zobrazit i její vnitřní zapojení. Ostatní skryté v blocích mají zcela totožnou strukturu.



Obrázek 11 - RTL Map šesti majorit

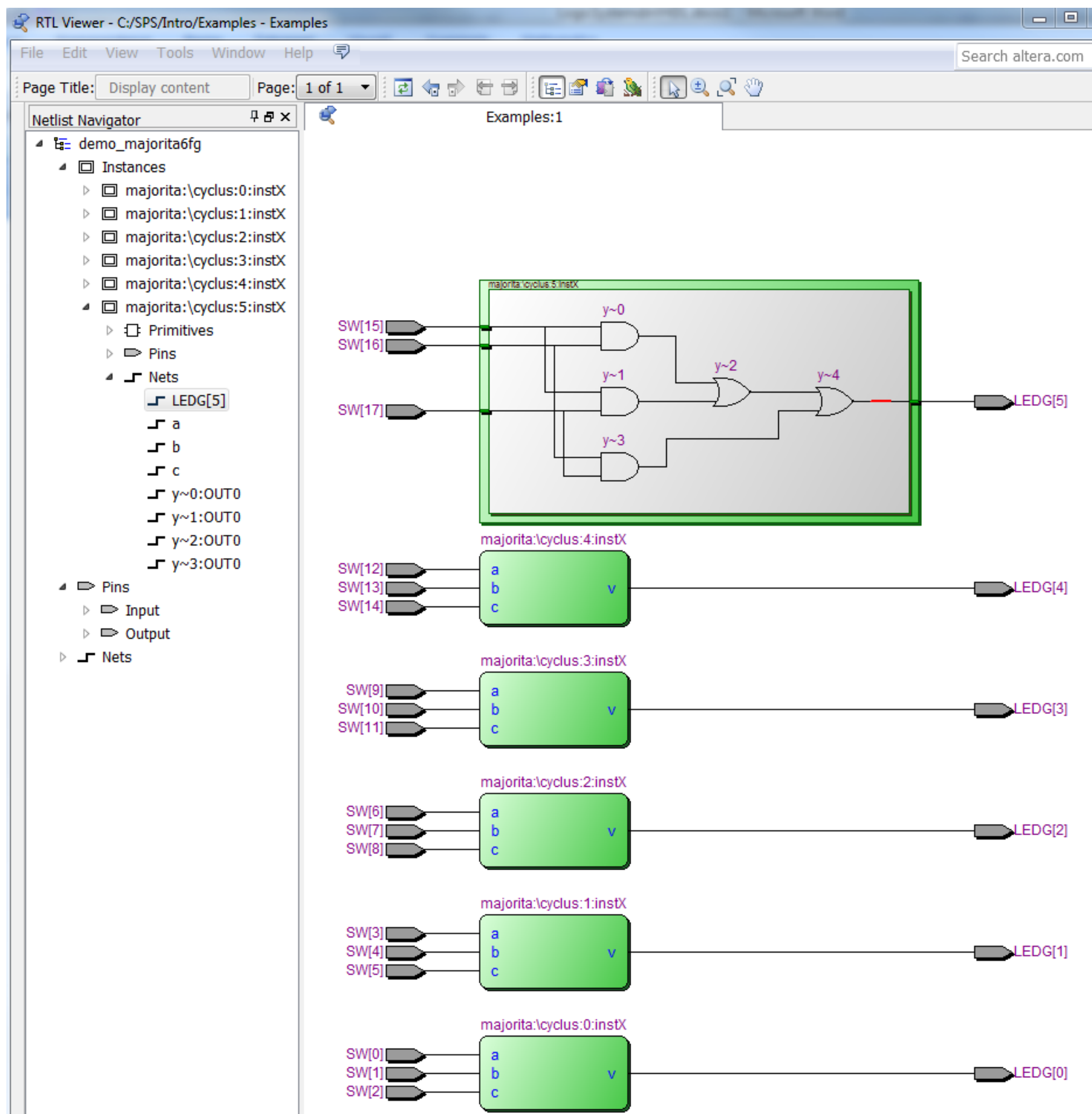
6.6 @Příklad XVII. - Vektor majority s for generate

Nabízí se otázka, zda by v předchozím příkladu nešlo využít příkaz for generate? Zkusme to. Vytvoříme si nový VHDL soubor demo_majorita6fg.vhd a v něm napíšeme cyklus.

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_majorita6fg is     port (  SW :  in  std_logic_vector(17 downto 0);
                                       LEDG :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of demo_majorita6fg is
component majorita is     port( a, b, c : in std_logic; y : out std_logic);
end component;
begin
   cyclus: for i in 0 to 5 generate
      instX : majorita port map(a=>SW(3*i), b=>SW(3*i+1), c=>SW(3*i+2), y=>LEDG(i));
   end generate;
end architecture;  -- for clarity of our code, we have emphasized that this end belongs to architecture
```

Kód 1 - Tvorba instance ve for generate

Popis obvodu vygeneruje stejně jednoduchou RTL Map jako bez příkazu for generate. Během překladu se sice opakovaně vkládá identifikátor instX, ale Quartus ho nahradil automaticky generovaným názvy "\cyclus:0:instX" až "\cyclus:5:instX".



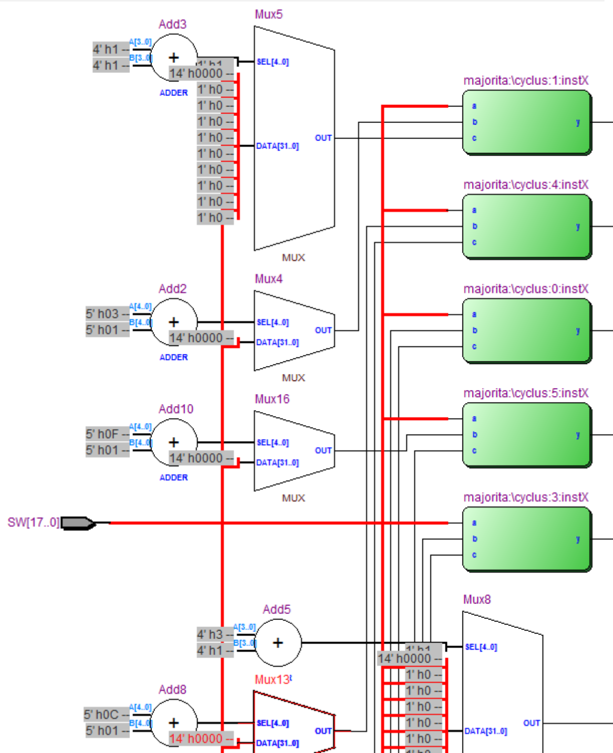
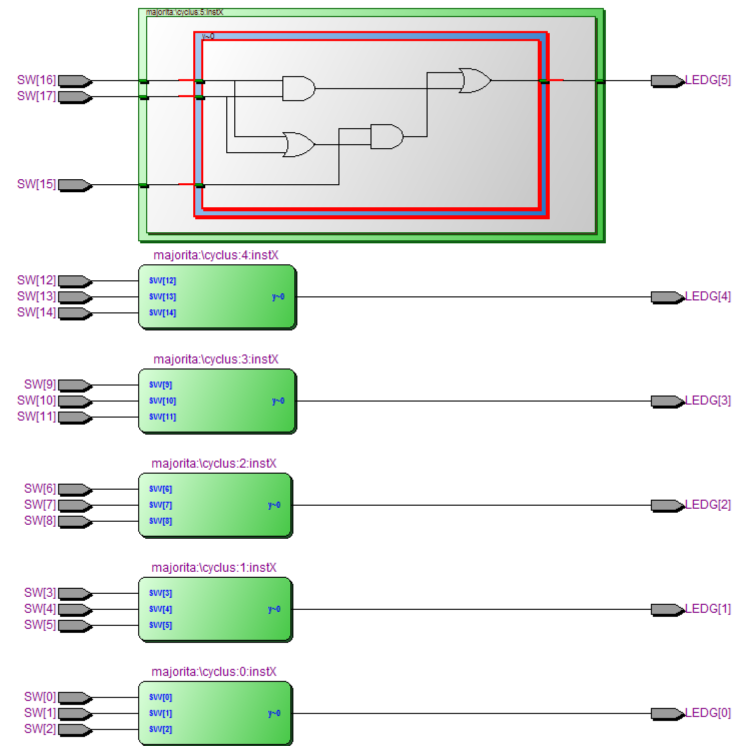
6.6.a Špatné užití pomocné definice

Uvnitř předchozího kódu se píší výrazy 3*i, 3*i+1 a 3*i+2, což vzbuzuje iluzi, že by nám usnadnilo práci zavedení pomocné definice k uložení 3*i, což by se v C programu třeba považovalo i za dobrý styl, jímž se šetří trojí násobení. Ve VHDL se však analogická operace **obrátí často v pravý opak**.

```vhdl
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_majorita6fg_wrong is
  port ( SW : in  std_logic_vector(17 downto 0); LEDG :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of demo_majorita6fg_wrong is
component majorita is   port( a, b, c : in std_logic; y : out std_logic);
end component;
begin
      cyclus: for i in 0 to 5 generate
          signal n : integer range 0 to 15; -- the range hint for compiler to simplify RTL Map,
      begin
          n<=3 * i;
          instX : majorita port map(a=>SW(n), b=>SW(n+1), c=>SW(n+2), y=>LEDG(i));
      end generate;
end architecture;  -- for clarity of our code, we have emphasized that this end belongs to architecture
```

Obrázek 12 - Instance komponenty ve for - generate

Kód se přeloží v pořádku, a přesto není dobrý.

Část **RTL Map** - *obří, něco bude špatně:-(*	**Technology Map** - *naštěstí minimální OK*
	

Tabulka 9 - Opakované vytvoření instance pomocí for generate

V RTL Map se jednotlivá násobení 3*i nahradila konstantami 0, 3, 6, až 15, jelikož parametr i je uvnitř cyklu pouze read-only a překladač kompiluje for smyčky v kódu pro syntézu stylem inline expansion, tedy vkládání. Sčítání se zase reprezentovalo sčítačkami a indexování prvků SW provádí 32-vstupovými multiplexory, na jejichž nepoužité vstupy se připojily '0' (v obrázku uvedené jako 1'h0).

Poskytli jsme překládači nápovědu, že n uvnitř cyklu bude nabývat hodnot jen v rozsahu od 0 do 15. Kompilace by sice proběhla v pořádku i bez tohoto upřesnění, ale RTL Map by pak vypadala ještě složitěji než nyní, jelikož pomocný signál n by se v ní modeloval 32bitovým signed integer.

Opakované užité identifikátoru instance *instX* se uvnitř cyklu opět nahradilo interními názvy \cyklus:1:instX až \cyklus:6:instX. Totéž se provedlo i u šesti vložených přiřazení n<=3 * i; I zde překladač již uplatnil behavioral kompilaci a opakované zápisy nahradil s využitím automaticky generovaných identifikátorů.

Naštěstí jsme do n záměrně zapisovali jen hodnotu závislou na parametru i cyklu, takže se po minimalizaci zapojení výrazně zjednodušilo. Technogy Map ukazuje, že v závěru dostaneme úplně stejný obvod jako v předchozí variantě bez for generate.

Proč ale vyšla RTL Map tak složitá? Zamaskovali jsme totiž před překladačem, že zpracováváme oddělené trojice. I když máme všechny vstupy v jednom vektoru, kompilátor nepřišel na jejich vazbu. Když modeloval úvodní kód jako RTL Map, zjistil, že indexy na členy SW vektoru závisí na signálu n, jehož hodnota se nastavuje kdesi jinde, a vkládal 32-vstupové multiplexory pro výběr jednoho z 18 prvků.

> *Poznámka: Musíme si tady uvědomit, že my lidé poznáme, že n se používá jen v* port map *pro tři členy odvozené od 3*i, protože vidíme celý kód. Překladač sleduje pouze jeho zlomek. Jde o situaci analogickou zpracování obrázků. Člověk vidí celý snímek naráz, ale algoritmus zpracovává v jednom časovém okamžiku pouze malý fragment. Samozřejmě by se daly do Quartusu přidat další heuristiky, které by detekovaly obdobné situace. Existuje jich v něm hodně, ale ne pro každý možný případ, jelikož nárůst počtu jejich aplikací by citelně prodloužil dobu překladu. Musíme kompilátoru napovědět sami.*

Vidíme-li RTL Map komplikovanější, než by odpovídalo očekávané struktuře námi navrhovaného obvodu, pak náš kód má špatnou strukturu.

*Námitka: **A co když netušíme, jak má výsledek zhruba vypadat?** No, pak začínáme návrh ze špatné strany. Lepší bývá napřed si rozmyslet, jak by zapojení mělo zhruba fungovat a teprve pak psát jeho VHDL kód. Přitom si kontrolujeme, zda výsledek aspoň trochu odpovídá naší výchozí představě.*

Komplikovaná RTL Map není tragédií, dokud nám neroste složitost výsledného obvodu a doba kompilace se prodlužuje leda o několik vteřin či dokonce milisekund. Pokud nám však vyjde něco složitějšího, musíme si ověřit, zda překladač stále zvládá minimalizace. Zkoumat Technology Map, to se žel hodí jen u menších obvodů, u nichž zůstává stále přehledná. U složitějších nám už příliš nenapoví. Tam se mohou analyzovat doby zpoždění mezi vstupem a výstupem a počty použitých prvků.

*Poznámka: Nutno zde zdůraznit, že **zavedení pomocného signálu či proměnné není vždy špatným krokem.** Ve VHDL kódech se jím často výrazně zjednoduší výsledný obvod, ale rozhodně ne ve všech případech! Existují situace, jako například výše uvedená, kdy se podobné pokusy zvrátí v pravý opak. Jak je poznat? Buď na základě zkušenosti, nebo vždy z RTL Map.*

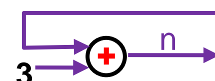
6.6.b Fatální chyba: Opakované přiřazení signálu ve for generate

Kódy for-generate se musí psát velmi opatrně. V následujícím "obvodo-vražedném" VHDL zvyšujeme n o 3 ve smyčce generate. V klasickém programu jde o běžnou operaci, tady o likvidační konstrukci:-(

Quartus kód zdánlivě přeloží bez chyb, vypíše však výstrahy, protože výsledkem bude mizerný obvod s tendencemi k rozkmitání. Zvyšování n o 3 by nevadilo v C programu, avšak zde **zapojujeme obvod!**

```
begin -- architecture
    cyclus: for i in 0 to 5 generate
      signal n : integer range 0 to 15:=0; -- assigning initial value,
    begin
      n <= n+3;  -- fatal operation not only inside for generate, but in many other concurrent parts
      instX : majorita port map(a=>SW(n), b=>SW(n+1), c=>SW(n+2), y=>LEDG(i));
    end generate;
end architecture;  -- for clarity of our code, we have emphasized that this end belongs to architecture
```

Cyklus for generate **nesmí** využívat jakoukoli hodnotu vzniklou v předchozích průchodech cyklem! Bez detekce náběžné hrany hodinového signálu nevzniká paměťový prvek použitelný v FPGA obvodech! Vytvoří se leda přísně zakázané nestabilní kombinační smyčky. Bude podrobněji vysvětleno na přednáškách.



Závěr:	Demonstrovali jsme, že techniky psaní klasických programů se nedají slepě napodobovat ve VHDL kódech určených pro fyzickou syntézu obvodů.

6.7 @Příklad XVIII. - Příkaz for generate pro oddělené vstupy

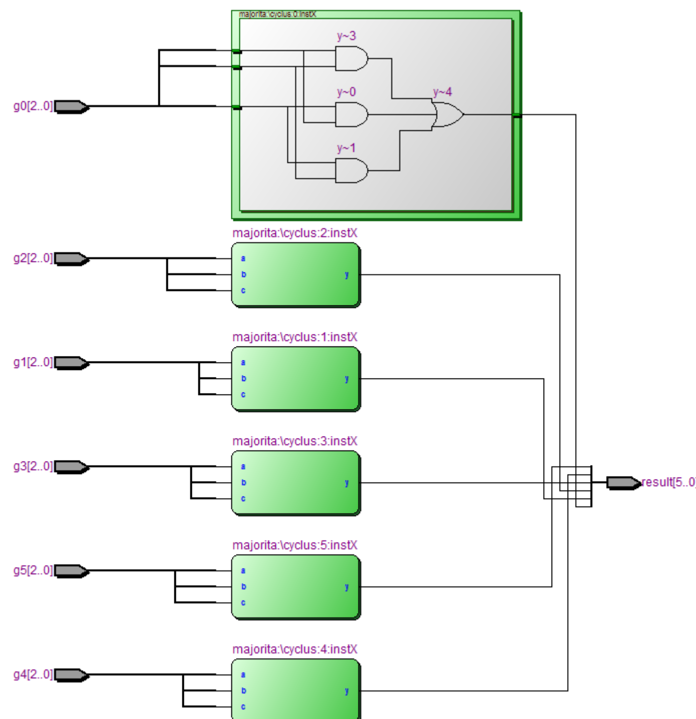
VHDL neobahuje možnost jak dynamicky generovat identifikátory.

Kdyby trojice čidel v sekci port byly v samostatných vstupech s odlišnými názvy, již nesestavíme cyklus tak lehce. Nemůžeme totiž složit identifikátor jako řetězec a ten následně použít. Některé jazyky to umí, třeba Matlab, ale VHDL ne. Ostatně ani jazyk C tohle nedovoluje, v něm se podobná potřeba často obchází vytvořením slovníku.

Ve VHDL lze podobný problém zase vyřešit inicializovaným polem:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity majorita6ga is
    port (  g0, g1, g2, g3, g4, g5 :  in  std_logic_vector(2 downto 0);
            result :  out  std_logic_vector(5 downto 0));
end;
architecture dataflow of majorita6ga is
component majorita is   port( a, b, c : in std_logic; y : out std_logic);
end component;
  type gv_t is array (0 to 5) of std_logic_vector(g0'RANGE);
  signal gv : gv_t := (g0, g1, g2, g3, g4, g5); -- initialized array
begin
    cyclus: for i in 0 to 5 generate
      instX : majorita port map(a=>gv(i)(0), b=>gv(i)(1), c=>gv(i)(2), y=>result(i));
    end generate;
end architecture;  --we have emphasized that this end belongs to architecture
```

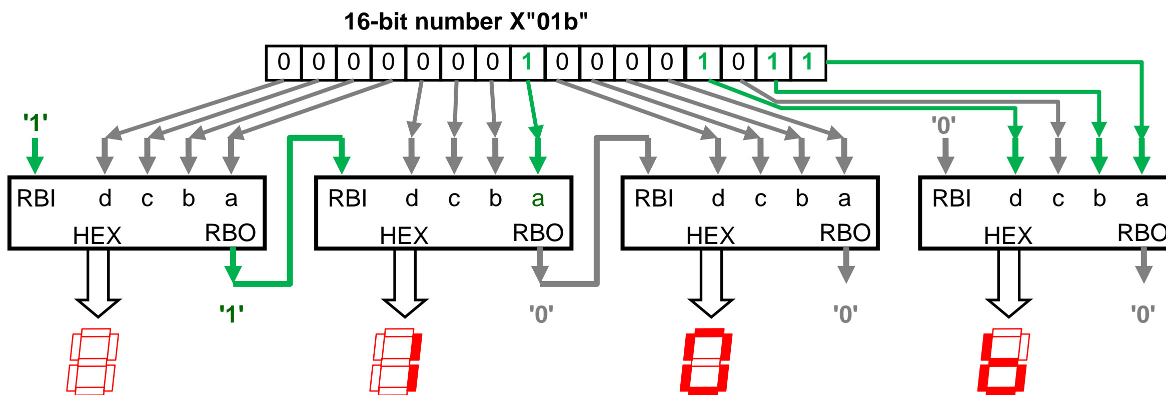
Výsledný kód má také velmi jasnou RTL Map, a to kvůli jednoznačnému popisu a indexům odvozeným od parametru cyklu.



Inicializované pole se někdy hodí i u jiných rozsáhlejších kódů, kde dovede zjednodušit komplikovanější konstrukce, v nichž lze zavést for generate.

6.8 *** Cvičná úloha 8: Zobrazení 16-bitového čísla na 7segmentovém displeji

Již jste si sestavili obvod 7segmentového displeje, který má vstup RBI a výstup RBO. Zkuste ho nyní využít, jako již naznačil Obrázek 9 na str. 45, jehož kopii vidíte dole.



Zadání: Napište ve VHDL obvod, který vezme binární číslo nastavené na přepínačích SW(15 downto 0) desky DE2 a zobrazí ho hexadecimálně na čtyřech 7segmentových displejích:

HEX0, HEX1, HEX2, HEX3 : std_logic_vector(6 downto 0);

kde HEX0 označuje displej nejvíce vpravo, tedy s nejnižší vahou, zatímco HEX3 má váhu nejvyšší.

* Obvod realizujte s potlačením úvodních 0.

* Využijte vytvoření instance od již existujícího obvodu 7segmentového dekodéru s RBI a RBO a definujte si signály propojovacích vodičů, které využijete v jednotlivých port map. Příkaz for generate se tady nevyplatí, protože instance mají odlišnou strukturu vstupů a výstupů.

* Překontrolujte si výsledek v RTL Map.

* Alternativně můžete vytvořit variantu, že potlačení úvodních nul se zapíná či vypíná dalších signálem. *Nápověda: Jedná se o velmi jednoduchou úpravu, jejíž realizace vyžaduje víc přemýšlení než psaní. Stačí opravdu jen nepatrná drobnost. Zkuste na ni přijít sami.*

V příloze na str. 67 si můžete svá řešení obou případů porovnat s možnými realizacemi.

6.9 *** Cvičná úloha 9: Prioritní inhibitor pro tři trojice

Ve cvičné úloze 5.3 na str. 37 se řešil univerzální prioritní inhibitor. Použijeme ho pro skupinovou volbu, v níž se žádá selekce jen jednoho signálu ze šesti.

Předpokládejme opět vstup z přepínačů SW(17 downto 0).

Zadání: Přepínače SW(5 downto 0) tvoří první skupinu, z níž se při vícenásobné volbě propustí na odpovídající výstup maximálně jen jeden vstup v '1', a to ten, který má nejvyšší prioritu. Analogicky se zpracují i zbývající skupiny SW(11 downto 6) a SW(17 downto 12) .

* Připojte výstup obvodu na červené led diody LEDR(17 downto 0).

* Využijte tentokrát již příkaz for generate, jímž třikrát vložíte instanci prioritního inhibitoru.

* Překontrolujte v RTL Map, zda se vytváří přehledné schéma.

* Můžete si zkusit i variantu, kdy se stejným způsobem zpracovává 6 trojic a z každé se propustí nejvýš jediný signál v '1', tedy SW(2 downto 0) tvoří první skupinu, SW(5 downto 3) druhou, atd. Bude úprava na trojice náročná? A dala by se změna celá provést za 10 sekund? Ano, při použití for generate. Zkuste si to.

V příloze na str. 69 najde možná řešení obou úloh.

7 @Příklad XIX. - Vytvoření vlastní knihovny ("package")

Vkládání deklarací komponent zvyšuje bezpečnost kódu, ale s jejich vzrůstajícím počtem klesá přehlednost kódu. V jazyce C se situace zjednodušila direktivou preprocesoru #include a v C# se zase používá příkaz using. V Java máme možnost odkázat příkazem import na vytvořený package.

Stejný koncept se aplikuje i ve VHDL, nazývá se package a představuje velkou výhodu oproti Verilogu, který něco podobného neumí. Balíčku se vytvoří vložením potřebných deklarací a definici.

Základní syntaxe package s hlavními prvky, které se v něm mohou objevit při syntéze, přičemž žluté zvýraznění označuje části, které již známe. Ostatní, až na alias, patří do VHDL behavioral stylu:

```
<library statement> <use clauses>
package  package_name  is
    <components>
    <constant full definitions with assigned values>
    <constants deferred - no assigned values>
    <type and subtype declarations>
    <aliases definitions of alternatives names for existing objects>
    <attribute declarations and specifications>
    <functions headers>
    <procedure headers>
end  [package]  [package_name];

package body  package_name  is
    <assigned values for deferred constants>
    <other declarations internally used inside body>
    <complete definitions of functions>
    <complete definitions of procedures>
end  [package body]  [package_name];
```

Balíček má dvě části, tedy něco podobného rozdělení obvodu na entity a architecture. Úvodní část

```
package package_name is
end [package]  [package_name];
```

deklaruje prvky viditelné zvnějšku. Její název package_name představuje námi přidělený VHDL identifikátor a koncové end lze napsat podobně jako u entity, viz Tabulka 3 na str. 12. Můžeme uvést jen end či za něj u delších bloků přidat ještě upřesnění, k čemu se vlastně vztahuje.

```
package body package_name is
end [package body] [package_name];
```

poté provede úplné definice a jeho koncové end má stejné možnosti.

Platí zde opět pravidlo **trojího pojmenování**. Pokud vytvoříme balíček a rozhodneme se mu přiřknout název lsp_codes, poté

1. soubor uložíme jako lsp_codes.vhd,
2. stejný název napíšeme do hlavičky balíčku: package lsp_codes is
3. a nezapomeneme shodně pojmenovat i jeho tělo:
   package body lsp_codes is

Ukážeme se definici balíčku, do něhož vložíme jedinou komponentu, aby se nám text i s následným příkladem importu balíčku vešel na jednu stránku:

```vhdl
-- Soubor lsp_codes.vhd
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;                    --L01
package lsp_codes is                                                                    --L02
   constant LEDR_LEN:integer:=18;                                                       --L03
   constant LEDG_LEN:integer;  -- deferred constant                                     --L04
   subtype slv_ledr  is  std_logic_vector(LEDR_LEN-1 downto 0);                         --L05
   component majorita is port( a, b, c : in std_logic; y : out std_logic);              --L06
   end component;                                                                       --L07
   alias LG : integer is LEDG_LEN;                                                      --L08
end; -- or  also e.g.  end package;                                                     --L09
package body lsp_codes is                                                               --L10
   constant LEDG_LEN:integer:=9; -- assigning value to deferred constant                --L11
end;  -- or  also e.g.  end package body;                                               --L12
-- Soubor demo_majorita6fg_package.vhd
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;              --R01
use work.lsp_codes.all;                                                                 --R02
entity demo_majorita6fg_package is                                                      --R03
   port ( SW :  in  slv_ledr; LEDG :  out  std_logic_vector(LG-1 downto 0));            --R04
end;                                                                                    --R05
architecture dataflow of demo_majorita6fg_package is                                    --R06
begin                                                                                   --R07
   cyclus:  for i in 0 to 5 generate                                                    --R08
            instX : majorita port map(a=>SW(3*i), b=>SW(3*i+1), c=>SW(3*i+2), y=>LEDG(i));  --R09
            end generate;                                                               --R10
   LEDG(LEDG_LEN-1 downto 6)<="100"; --equal to LEDG(LEDG'HIGH downto 6)<="100";        --R11
end architecture;  -- for clarity, we have emphasized that this end belongs to architecture  --R12
```

L03 a L04 - konstanty

V balíčku lze definovat konstantu buď jako úplnou včetně její inicializace, nebo provést pouze její deklaraci, což znamená uvedení názvu a typu, ale bez hodnoty. Druhý případ se nazývá odloženou konstantou (deferred constant). Používá se, když se její hodnota stanovuje složitějším postupem, jako například odvozením od jiných konstant nebo vytvořením inicializovaného pole, aby komplikovaný výraz s mnoha členy nesnižoval čitelnost hlavičky. Definice se odloží až na tělo balíčku.

Odložená konstanta se nesmí v hlavičce balíčku použít v deklaracích, u nichž je třeba znát její hodnotu, protože tu získá až v těle po své definici. Naší odloženou konstantu LEDG_LEN nemůžeme v hlavičce použít ani u subtypu a ani u komponenty, protože konstanta ještě nemá známou hodnotu.

Poznámka: VHDL není objektové programování, ve kterém definice uvnitř celé třídy jsou platné od jejího samého začátku. VHDL se podobá víc neobjektovému C, kde cokoli existuje až po provedení odpovídajícího příkazu. Odložená konstanta získá svou hodnotu až na řádku L11.

L05 - subtyp

Definovali jsme si subtyp slv_ledr a v něm jsme použili konstantu LEDR_LEN s úplnou definicí. Nový subtyp bychom pak mohli aplikovat i v dalších řádcích hlavičky balíčku, kdybychom to potřebovali.

Definice subtypů se často vkládají do balíčků, zejména u větších celků složených z dílčích obvodů. Využijí se ve všech dílčích entitách a případné hodnoty indexů, které se potřebují, se od nich odvodí VHDL atributy. Sestavujeme-li například procesor, navrhneme si subtyp pro jeho adresní datovou sběrnici. Chceme-li pak zvýšit/snížit její šířku, stačí změna na jednom místě ─ opravíme jen subtyp v balíčku.

L06 - component

Zde vkládáme jedinou komponentu, avšak klidně jsme jich mohli uvést víc, třeba od všech souborů, které jsme si vytvořili.

L08 - alias

Příkaz alias nevytváří nový objekt, ale dává pouze alternativní jméno již existujícímu objektu, kterým musí být pouze prvky constant, signal nebo proměnná variable (ta je však už prvkem behavioral stylu). Není tedy možná udělit jiné jméno například typu nebo komponentě. Přezdívku jsme tady přidělili odložené konstantě LEDG_LEN, což jsme mohli, jelikož příkaz alias nepoužívá její dosud nepřiřazenou hodnotu.

L11 - dodefinování odložené konstanty

Definice konstanty představuje jediný příkaz v těle balíčku, který zatím známe. Další již patří do behavioral stylu, kde jde hlavně o funkce a procedury.

R01 - knihovny v souboru používajícím balíček

V souboru, v němž chceme odkazovat na balíček, musíme vložit i všechny knihovny, které v něm budeme sami používat. Jejich direktivy napsané do balíčku se sem nepřenesou, s nimi pracoval jen balíček. Z něho se zavedou výhradně prvky uvedené v hlavičce balíčku, jinými slovy, jejich definice z balíčku se uloží do tabulek překladače coby páry složené z identifikátoru a jeho definice.

Úvodem poznámka: VHDL nepoužívá přímé analogii příkazů #include *jazyka C — ty zpracovává C preprocesor vložením jejich kompletního kódu. VHDL zachází s knihovnami analogicky jako Java s* import *příkazy nebo C# s* using *direktivami k importu z jiného namespace. Vkládá výsledek, ale ne dílčí součásti, které se uvnitř něho využily k tvorbě definic a deklarací.*

R02 - using package

Náš package lsp_codes vložíme stejně jako knihovny std_logic_1164 a numeric_std, i ty jsou ve své podstatě balíčky. Jelikož máme lsp_codes.vhd v adresáři Quartus projektu, upřesníme, že patří do work, viz popis jejího významu na str. 40. Napíšeme tedy use work.lsp_codes.all;

R04 - použití subtypů a konstant z balíčku

Jakmile máme balíček vložený v kódu, pak se během překladu zpracuje a odložené konstanty získají hodnotu. Můžeme definice v balíčku používat i v portech entity. Lze tak tvořit vysoce variabilní kód.

R09 - instance od komponent v balíčku

Nevložili jsme tentokrát žádnou definici komponenty, nemusíme, už je máme v balíčku. Rovnou vytváříme jejich instance již známým postupem komponent.

Samozřejmě musí vždy platit, že někde existují soubory, v nichž se definují architektury komponent, buď se nachází přímo mezi soubory projektu, nebo leží někde na cestě definované seznamem adresářů knihoven, který jsme zadali v Quartus menu: Tools->Options->General->Libraries: Project Libraries.

Poznámka: Kdybychom mezi řádky R06 a R07, tedy mezi architecture *a její* begin *vložili deklaraci komponentu* majorita*, nic by se nestalo, pokud by byla identická s balíčkem. Kdyby se však lišila, třeba i jménem vstupu nebo výstupu, nová definice by překryla původní a stala by se rozhodující pro náš kód.*

R11 - použití konstanty

Do kódu jsme přidali řádek zápisu na LEDG, abychom demonstrovali, že můžeme odkazovat jak na konstantu LEDG_LEN, tak i na její alias LG. Oba identifikátory referují k totožnému objektu.

7.2 @Příklad XX. - Distribuce - vše v jednom

Je-li vše v jednom souboru, vznikne nepřehledný kód, takže se od podobného stylu zrazuje. Hodí se však třeba pro distribuci finálního odladěného kódu. Quartus si celek přečte v jednom proudu a vytvoří všechny definice v něm uvedené. Příklad z předchozí kapitoly může uložit celý do jednoho souboru s obsahem:

```
------------------------------------------------------------------------------------
-- Following text was saved as  demo_majorita6fg_distribution.vhd
   -----------------------------------------------------------------------------------
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
package lsp_codesL is
   constant LEDR_LEN:integer:=18;
   constant LEDG_LEN:integer;  -- deferred constant

   subtype slv_ledr  is  std_logic_vector(LEDR_LEN-1 downto 0);

   component majoritaL is port( a, b, c : in std_logic; y : out std_logic); end component;
   alias LG : integer is LEDG_LEN;
end; -- or  also e.g.  end package;

package body lsp_codesL is
   constant LEDG_LEN:integer:=9; -- assigning value to deferred constant
end;  -- or  also e.g.  end package body;
------------------------------------------------------------------------------------
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
use work.lsp_codesL.all;
entity demo_majorita6fg_library is
   port ( SW :  in  slv_ledr; LEDG :  out  std_logic_vector(LG-1 downto 0));
end;
architecture dataflow of demo_majorita6fg_library is
begin
   cyclus: for i in 0 to 5 generate
           instX : majoritaL port map(a=>SW(3*i), b=>SW(3*i+1), c=>SW(3*i+2), y=>LEDG(i));
           end generate;
   LEDG(LEDG_LEN-1 downto 6)<="100"; --equal to LEDG(LEDG'HIGH downto 6)<="100";
end architecture;  -- for clarity, we have emphasized that this end belongs to architecture
------------------------------------------------------------------------------------
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity majoritaL is port ( a, b, c : in std_logic; y : out std_logic ); end;
architecture dataflow of majoritaL is
begin y <= (a AND b) OR (a AND c) OR (b AND c);
end;
------------------------------------------------------------------------------------
```

Entity jsme přejmenovali, aby se nám nepletly s již existujícími soubory v projektu. Dbali jsme na jejich správné pořadí, protože objekty musí již existovat před jejich použitím. Začali jsme balíčkem, jehož definice se uplatní v následujících částech. Blok majoritaL jsme klidně umístili až na konec, protože v balíčku se již vložila jeho hlavička, takže lze na něj odkazovat i před jeho zpracováním.

Chceme-li soubor přeložit, uděláme z něj Top-Level entity. Alternativně by se dal proměnit na balíček, který odněkud použijeme z jiné Top-Level entity, ale pak by se musel uložit pod jménem balíčku, tedy jako lsp_codesL.vhd, jinak by se nemusel najít.

Všimněte si opakovaného vkládání knihoven! S koncem bloku end (package body) nebo end (architecture) se zneplatní předchozí definice library a use, takže se musí zavést znovu. Právě tohle usnadňuje spojení více souborů. Dříve importované balíčky neovlivní další části, každá si provede své vlastní definice.

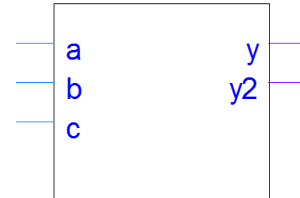
8 Příloha A: Řešení cvičných úloh

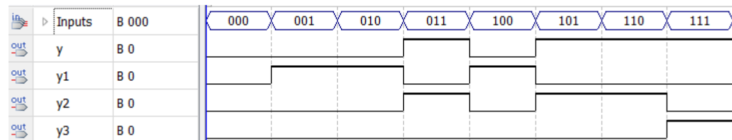
8.1 Cvičná úloha 1: Majorita123

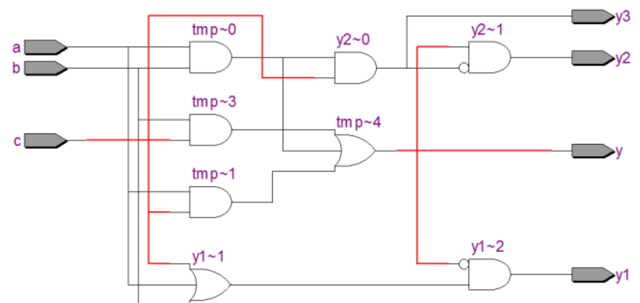
Možné řešení úlohy zadané na str. 10.

VHDL kód majorita123.vhd	**Schematická značka obvodu**

```
-- Majorita ze 3 s udaji o vstupech v 1
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity majorita123 is
        port (   a, b, c : in std_logic;
                 y, y1, y2, y3 : out std_logic );
end;
architecture dataflow of majorita123 is
signal tmp : std_logic;
begin
        tmp <= (a AND b) OR (a AND c) OR (b AND c);
        y<= tmp; y2 <= tmp AND NOT (a AND b AND c);
        y1<=(a OR b OR c) AND not tmp;
        y3<=a AND b AND c;
end;
```

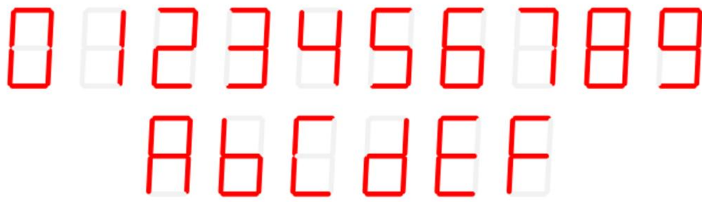


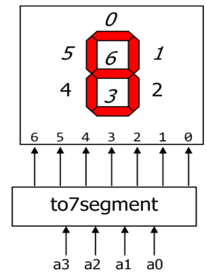




8.2 Cvičná úloha 2: Dekodér pro 7segmentový displej

Úlohu ze str. 27 můžeme řešit polem, do něhož uložíme tvary hexadecimálních číslic. Píšeme pro vývojovou desku DE2, kde segment svítí, když odpovídající výstup je v '0'.



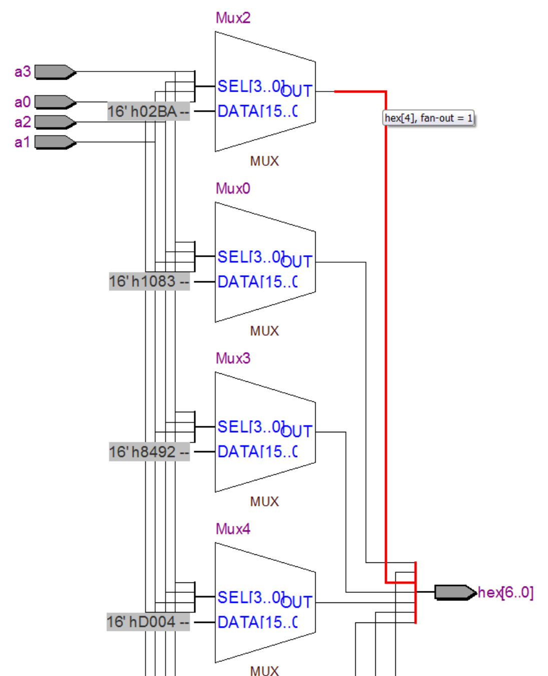


Vstupy a3, a2, a1 a a0 spojíme operátorem & spojení s nápovědou cílového typu a zadáme, že výsledný std_logic_vector chceme brát jako číslo unsigned. To převedeme na integer a indexujeme jím pole hexarray.

```
-- Display hexadecimal number on 7segment display
library ieee;use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity to7segment is
    port(   a0, a1, a2, a3 :in std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
            hex: out std_logic_vector(6 downto 0) ); -- hex[6..0] output to 7 segment display
end;
architecture dataflow of to7segment is
  type hexarray_t is array(0 to 15) of std_logic_vector(hex'RANGE);
  constant hexarray : hexarray_t :=
      ("1000000","1111001","0100100","0110000", "0011001","0010010","0000010","1111000",
       "0000000","0010000","0001000","0000011","1000110","0100001","0000110","0001110");
begin
      hex<= hexarray(to_integer(unsigned(std_logic_vector'(a3 & a2 & a1 & a0))));
end;
```

Poznámka: Pokud jste pole náhodou kódovali obráceně, tedy '1' jako svícení, nevadí, konverzi vyřeší operátor NOT ve výrazu, který lze použít i na vektor, tedy na výsledek vybraný indexem:

```
hex<= NOT hexarray(to_integer(unsigned(std_logic_vector'(a3 & a2 & a1 & a0))));
```



Část RTL Viewer schématu vidíte vpravo. Quartus konvertoval pole napřed na 16-vstupové multiplexory pro bity výstupu hex zapsaného zkráceně jako hex[6..0], tedy hex(6 downto 0).

Multiplexory mají vstupy adresy, které jsou označené jako SEL[3..0], tedy SEL(3 downto 0), na něž jsou připojené adresní vodiče a3, a2, a1 a a0.

Každý multiplexor má 16 vstupů DATA[15..0] a jejich hodnoty zapsané stručnými konstantami. Například 16'h02BA u multiplexoru Mux2, je 16 bitů X"02BA", tedy vektor "0000001010111010", který určuje svícení segmentu hex(4) u hexadecimální číslice F, E, d, C, b, A, 8, 6, 2 a 0.

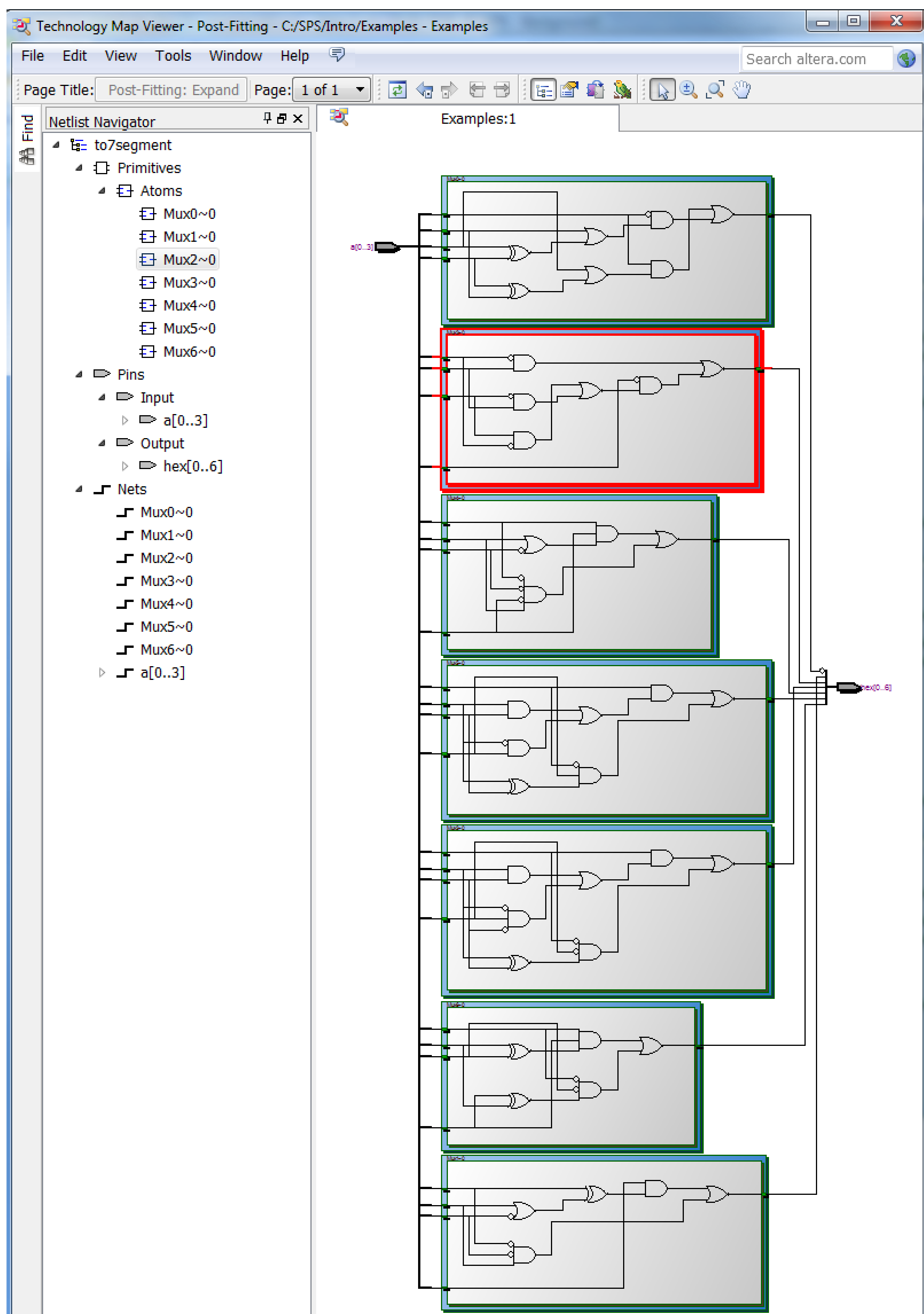
Skutečnost, že Mux2 vede na segment hex(4) zjistíme třeba po najetí myší na vodič. Jména multiplexorů Mux# byla automaticky vygenerovaná během překladu a obecně se nevážou k číslu výstupu.

Kontextová nápověda u výstupu Mux2 má tvar

"hex[4], fan-out = 1"

První údaj hex[4] označuje, kam vybraný vodič vede. Číslo za fan-out zase specifikuje, na kolik prvků je vodič celkem připojen, tedy zatížení výstupu. Zde jen jedním.

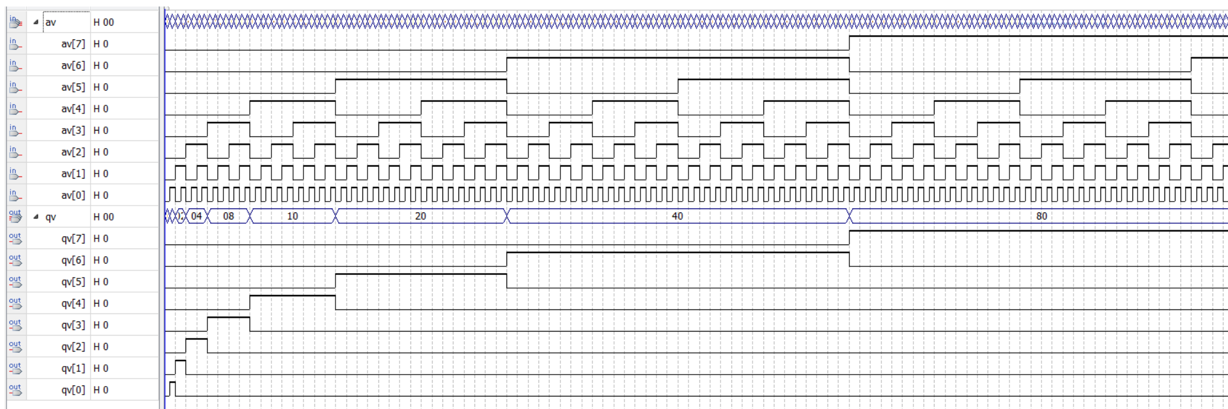
Vlastní implementaci nám ukáže Technology Map Viewer, kde si najdeme třeba zase Mux2. Vidíme, že se Quartus nesnažil o žádnou skupinovou minimalizaci. Neměl k ní důvod, protože jeden logický element v obvodu Cyclone II umí realizovat právě jednu 4-vstupovou logickou funkci, tedy přesně takovou, jakou potřebuje jeden segment.



8.3 Cvičná úloha 3: Prioritní inhibitor

Řešení úlohy ze str. 31 lze vytvořit modifikací příkladu z kapitoly 4 na str. 28.

```
--Priority inhibitor 8 inputs
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity prioritni_inhibitor8 is
        port (  av : in std_logic_vector(7 downto 0);
                qv : out std_logic_vector(7 downto 0));
end;
architecture dataflow of prioritni_inhibitor8 is
begin
qv <=  X"80" when av(7)='1' else
       X"40" when av(6)='1' else
       X"20" when av(5)='1' else
       X"10" when av(4)='1' else
       X"08" when av(3)='1' else
       X"04" when av(2)='1' else
       X"02" when av(1)='1' else
       X"01" when av(0)='1' else
       X"00";
end;
```



8.4 Cvičná úloha 4: Univerzální prioritní inhibitor

V úloze ze str. 37 můžeme opět využít naši šablonu.

Zadáváme-li proměnnou délku, musíme skládat výstup po jednotlivých signálech. Vytvoříme si pomocný signál inh (*inhibit*) s rozsahem shodným s výstupem qv, jímž budeme přenášet informaci o aktivaci některého prioritnějšího výstupu do '1'. Přijmeme-li pravidlo, že vyšší index ji vyšší prioritu, pak platí:

```
qv(qv'HIGH)<=av(qv'HIGH); inh(qv'HIGH)<=av(qv'HIGH);
```

Podřízený výstup vytvoříme v cyklem s parametrem i v rozsahu Q'HIGH-1 downto 0 výrazem

```
qv(i) <= NOT  inh(i+1) AND av(i);
```

tedy propustíme qv(i), není již prioritnější výstup v '1' a vstup je v '1'. Informaci o aktivaci výstupu akumulujeme v poli inh, jehož nižší členy jsou v '1', pokud byl nějaký vyšší signál v '1' nebo vstup v '1'.

```
inh(i) <= inh(i+1) OR av(i);
```

```
--Prioritni inhinitor
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity prioritni_inhibitor is
        generic( N: natural := 18);  -- delka vektoru > 1
        port (  av : in std_logic_vector(N-1 downto 0);
                qv : out std_logic_vector(N-1 downto 0) );
begin
        assert N>1 report "required N>1" severity failure;
end;
architecture dataflow of prioritni_inhibitor is
signal inh: std_logic_vector(qv'RANGE);
begin
        qv(qv'HIGH)<=av(av'HIGH); inh(qv'HIGH)<=av(av'HIGH);
        cyklus: for i in qv'HIGH-1 downto 0 generate
                qv(i)<=NOT inh(i+1) AND av(i); inh(i)<=inh(i+1) OR av(i);
        end generate;
end;
```

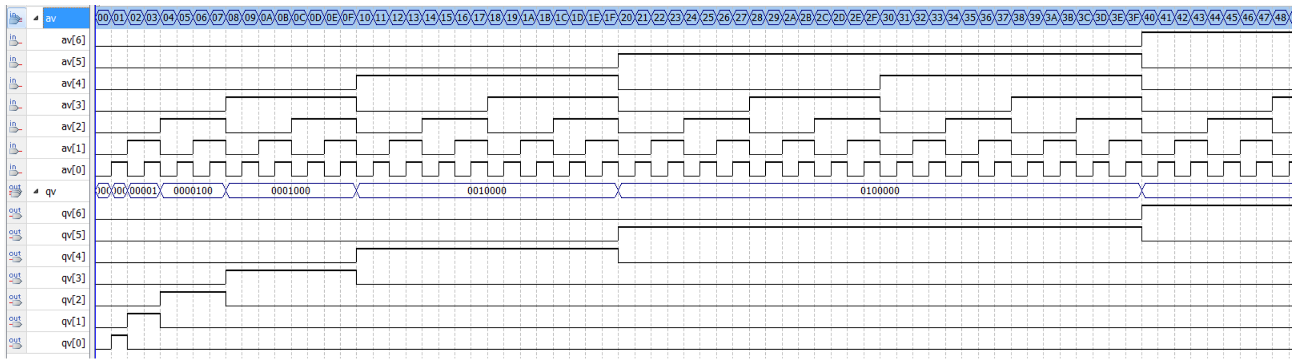
Zamyslete se ještě nad skutečností, zda by se změnil výsledek, kdyby se cyklus generace obrátil?

```
begin
         qv(qv'HIGH)<=av(av'HIGH); inh(qv'HIGH)<=av(av'HIGH);
        cyklus: for i in 0 to qv'HIGH-1 generate
                qv(i)<=NOT inh(i+1) AND av(i); inh(i)<=inh(i+1) OR av(i);
        end generate;
end;
```

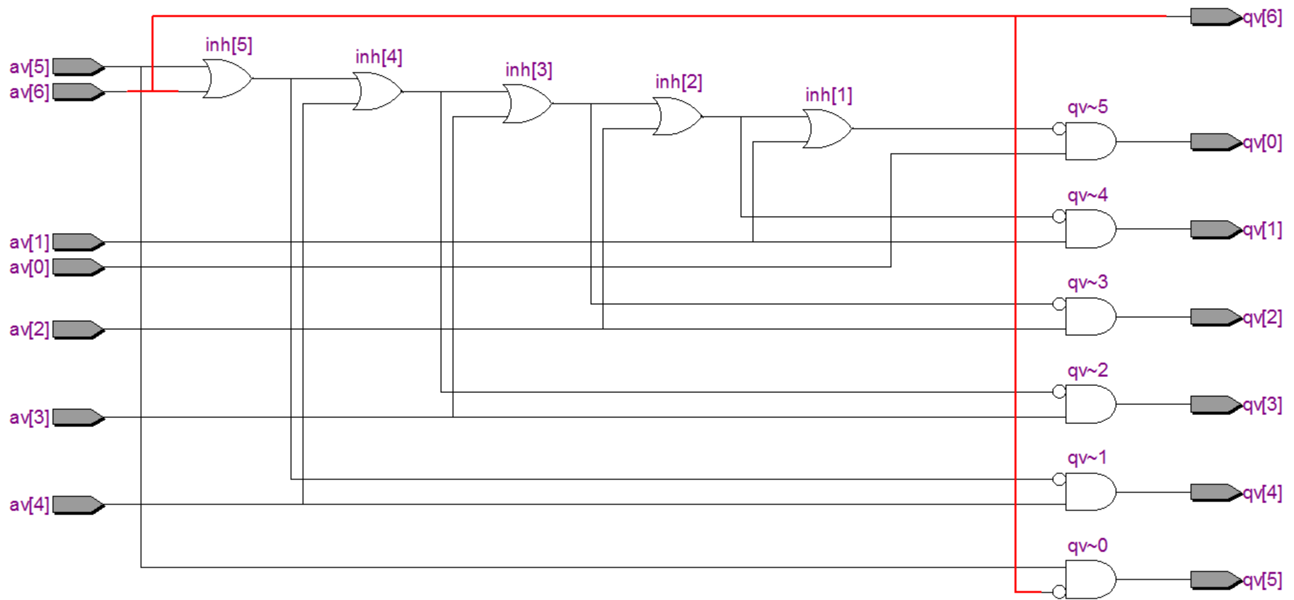
V klasickém sekvenčním programu by nepochybně vzniklo nefunkční řešení. Ale v souběžném kódu pouze propojujeme signály mezi sebou!

Vyzkoušejte si v simulátoru jak to tak downto cyklus generate. Kvůli simulaci zmenšete hodnotu generic parametru N třeba na 7, aby se výsledek dal ještě zobrazit. Jednoduchý Simulation Waveform Editor nedovolí totiž delší běh simulace než 100 mikrosekund, takže jím můžeme testovat nejvýše prioritní inhibitor maximálně s N=12. Delší časy lze zvládnout jedině pomocí testbench, které se budeme učit na přednáškách.

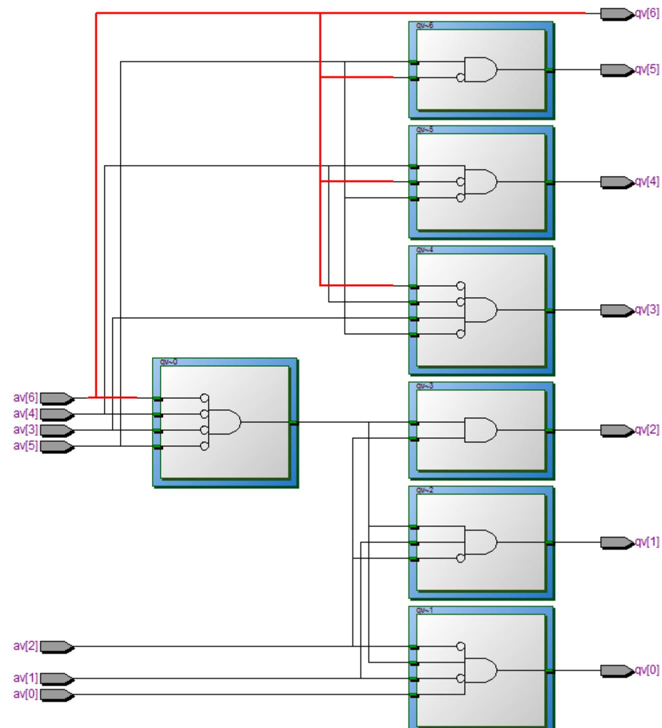
*Poznámka: Nezapomeňte na informaci z přílohy C, že doba běhu simulace se ve Simulation Waveform Editor musí to provést **před vkládáním jakýchkoli signálů**, tedy ihned po vytvoření souboru *.wvf. Mění z jeho menu: Edit->Set End Time... Pozdější úprava doby simulace může žel vést ke zhroucení Quartusu!*



Obrázek 13 - Část simulace pro N=7



Obrázek 14 - RTL Viewer schéma pro N=7

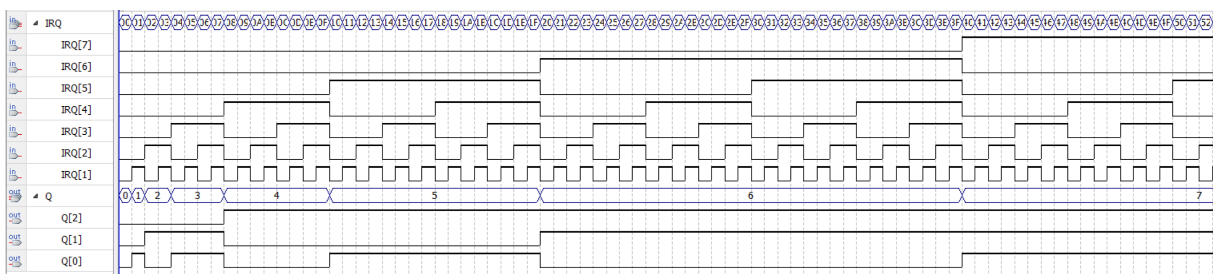


Obrázek 15 - Výsledná Technology Map pro N=7

8.5 Cvičná úloha 5: Univerzální prioritní kodér

Možné řešení úlohy ze str. 38 vidíte v následujícím kódu, v němž jsme zvýraznili doplněné části.

```
--Universal interrupt priority decoder
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity irq_priorityN is
    generic (   IRQ_MAX : integer:=15; --number of interrupt inputs
                Q_LENGTH : integer:=4); -- width of number output
    port (  -- interrupt request, higher index has higher priority
            IRQ : in std_logic_vector(IRQ_MAX downto 1);
            -- unsigned number of the highest active interrupt
            Q : out std_logic_vector(Q_LENGTH-1 downto 0) );
    begin
    assert IRQ_MAX>=2 report "Required IRQ_MAX >=2" severity failure;
    assert Q_LENGTH>=2 report "Required Q_LENGTH >= 2" severity failure;
    assert 2**Q_LENGTH > IRQ_MAX report "Q_LENGTH too low for encoding IRQ_MAX"
    severity warning;
end;
architecture dataflow of irq_priorityN is
type qtmp_array_t is array (IRQ'RANGE) of unsigned(Q'RANGE);
signal qtmp : qtmp_array_t;
begin
    qtmp (IRQ'LOW)<= (others=>'0') when IRQ(IRQ'LOW)='0'
                                        else to_unsigned(IRQ'LOW, Q_LENGTH);
    cyclus: for i in IRQ'LOW+1 to IRQ'HIGH generate
            qtmp (i) <= qtmp (i-1) when IRQ(i)='0' else to_unsigned(i, Q_LENGTH);
            end generate;
    Q <= std_logic_vector(qtmp(IRQ'HIGH));
end;
```



Obrázek 16 - Část simulace pro N=7

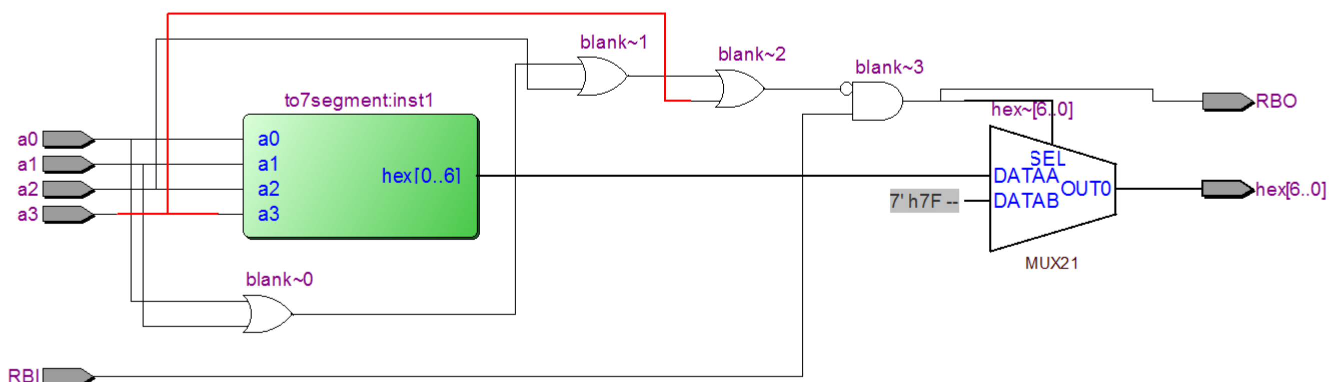
Poznámka: Obvod by se snadněji sestavil behavioral *stylem. Překladač by za nás provedl nutné definice pomocného pole* qtmp *a jeho použití v expanzích opakovaných zápisů na* concurrent *příkazy dovolující jediné přiřazení. Ukážeme si na některé přednášce. Výše uvedený kód nicméně odpovídá způsobu, jak by překlad mohl transformovat* behavioral *kód na* concurrent.

8.6 Cvičná úloha 6: 7segmentový dekodér s potlačením úvodních 0

Možné řešení úlohy ze str. 45 lze napsat třeba takto:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity to7segment_RBIO is
	port (	a0, a1, a2, a3 :  in  std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
		RBI : in std_logic; -- ripple blank in
		hex : out std_logic_vector(6 downto 0); -- hex[6..0] output to 7 segment display
		RBO: out std_logic); -- ripple blank out
end;

architecture dataflow of to7segment_RBIO is
component to7segment is
	port(	a0, a1, a2, a3 :in std_logic;
		hex: out std_logic_vector(6 downto 0) );
end component;

signal tmp : std_logic_vector(hex'RANGE);
signal blank : std_logic;
begin
	blank <= RBI AND NOT (a0 OR a1 OR a2 OR a3); -- RBI='1' and all inputs = '0'
	inst1 : to7segment port map(a0, a1, a2, a3, tmp);
	hex <= (others=>'1') when blank='1' else tmp; -- On DE2, hex led is ON on '0'
	RBO <= blank;
end;
```

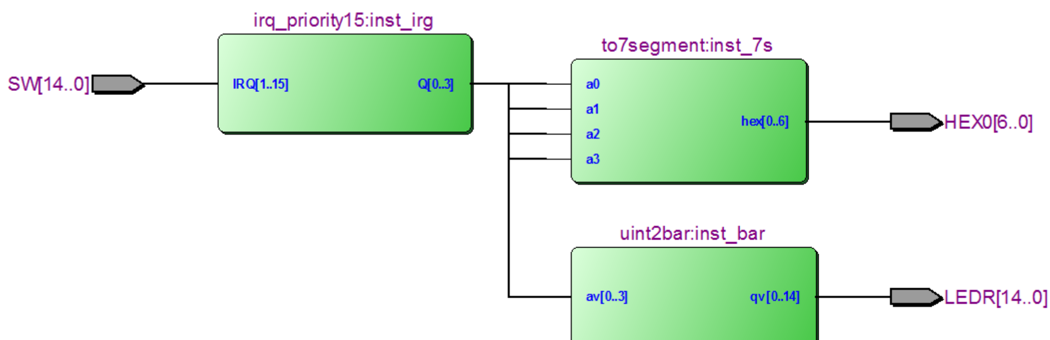
Quartus sestavil obvod z 7segmentového dekodéru, za který připojil multiplexor, jímž se potlačí svícení, je-li vstup 0 a RBI='1'.



8.7 Cvičná úloha 7: Přidání 7segmentového displeje

Úlohu ze str. 47 vyřešíme tak, že vložíme komponentu to7segment do kódu. Vytvoříme od ní instanci, a tu napojíme na již existující signál Q2av a výstup HEX0, který přidáme do sekce port. Název identifikátoru odpovídá pravé číslici na vývojové desce DE2. Bude-li demo_irq_barh jako Top-Level entity, pak se na něj provede i propojení.

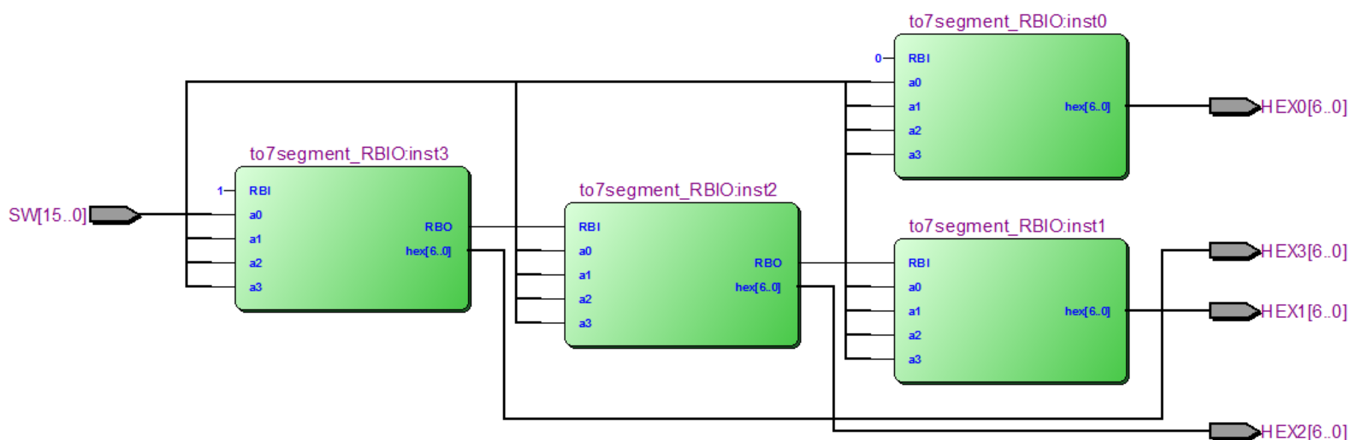
```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_irq_barh is  port ( SW :  in  std_logic_vector(14 downto 0);
                                LEDR :  out  std_logic_vector(14 downto 0);
                                HEX0: out std_logic_vector(6 downto 0));
end;
architecture dataflow of demo_irq_barh is
  component uint2bar
      generic ( AV_LENGTH : integer:=4; QV_LENGTH : integer:=15);
      port(   av : in std_logic_vector(AV_LENGTH-1 downto 0);
              qv : out std_logic_vector(QV_LENGTH-1 downto 0));
  end component;
  component irq_priority15 is
      port (  IRQ : in std_logic_vector(15 downto 1);
              Q : out std_logic_vector(3 downto 0) );
  end component;

component to7segment is
      port(   a0, a1, a2, a3 :in std_logic;
              hex: out std_logic_vector(6 downto 0) );
end component;
  signal Q2av :  std_logic_vector(3 downto 0);
begin
  inst_bar : uint2bar generic map(Q2av'LENGTH, LEDR'LENGTH) port map( Q2av, LEDR );
  inst_irg : irq_priority15 port map( IRQ=>SW, Q=>Q2av );
  inst_7s: to7segment port map(   a0=>Q2av(0),a1=>Q2av(1), a2=>Q2av(2), a3=>Q2av(3),
                                  hex=>HEX0);
end architecture;
```



8.8 Cvičná úloha 8: Zobrazení 16-bitového čísla na 7segmentovém displeji

V úloze ze str. 53 se cyklus for generate zde nevyplatí. Možné řešení lze sestavit pouhým kopírováním vytváření instance a následnou edicí. Použili jsme přímo vstupy z přepínačů a výstup na HEX. Kvůli propojení podle obrázku v zadání si vytvoříme dva pomocné signály.

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_uint16toHex is
port (  SW :  in  std_logic_vector(15 downto 0);
        HEX0, HEX1, HEX2, HEX3: out std_logic_vector(6 downto 0));
end;
architecture dataflow of demo_uint16toHex is
  component to7segment_RBIO is
      port (  a0, a1, a2, a3 :  in  std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
              RBI : in std_logic; -- ripple blank in
              hex : out std_logic_vector(6 downto 0); -- hex[6..0] output to 7 segment display
              RBO: out std_logic); -- ripple blank out
  end component;
signal s3tos2, s2tos1 : std_logic;
begin
      inst3: to7segment_RBIO
              port map(  a0=>SW(12),a1=>SW(13), a2=>SW(14), a3=>SW(15),
                         RBI=>'1', hex=>HEX3, RBO=>s3tos2);
      inst2: to7segment_RBIO
              port map(  a0=>SW(8),a1=>SW(9), a2=>SW(10), a3=>SW(11),
                         RBI=>s3tos2, hex=>HEX2, RBO=>s2tos1);
      inst1: to7segment_RBIO
              port map(  a0=>SW(4),a1=>SW(5), a2=>SW(6), a3=>SW(7),
                         RBI=>s2tos1, hex=>HEX1);
      inst0: to7segment_RBIO
              port map(  a0=>SW(0),a1=>SW(1), a2=>SW(2), a3=>SW(3),
                         RBI=>'0', hex=>HEX0);
end architecture;
```



Pokud chceme provést úpravu, která dovolí zapínání a vypínání potlačení úvodních nul, pak se opravdu jedná o směšně primitivní změnu. Potlačení úvodních nul se totiž zcela vypne, pokud se na RBI nejvyšší

číslice přivedeme '0' místo původní '1', tedy navolíme, že se číslice má zobrazit. Poté se vždy ukážou i všechny nižší.

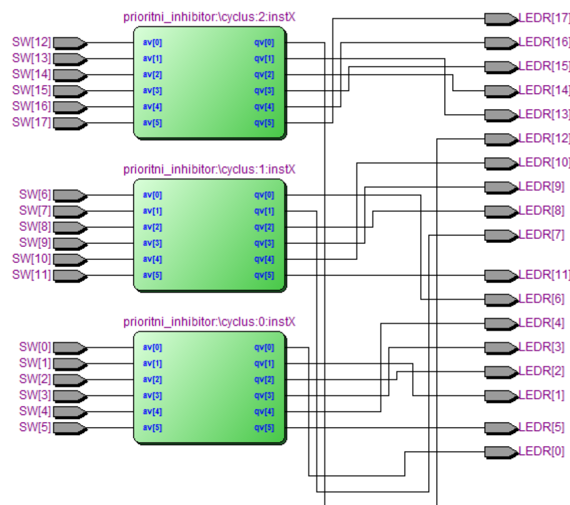
Můžeme si tedy rozšířit vstup SW a SW[16] zapojit na RBI inst3. Úpravy kódu mohou vypadat třeba takto:

```
library ieee, work; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity demo_uint16toHexR is
port (  SW :  in  std_logic_vector(16 downto 0);
        HEX0, HEX1, HEX2, HEX3: out std_logic_vector(6 downto 0));
end;

architecture dataflow of demo_uint16toHexR is
  component to7segment_RBIO is
      port (  a0, a1, a2, a3 :  in  std_logic; -- a3.. a0 input hexadecimal number a0-lsb  a3-msb
              RBI : in std_logic; -- ripple blank in
              hex : out std_logic_vector(6 downto 0); -- hex[6..0] output to 7 segment display
              RBO: out std_logic); -- ripple blank out
  end component;

signal s3tos2, s2tos1 : std_logic;

begin
      inst3: to7segment_RBIO
              port map(  a0=>SW(12),a1=>SW(13), a2=>SW(14), a3=>SW(15),
                         RBI=>SW(16), hex=>HEX3, RBO=>s3tos2);
      inst2: to7segment_RBIO
              port map(  a0=>SW(8),a1=>SW(9), a2=>SW(10), a3=>SW(11),
                         RBI=>s3tos2, hex=>HEX2, RBO=>s2tos1);
      inst1: to7segment_RBIO
              port map(  a0=>SW(4),a1=>SW(5), a2=>SW(6), a3=>SW(7),
                         RBI=>s2tos1, hex=>HEX1);
      inst0: to7segment_RBIO
              port map(  a0=>SW(0),a1=>SW(1), a2=>SW(2), a3=>SW(3),
                         RBI=>'0', hex=>HEX0);
end architecture;
```

8.9 Cvičná úloha 9: Prioritní inhibitor pro tři šestice

V úloze ze str. 53 již využijeme s výhodou for generate, jímž se dá napsat třeba takto:

```
--Priority inhibitor for 3 groups with 6 members
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity demo_priority_inhibitor3x6 is
   port (  SW :  in  std_logic_vector(17 downto 0);
           LEDR : out std_logic_vector(17 downto 0) );
end;

architecture dataflow of demo_priority_inhibitor3x6 is

component prioritni_inhibitor is
   generic( N: natural := 7);  -- delka vektoru > 1
   port (  av : in std_logic_vector(N-1 downto 0);
           qv : out std_logic_vector(N-1 downto 0) );
end component;

begin
     cyclus:  for i in 0 to 2 generate
          instX : prioritni_inhibitor
                 generic map(N=>6)
                 port map(av=>SW(6*i+5 downto 6*i), qv=>LEDR(6*i+5 downto 6*i));
              end generate;
end architecture;
```



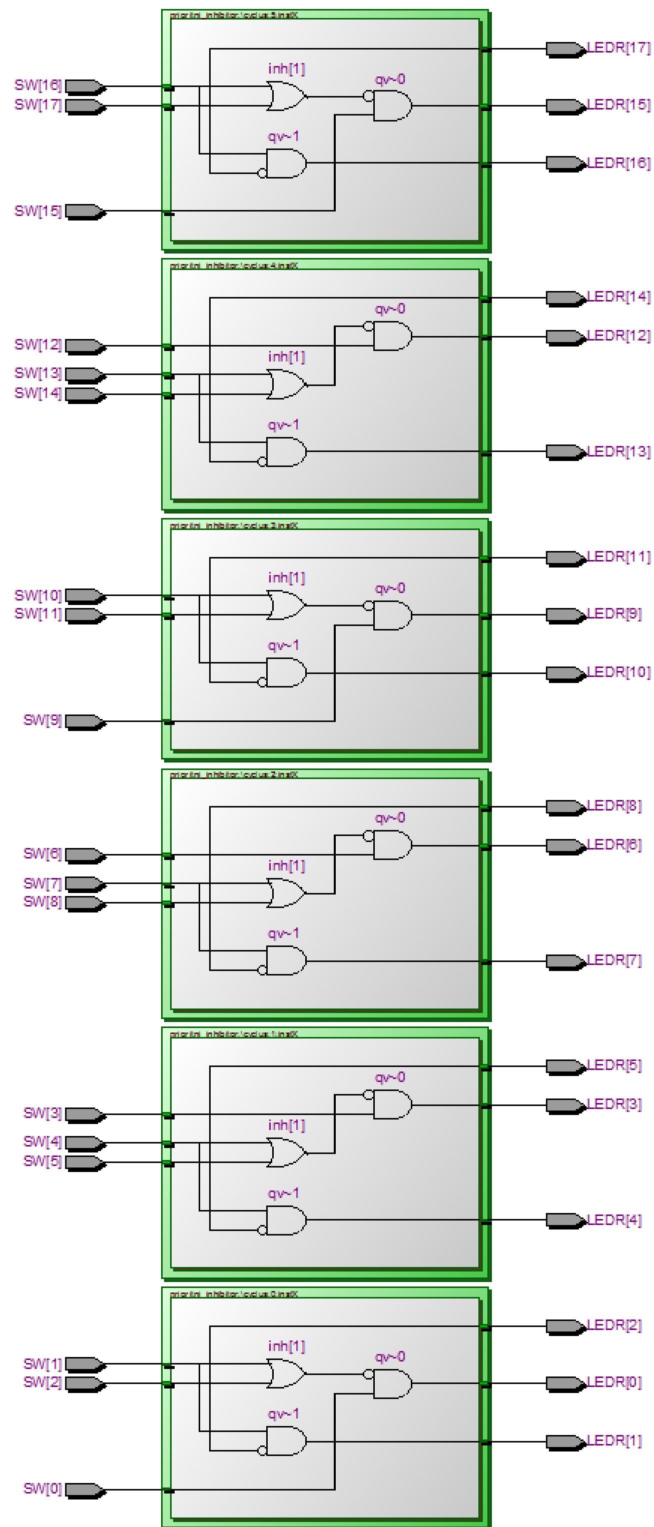
Obrázek 17 - Prioritní inhibitor pro 3 šestice

Nepoužili jsme záměrně žádnou pomocnou definici, ale napsali opakovaně 6*i. Důvod byl podrobně rozebraný na str. 49 v diskuzi o nevhodných VHDL kódech.

Změna na 6 trojic se provede za několik vteřin. Stačí totiž v cyklu přepsat čísla 2 na 5, 6 na 3 a 5 na 2.

```
...
begin --Architecture block of priority inhibitor for 6 groups with 3 members
     cyclus:  for i in 0 to 5 generate
          instX : prioritni_inhibitor
                 generic map(N=>3)
                 port map(av=>SW(3*i+2 downto 3*i), qv=>LEDR(3*i+2 downto 3*i));
              end generate;
end architecture;
```

Blokové schéma by se rozhodně tak svižně nepřekreslilo ☺



Obrázek 18 - Prioritní inhibitor pro 6 trojic

9 Příloha B: Definice operací s std_logic - Resolution tables

Knihovna ieee.std_logic_1164 definuje typ std_logic ve dvou krocích. Napřed zavede 9-hodnotovou logiku std_ulogic, s výčtovými členy nám již dobře známými:

```
type std_ulogic is ( 'U', -- Uninitialized
                     'X', -- Forcing  Unknown
                     '0', -- Forcing  0
                     '1', -- Forcing  1
                     'Z', -- High Impedance
                     'W', -- Weak Unknown
                     'L', -- Weak  0
                     'H', -- Weak 1
                     '-'  -- Don't care );
```

Tady ulogic znamená unresolved — není u ní známo, jaká bude výsledná hodnota při spojení více výstupů do jednoho bodu, což se vyskytuje zejména na datových sběrnicích. Z std_ulogic se poté vytvoří

```
subtype std_logic is resolved std_ulogic;
```

kde podobná situace se již řeší speciální funkcí resolved definovanou v knihovně std_logic_1164 pomocí tabulky:

propojení	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Tabulka 10 - Tabulka resolved typu std_logic

Poznámka: Na FPGA obvodu Cyclone II, který máme v deskách DE2, se může víc výstupů spojit jedině tehdy, když mají stejnou konstantní hodnotu, anebo nejvýše jeden z nich je ve stavu '0' nebo '1' a ostatní jsou ve vysoké impedanci 'Z'.

FPGA Cyclone II neobsahuje logiku, která zvládala jiné situace, protože v něm nejsou obvody s otevřenými kolektory (stavy 'L', 'W' a 'H'), s nimiž lze provádět vícenásobná spojení (ta se u nich nazývají termínem Wired-OR, viz https://en.wikipedia.org/wiki/Wired_logic_connection*).*

Univerzální knihovna řeší ale všechny případy.

Výsledky logických operací s std_ulogic, a tedy i s jeho subtypem std_logic, se definují tabulkami uvedenými na následující stránce.

AND	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'0'	'U'	'U'	'U'	'0'	'U'	'U'
'X'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'1'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'Z'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'W'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
'L'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
'H'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'-'	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'

OR	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'1'	'U'	'U'	'U'	'1'	'U'
'X'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'Z'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'W'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'	'1'
'-'	'U'	'X'	'X'	'1'	'X'	'X'	'X'	'1'	'X'

NOT	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'

XOR	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'1'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'
'Z'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'W'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'L'	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
'H'	'U'	'X'	'1'	'0'	'X'	'X'	'1'	'0'	'X'
'-'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'

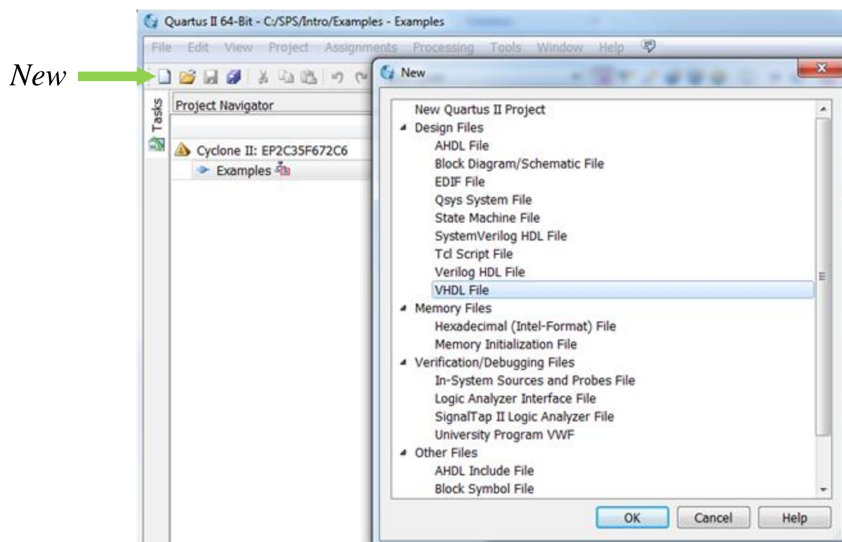
Tabulka 11 - Logické operace s typem std_logic

~o~

10 Příloha C: Vytvoření a překlad VHDL souboru majorita.vhd

Napřed musíme vytvořit Quartus projekt. Buď postupujeme podle návodu, který se nachází na DCENET: http://dcenet.felk.cvut.cz/edu/fpga/doc/VytvoreniProjektuProQuartus.pdf , nebo využijeme možnosti okopírovat si výchozí projekt, Quartus menu Project->Copy Project.

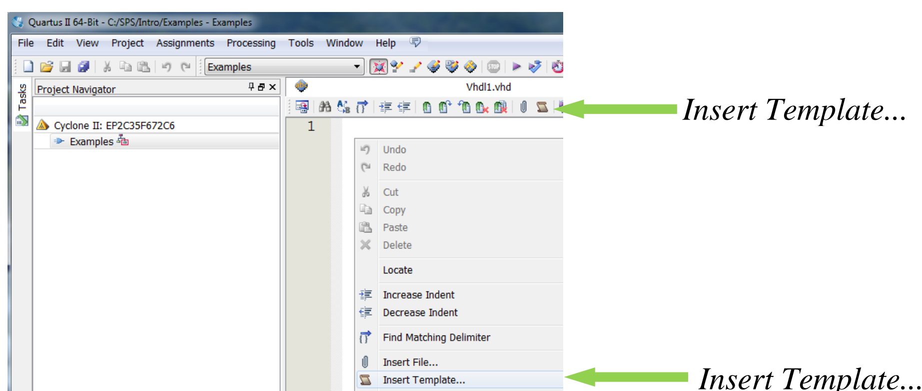
Nový projekt pojmenujeme ho třeba Examples a vložíme si do něho nový VHDL soubor, což lze učinit buď z Quartus menu volbou File->New... nebo kliknutím na ikonu New na ovládací liště. V dialogu zvolíme možnost VHDL File a dáme [Ok].



Do nově vzniklého souboru nakopírujeme naši šablonu z kapitoly 2.6 "VHDL šablona" na str. 11.

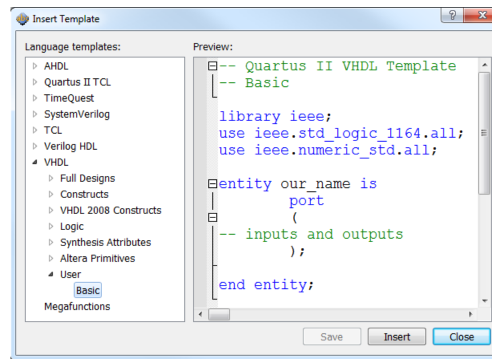
Alternativně si nastavíme uživatelský adresář šablon v menu Quartusu Tools->Options->Text Editor. Vytvoříme/zadáme User template library directory **odlišný** od adresáře předinstalovaných šablon v C:\altera. *Pozor, jde o globální nastavení, které mohou v učebně změnit i jiní. U počítače v laboratoři si ho tedy musíte pokaždé zkorigovat.*

Poté vyvoláme dialog šablon buď kliknutí na ikonu "Insert Template" nebo přes kontextové menu okna VHDL editoru.



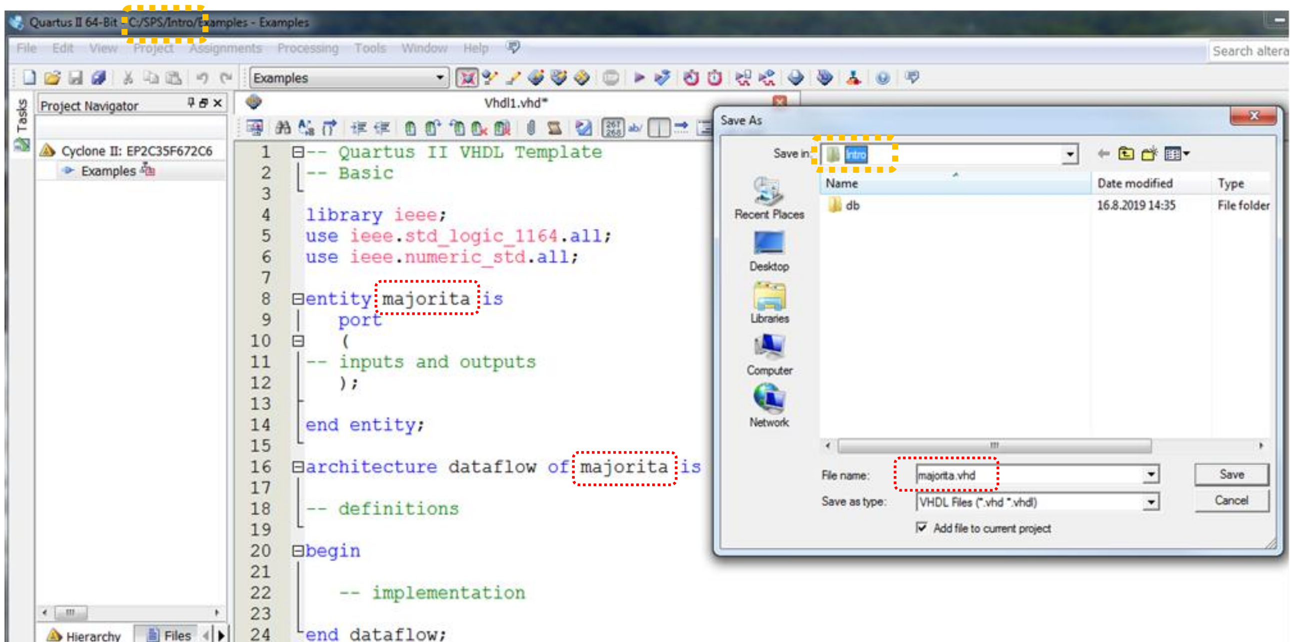
V dialogu šablon si vytvoříme novou šablonu přes kontextové menu na VHDL v části Language Template. Přejmenujeme ji třeba na Basic (dvojklik na automaticky vytvořený název), do Preview části vložíme/napíšeme text šablony a dáme Save. Vkládání je pak již snadné i do budoucna:

1. V okně VHDL editoru vyvoláme dialog Insert Template.
2. V dialogu zvolíme naši šablonu v sekci VHDL -> User -> Basic
3. Stisknutím Insert se šablona vloží na pozici kurzoru v okně VHDL editoru.
4. Zavřeme dialog šablon volbou OK.

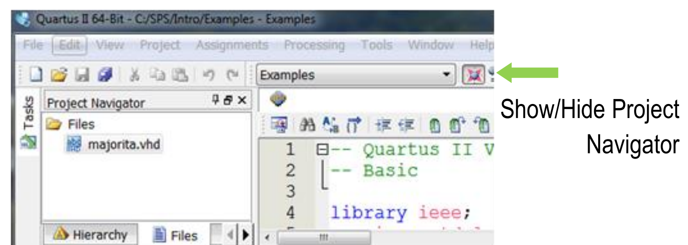


10.1 Uložení VHDL souboru

Ihned provedeme **trojí pojmenování** nově vzniklého souboru. Nazveme jeho entitu majorita, stejný název vložíme do architektury a soubor uložíme jako majorita.vhd volbou File->Save As... Při ukládání pečlivě kontrolujeme, zda **zapisujeme do adresáře našeho projektu**. Linuxová aplikace Quartus běžící pod Cygwin občas převezme od Windows jejich jiný výchozí adresář. Musíme si na to dávat pozor.



Přepneme "Projekt Navigator" na jeho záložku Files a ověříme si, zda se jméno našeho souboru vypisuje bez / úvodních podadresářů, což znamená, že se nachází v hlavním adresáři v našeho projektu.



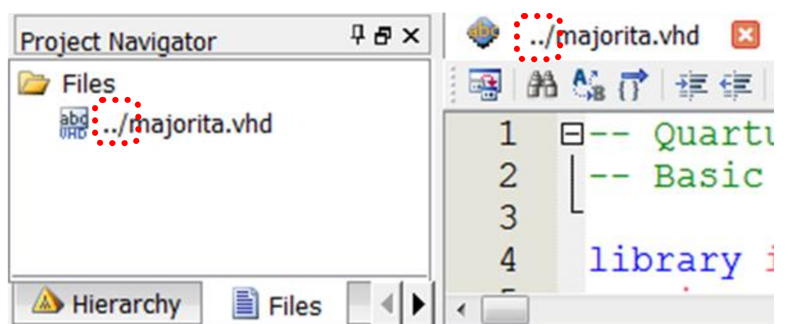
Vidíme-li / před jménem souboru, pak jsme omylem uložili do jiného adresáře.

Využijeme toho, že Quartus nezamyká přístup k souboru na disku, ale má jeho obsah celý ve své interní paměti.

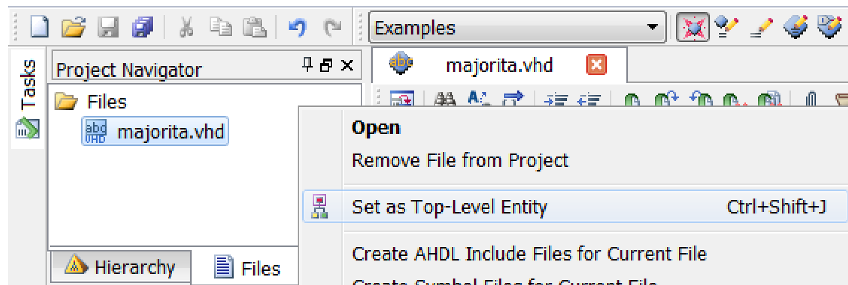
V okně "Projekt Navigator" na záložce Files vymažeme klávesou [Delete] soubor ze seznamu souborů našeho projektu, případně ho také zrušíme i na disku.

Poté soubor otevřený v editoru znovu uložíme do správného adresáře.

File majorita.vhd is not stored in the project root directory!



Máme-li soubor správně uložený, vybereme ho v Projekt Navigator - Files a v jeho kontextové menu zadáme "Set as To-Level Entity", čím se majorita.vhd stane hlavní entitou.



Soubor zvolený za Top-Level entity se ukazuje v "Project Navigator"na jeho záložce Hierarchy.

> *Poznámka: Volba Top-Level entity určuje, co sestavujeme. Máme za Top-Level entity označený jiný obvod, než si myslíme, pak vytváříme ten, a ne náš požadovaný. Nahráváme výsledek do vývojové desky a divíme se, že nám zatrolený návrh, přes veškeré úpravy, stále nechodí a nechodí. V klasickém programování tomu odpovídá analogická situace, že omylem spouštíme předchozí verzi programu, a ne novou sestavenou překladačem.*

Doplníme vloženou šablonu na kód majority:

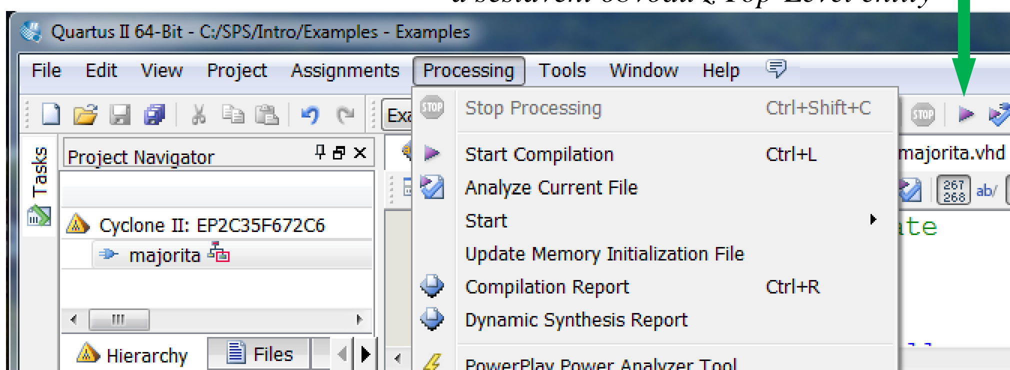
```
-- Majorita 2 of 3
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity  majorita is
        port (  a, b, c : in std_logic;
                y : out std_logic );
end;
architecture dataflow of majorita is
begin
        y <= (a AND b) OR (a AND c) OR (b AND c);
end;
```

10.2 Překlad souboru

Quartus si při kompilaci překontroluje vždy syntaktickou správnost všech souborů, které má uvedené v projektu, ale jde pouze o úvodní fázi překladu. Po ní následuje sestavování obvodu podle definic v souboru označeném jako Top-Level entity.

Můžeme tedy vyvolat jen první fázi, relativně rychlou, hodící se ke kontrole chyb. V závěru volíme úplný překlad.

Kontrola syntaxe všech souborů projektu a sestavení obvodu z Top-Level entity



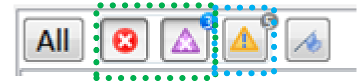
*Kontrola syntaxe všech souborů a analýza *.vhd zobrazeného v editoru.*

Obvod se nesestavuje.

10.3 Chyby a varování při překladu

Quartus vypisuje obrovské množství hlášek do okna "Messages" [View->Utility Windows -> Messages]. **Proč tohle dělá?** V průmyslové praxi se často spouští jako externí nástroj, který kooperuje s aplikacemi vývojových firem. Poskytuje jim zprávy, aby si je mohly analyzovat a získat důležitá fakta.

Pokud ho používáme oknovém módu, můžeme si zprávy seřadit podle jejich ID, filtrovat jejich texty, nebo je rychle vytřídit podle jejich kategorií. Tou nejzávažnější jsou Error a po nich Critical Warnings - ty mějte vždy vybrané tlačítky v okně Message, na obrázku orámovanými zeleně.



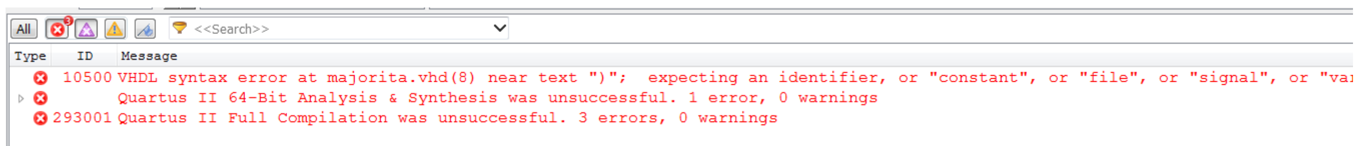
Všimněte si, že máme Critical Warnings a Warnings. Kritická varování odpovídají svou důležitostí varováním překladače jazyka C. Naproti tomu výpisy označené pouze jako Warnings můžeme většinou přehlížet, protože jde pouze o méně důležitá hlášení.

Poslední kategorii, Information, můžeme většinou opomíjet, ta většinou slouží coby pramen podkladů pro externí nástroje. Důležitá je jen většinou tu poslední informace - tou bývá, že vše se úspěšně přeložilo.

Hledání chyb si ukážeme na příkladu. Předpokládejme, že jsme v kódu majority udělali častou chybu - napsali jsme v sekci port středník navíc

```
port (  a, b, c : in std_logic;
             y : out std_logic; );
```

Po překladu uvidíme v okně zpráv (to se zobrazí volbou z menu View->Utility Windows->Messages):



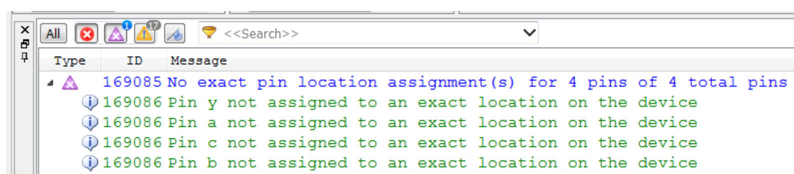
První řádek

Error (10500): VHDL syntax error at majorita.vhd(8) near text ")"; expecting an identifier, or "constant", or "file", or "signal", or "variable"

označuje chybu, ostatní jsou už její důsledky. Najdeme jméno našeho souboru *.vhd, za nímž se v závorce uvádí číslo řádku, na němž se detekovala chyba. Zpravidla stačí provést dvojklik myší na hlášení a editor se nastaví na vadný řádek. Pouze někdy musíme řádek vyhledat ručně.

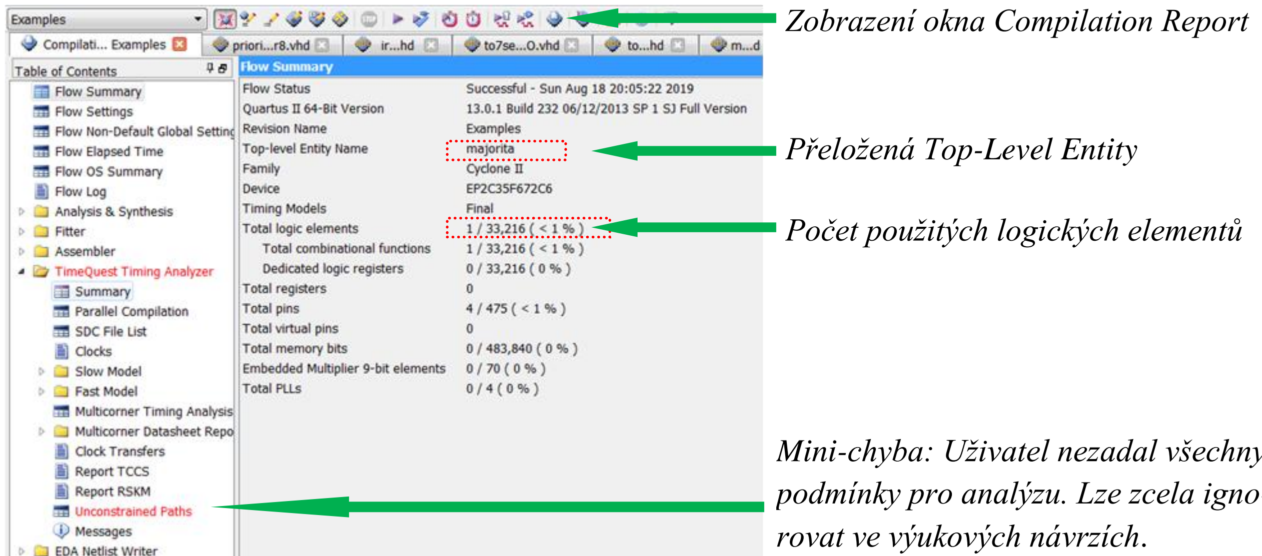
Chyba nemusí nutně být na uvedeném řádku, ale může se vyskytovat i někde před ním, viz debata na straně 9. Vypátráme její příčinu, opravíme ji a kód znovu přeložíme. Quartus vždy před překladem uloží všechny soubory, takže tím máme i jistotu, že se vše nachází na disku.

Po úspěšném překladu majority uvidíme kritické varování:



Quartus sděluje, že vstupy a výstupy v Top-Level entity souboru nevedou na vstupy a výstupy FPGA obvodu. Pokud bychom nahrávali obvod do vývojové desky, pak se jedná o velmi závažnou chybu. Přejeme-li si jen ho simulovat, pak nám nezapojené vstupy nevadí.

Důležité je také okno "Compilation Report:



Vidíme v něm, co jsme vlastně přeložili, a kolik logických elementů se spotřebovalo. Určitě by se měl ukazovat nejméně 1, je-li tam 0, je něco hodně špatně.

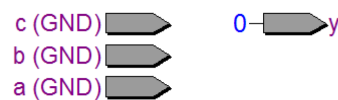
Napíšeme-li například třeba chybnou rovnici majority jako:

```
y <= (a AND b) XOR (a AND b);
```

pak se sice vše bez chyby přeloží, ale uvidíme:

Top-level Entity Name	majorita
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	0 / 33,216 (0 %)
Total combinational functions	0 / 33,216 (0 %)
Dedicated logic registers	0 / 33,216 (0 %)

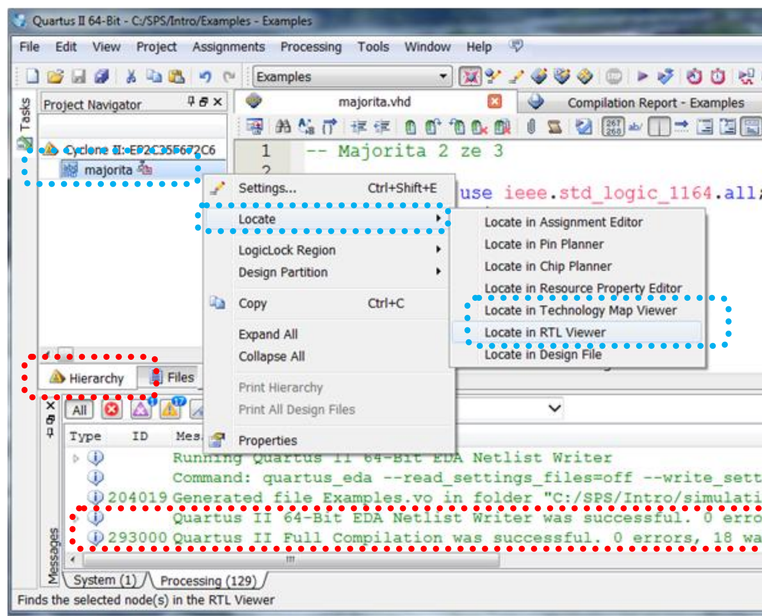
Rovnice se totiž minimalizovala na logickou kontradikci a výsledkem překladu je připojení výstupu y na '0'



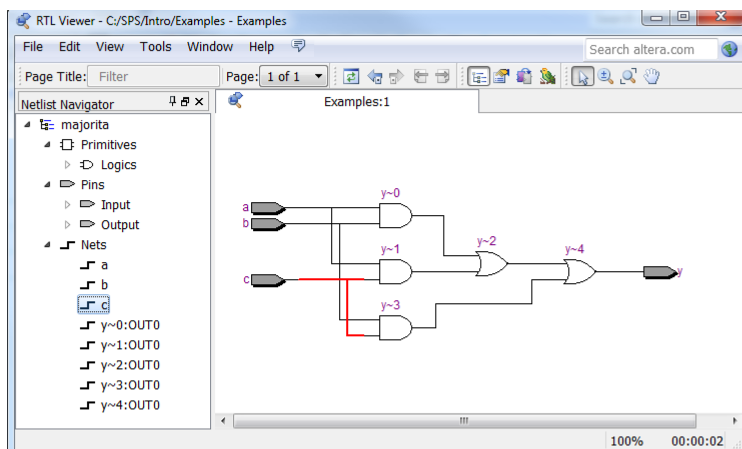
Podrobný rozbor některých chyb podle jejich čísel najdete v textu Vybraná chybová hlášení překladače Quartus, který se nachází na stránce http://dcenet.felk.cvut.cz/edu/fpga/navody.aspx

10.4 Zobrazení RTL Map a Technology Map

Obé lze sice zobrazit i po částečném rychlo překladu, ale lepší je zadat úplný překlad. Přepneme okno Project Navigator na záložku Hierarchy a pravou myší vyvoláme kontextové menu souboru Top-Level entity.



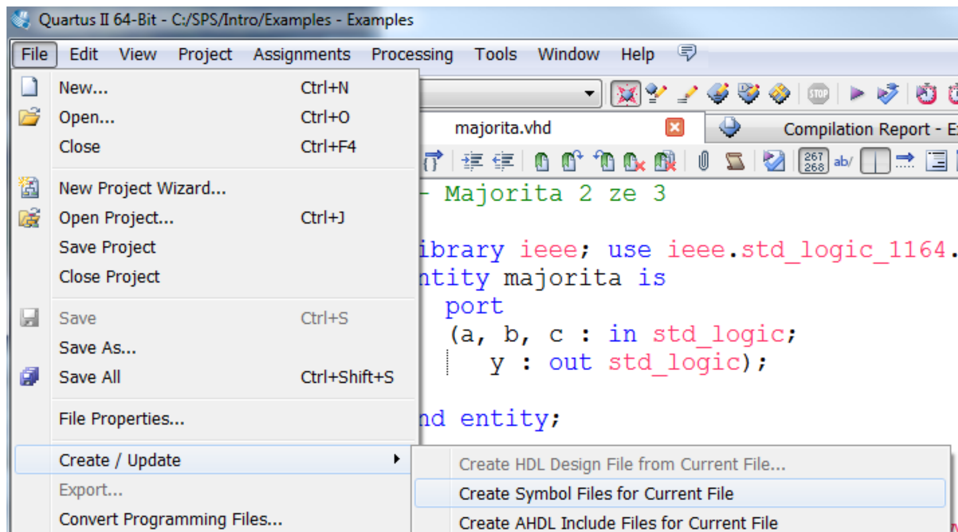
V zobrazeném schématu, jak RTL tak Technology, lze vybírat nakreslené prvky a měnit jejich kresbu přes kontextové menu. Formát lze upravit buď přes Viewer Options..., nebo z hlavního menu Tools->Options případně i Tools->Customize. Jednotlivé členy se dají vyhledat v podokně Netlist Navigator. Pokud jeho okno nevidíte, vyvolejte ho z menu View. Můžete tady volně experimentovat, vše je jen read-only.



10.5 Vytvoření schematické značky pro symbolický editor

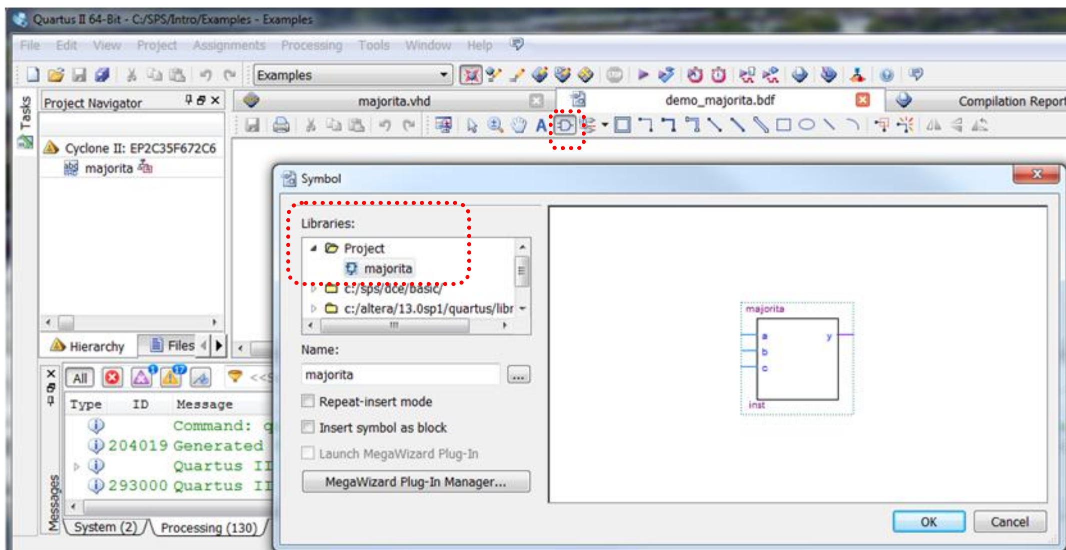
Schematickou značku vytváříme pouze tehdy, když chceme navržený obvod použít v symbolickém editoru. Jinak není potřeba; VHDL kódy ji nevyužívají.

Otevřeme soubor tak, aby byl viditelný v editoru. Vyzkoušíme, zda se přeloží, protože jinak se značka nevytvoří. Z hlavního menu volíme File-> Create/Update-> Create Symbol Files for Current File



Značka se ukládá jako textový soubor *.bsf, který obsahuje pouze seznam vstupů a výstupů a popis čar značky ve formátu podobném jazyku LISP. Souboru lze sice zobrazit textovým editorem, ale přímé opravy nedoporučujeme, struktura textu je příliš složitá. Zmiňujeme se o textovém formátu jen kvůli tomu, abychom zdůraznili, že *.bsf neobsahuje žádný kód, jen identifikátory vstupů a výstupů a čáry. Díky tomu **zůstává značka platná** tak dlouho, dokud nezměníme vstupy a výstupy, tedy sekci port či generic v entitě. Jedině pak ji musíme znovu vytvořit. Editujeme-li jen architekturu, není nutné ji znovu udělat.

Chceme-li ji vložit do symbolického souboru, vyvoláme Symbol Tool (buď z lišty nebo dvojklikem kamkoli na volnou plochu okna editoru). Značku najdeme v knihovně Projekt.

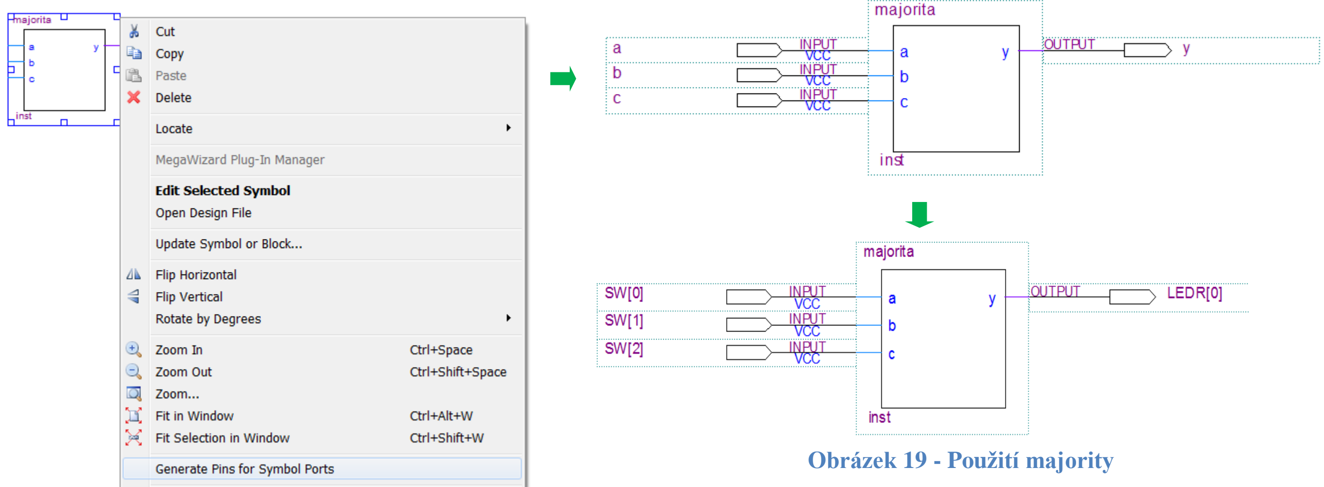


Důležitá poznámka: *Značka se vytváří tam, kde se nachází *.vhd soubor. Když ten neleží v hlavním adresáři projektu anebo na cestě zadané při konfiguraci jako adresář knihovny,* ***pak se značka nemusí najít****. Pokud jsme tedy omylem uložili *.vhd soubor mimo hlavní adresář projektu, musíme umístění souboru změnit, jak jsme již popsali v kapitole 10.1 na str. 74, a znova vygenerovat značku .*

Připojíme-li k majorita vstupy a výstupy pro její vyzkoušení, můžeme tak učinit buď manuálně, nebo si ušetřit trochu práce — pravou myší vyvoláme kontextové menu vložené schematické značky **majorita** a

necháme si pro ni automaticky vygenerovat vstupy a výstupy volbou "Generate Pins for Symbol Ports". Vložené "I/O pin" posléze přejmenujeme podle "assignments" desky DE2, buď přes kontextové Properties jednotlivých I/O pinů, nebo dvojklikem na pin.

Nejvýhodnější metodou je vybrat jen I/O název, což jde dvojklikem přímo na text uvnitř pinu, avšak jen za předpokladu, že není označený celý pin. Ten se od-označí klikem kamkoli na volnou plochu a pak již jde vybrat samotný text. Po edici textu dáme Enter, což provede přechod další vstup/či výstup, který ležící v grafickém editoru níže. A přejmenování více podobných výstupů lze urychlit, když si zkopírujeme si do schránky vhodný prototyp názvu, ten vložíme, opravíme a dáme enter k přesunu na další.



Obrázek 19 - Použití majority

Poznámka: Použití symbolického editoru bývá přehledné, ale pracné. Ve VHDL by se "Obrázek 19 - Použití majority" dal napsat s využitím příkazu port map*, který se probírá v kapitole 6, takto:*

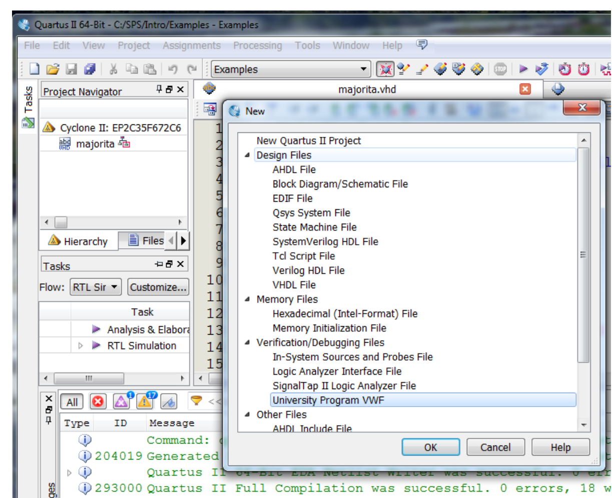
```
-- Majorita 2 of 3
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity  demo_majorita is
        port (  SW : in std_logic_vector(2 downto 0);
                LEDR : out std_logic_vector(0 to 0));
end;
architecture dataflow of demo_majorita is
    component majorita is port (a, b, c : in std_logic; y : out std_logic );
    end component;
begin
        inst : majorita port map(SW(0), SW(1), SW(2), LEDR(0));
end;
```

10.6 Simulace

<u>Předpokládejme</u>, že máte vytvořený obvod majorita a **jeho úplný překlad proběhl bez chyby,** tj. jako poslední se vypsalo informativní hlášení "*Info (293000): Quartus II Full Compilation was successful. 0 errors...*".

Z hlavního menu Quartus zadejte "File->New..." a v dialogu, který se poté objeví, vyberte typ souboru pro simulaci: "University Program VWF" (Vector Waveform File)

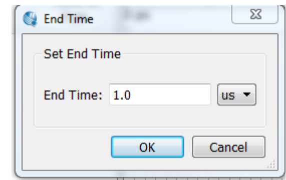
Dejte [OK]

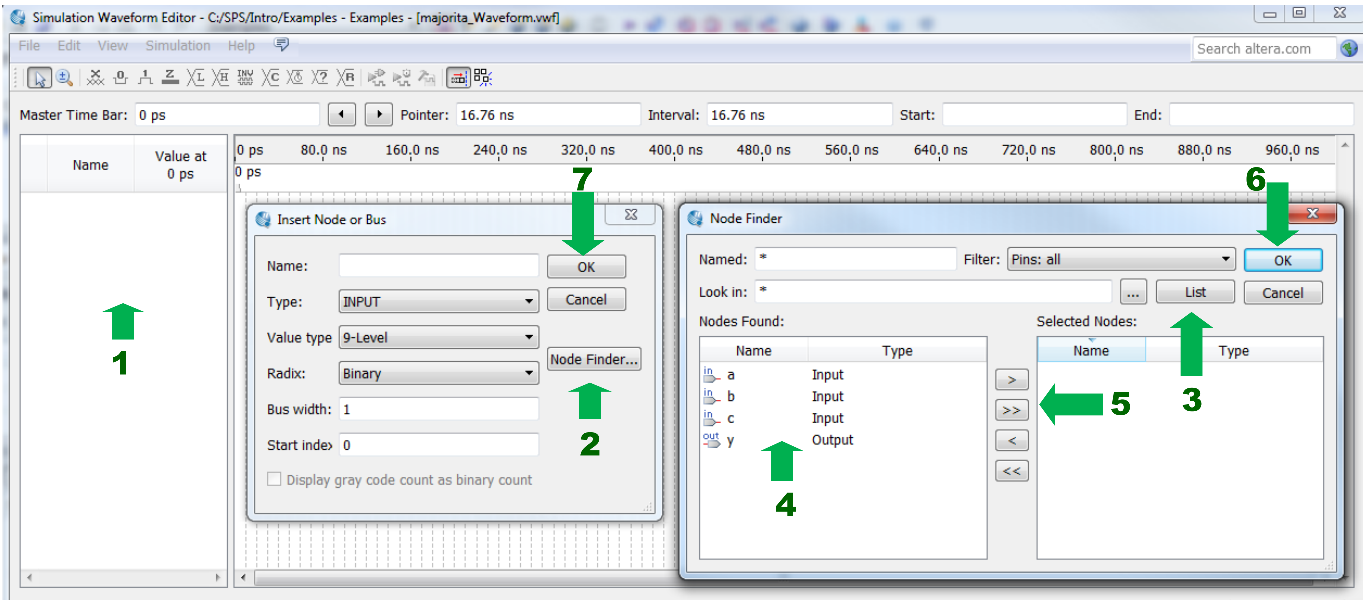


Otevře se okno Simulation Waveform Editor pro zadání průběhů, které hned uložíme File->Save As... jako majorita_Waveform.vwf

Poznámka: V Quartusu není vhodné rozlišovat soubory většinou pouhou příponou, dávejte jim odlišné názvy, jinak se vás editor bude často ptát, co vlastně chcete otevřít.

Případná **změna doby simulace** se musí provést nyní, tedy před vložením jakýchkoli signálů. Poté již ne. (*Tedy přesněji lze ji zadat i se signály, ale Quartus spadne na interní chybě, je-li delší. Zkrácení většinou projde.*)
Dialog se vyvolá z menu Edit->Set End Time...
Výchozí hodnota je 1 μs a maximální dovolená doba je 100 μs.

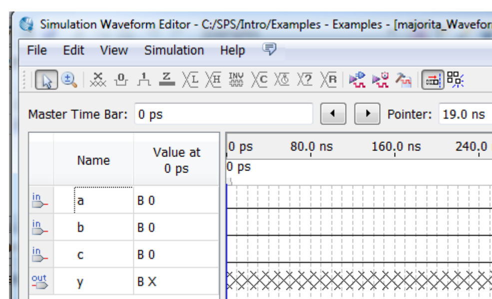




Nyní proveďte následující postup:

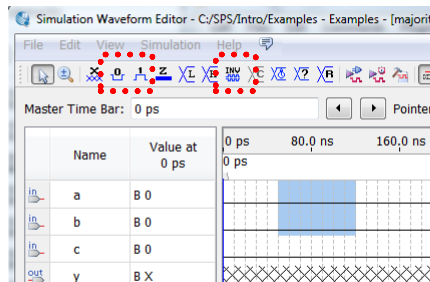
1. Z menu zvolíme Edit->Insert->Insert Node or Bus
   nebo levou myší provedeme dvojklik do okna Name, viz šipka 1.
2. V dialogu Insert Node or Bus můžeme napsat jméno vstupu či výstupu a vložit ho.
   Chceme-li si ušetřit práci, lze je vyhledat všechny vstupy a výstupy, když stiskneme [Node Finder...].
3. V dialogu Node Finder necháme výchozí *, což znamená vypsat vše, a nastavení Pins: all, tedy chceme vstupy a výstupy. Tedy volíme jen [List].
4. Buď vybereme v Nodes Found vstupy a výstupy pomocí Ctrl+levá myš a poté tlačítkem [>] je okopírujeme mezi Selected Nodes,
5. nebo můžeme použít přímo [>>] tlačítko a překopírovat vše. To je případ majority.
6. Tlačítkem [OK] zavřeme Node Finder dialog.
7. Tlačítkem [OK] zavřeme dialog Insert Node or Bus.

Pokud jste vše udělali správně, objevily se vybrané signály v okně Simulation Waveform Editor :

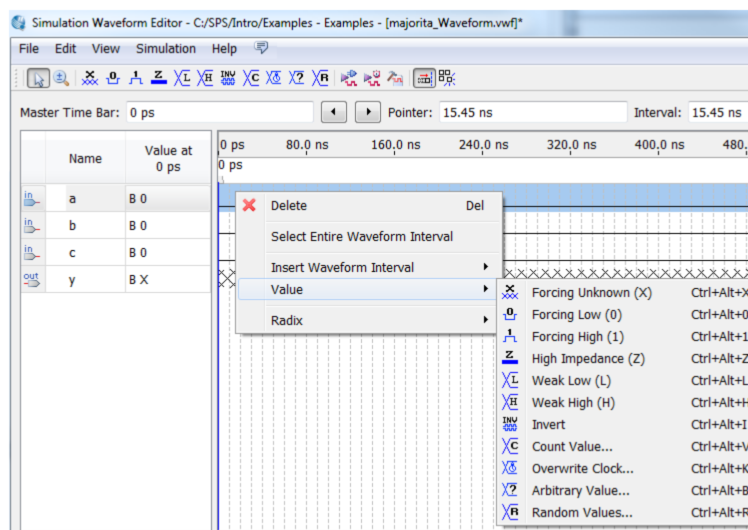


Máte-li signály v jiném pořadí, můžete si je libovolně přeuspořádat. Stačí přetáhnout řádky v Name podokně na jiné pozice běžným způsobem pomocí myši.

Nyní nakreslíme testovací signály pro vstupy. Vybereme levou myší buď celý signál klikem na jeho jméno, či část jednoho nebo více signálů, kterou chceme změnit najednou, a stisknete některé z označených tlačítek, pro definování na 0, na 1 či inverzi stávajících průběhu.

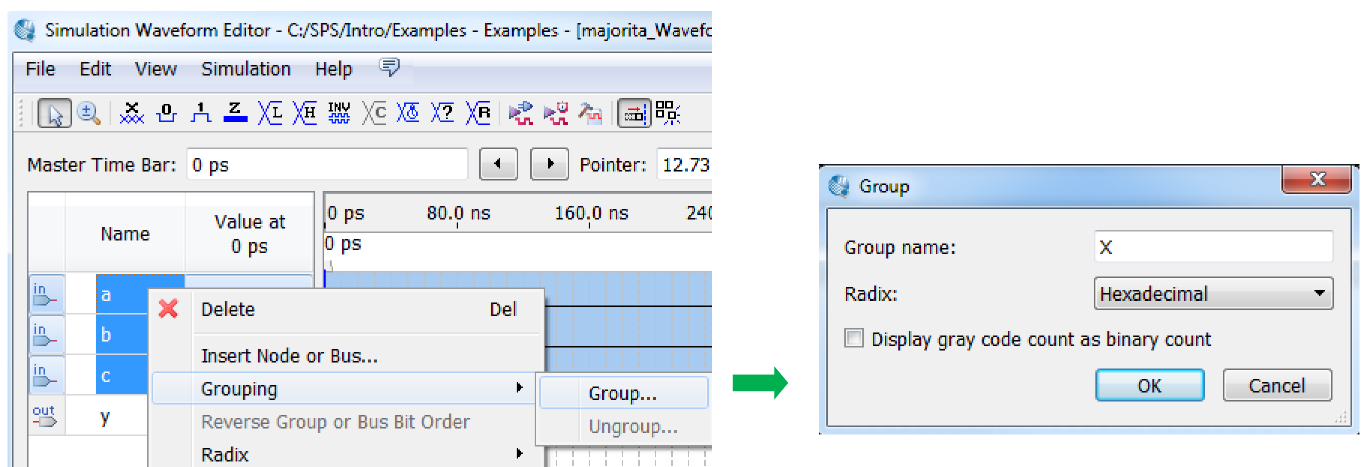


Eventuálně lze signály vkládat i přes kontextové menu (pravá myš), z lišty, či z menu Edit->Value, případně i klávesovými zkratkami.

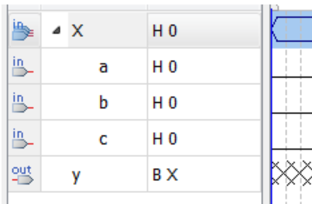


Potřebujeme-li vyčerpat všechny logické kombinace, využijeme čítač, avšak napřed si musíme vytvořit skupinu signálů.

1. V podokně Name vybereme signály pro skupinu a pravou myší vyvoláme kontextové menu, kde volíme Grouping->Group... V dialogu zadáme libovolné vhodné jméno, třeba X. Pozor, rozhodně nevolíme něco, co vypadá jako klíčové slovo ve VHDL či Verilogu, jako třeba in, či out nebo wire, jinak se simulace nemusí podařit. Nastavíme případně vhodný radix, v němž si přejeme skupinu vypisovat, a dáme OK.



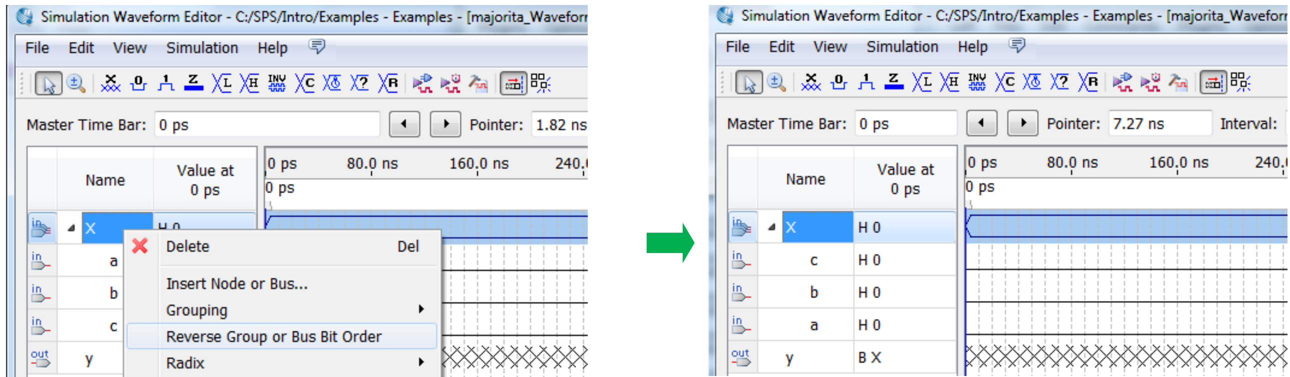
2. Vznikne nám skupina signálů.



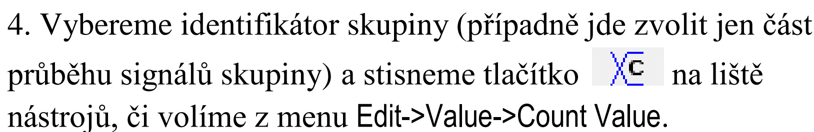
3. Čítač počítá od nejspodnějšího signálu, kterému přidělí nejnižší váhu, k dalším nad ním. Chceme-li obrátit váhy vodičů, můžeme to udělat, pokud vybereme skupinu a volíme buď z menu

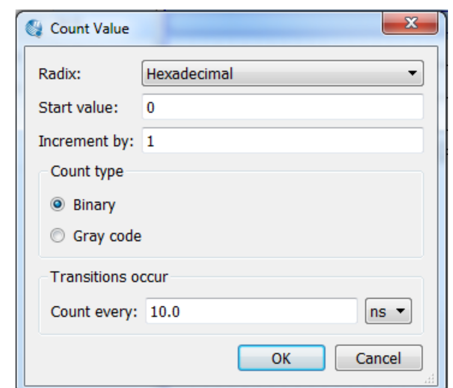
Edit->Reverse Group or Bus Bit Order

nebo také přes kontextové menu názvu skupiny.



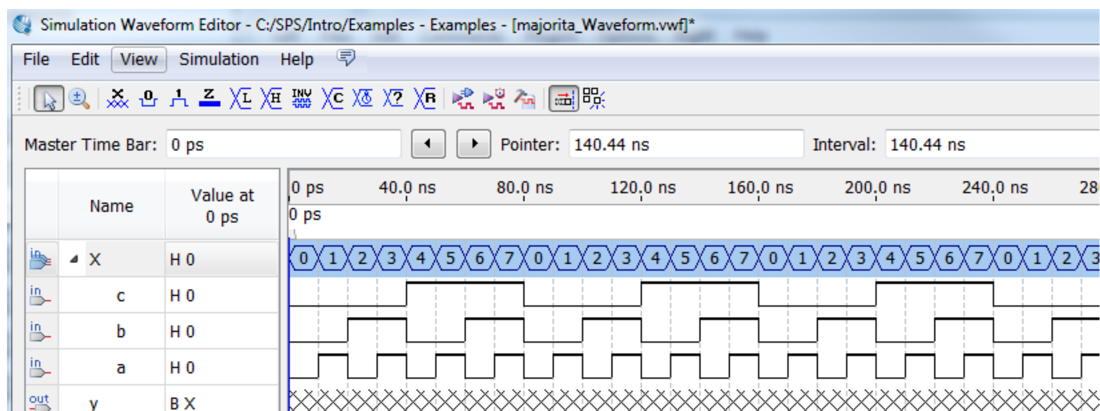
Podobným způsobem se dá kdykoli změnit i radix, v němž se vypisuje skupina.

4. Vybereme identifikátor skupiny (případně jde zvolit jen část průběhu signálů skupiny) a stisneme tlačítko  na liště nástrojů, či volíme z menu Edit->Value->Count Value.

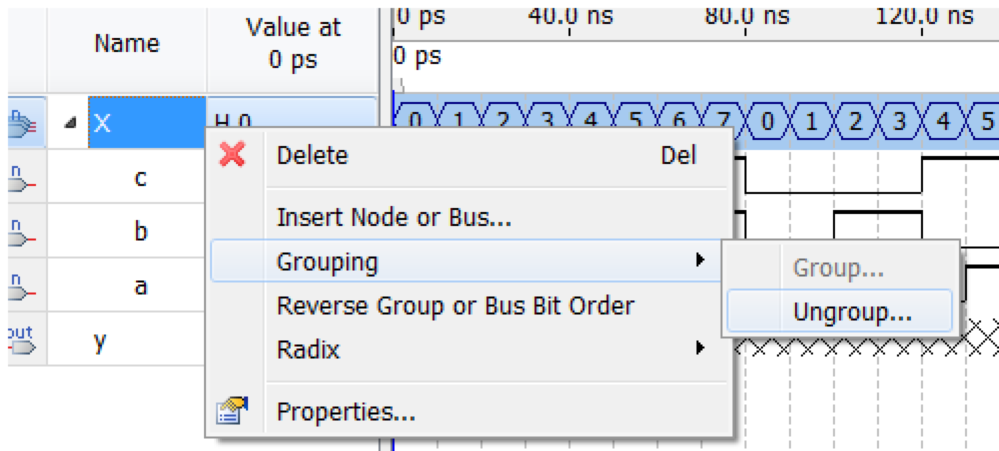


V dialogu, který se objeví, vyplníme údaje o periodě čítání. Hodnota 10 ns se nabízí jako výchozí hodnotu, což můžeme u majority klidně ponechat. Vybereme typ čítače, buď binární čítající v pořadí čísel nebo Gray code, kde jsou čísla uspořádána tak, aby se vždy mněnil jen jeden bit. I zde existuje možnost zadat radix výpisu.

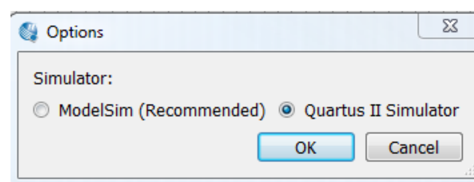
Zadejte [OK] a signály se nám naplní automaticky vytvořenými průběhy, viz dole. Ctrl+kolečko myši mění zoom, ten se dá upravit i přes hlavní menu View->



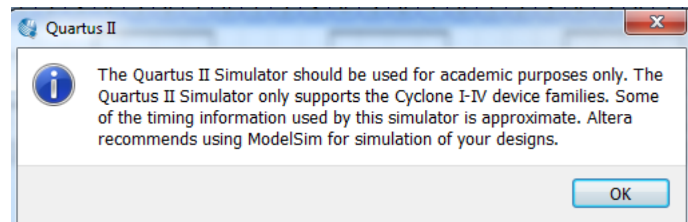
5. Poté můžete případně zrušit skupinu přes kontextové menu Grouping->Ungroup, nebo ji ponechat, závisí na nás. Někdy, avšak jen málokdy, se může stát, že simulátor odmítne skupinu, protože její název s něčím koliduje, takže nám nic jiného nezůstane.



6. Dále potřebujeme nastavit typ simulace. Z hlavního menu volíme Simulation->Options



V dialogu zadáme raději Quartus II Simulator určený pro University Program, který pracuje vždy. Firma Altera ho vložila do Quartus na žádost univerzit, a tak potvrdíme, že jde o výuku:

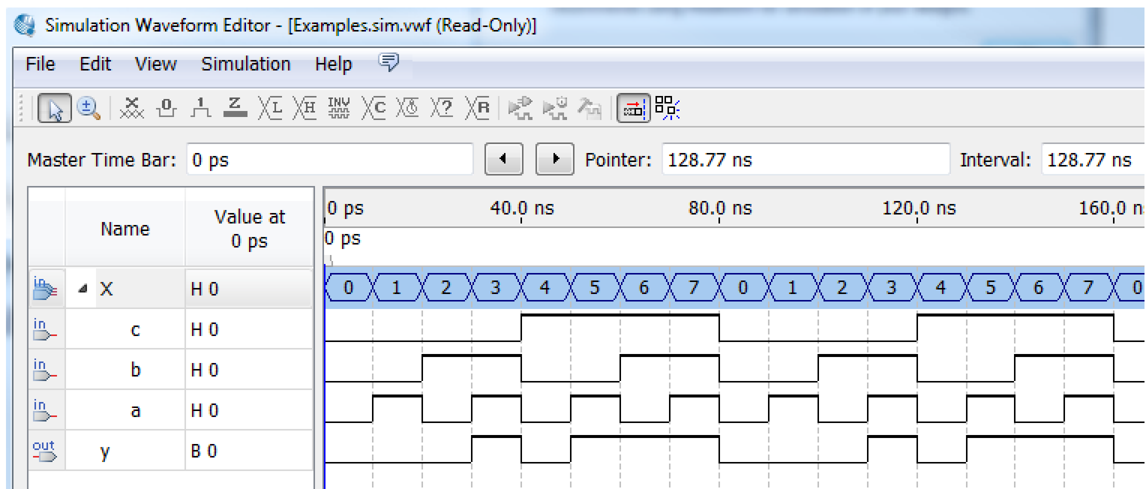


Můžete samozřejmě později zkusit i ModelSim, třeba bude fungovat, máte-li správně vytvořený projekt a nastavené cesty k ModelSim-Altera: Tools->Options->EDA Tool Option (default hodnota vytvořená při instalaci Quartusu na C disk je C:\altera\13.0sp1\modelsim_ase\win32aloem).

7. Spustíme simulaci: Simulation->Run Functional Simulation, která simuluje logické obvody jako logické operace ☺

Není předchozí věta divná? Není, skutečný logický obvod provádí nejen logickou operaci, ale také přidává časové charakteristiky jako třeba dobu zpoždění, což vyvolává četné parazitní jevy, na něž cílí časové simulace. Ty patří již k pokročilejší úrovni návrhu a budeme o nich mluvit na přednášce. Zatím vystačíme s pouhou funkcí.

8. Simulátor si z grafických průběhů sestaví testbench ve Verilogu a spustí jeho výpočet. Poté se objeví nové okno s výsledky, které nelze editovat, jen prohlížet, jako se ostatně uvádí i v jeho titulku. Chcete-li změnit průběhy a provést jinou simulaci, musíte se vrátit k původnímu oknu.



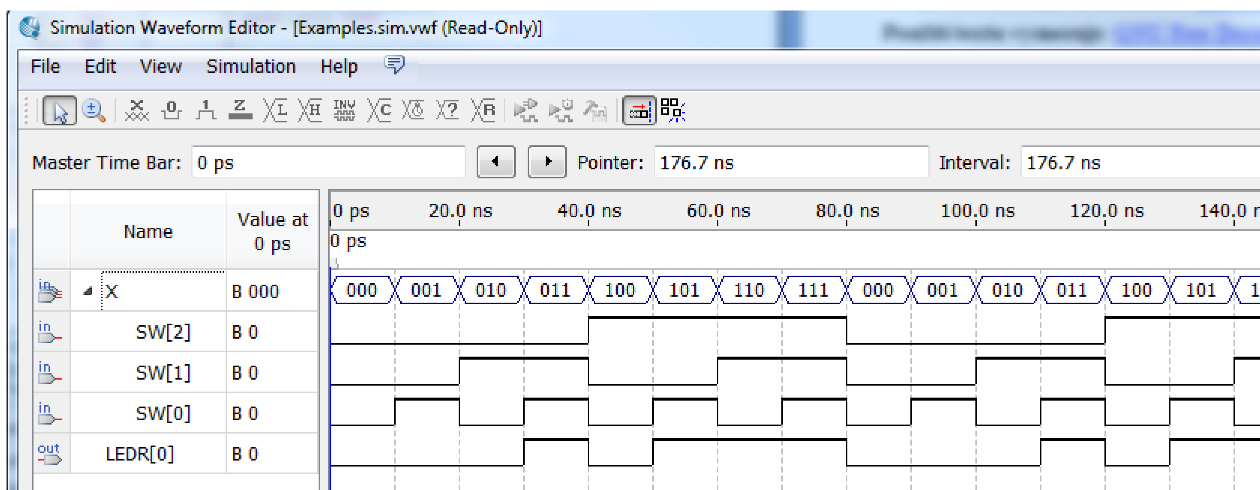
Další možnosti interního simulátoru najdete v manuálu Introduction to Quartus II Simulation na stránce http://dcenet.felk.cvut.cz/edu/fpga/navody.aspx.

Vestavěný simulátor slouží jen k výuce. Pokročilejší testy obvodů se provádí v aplikaci ModelSim, avšak ta již vyžaduje napsání souborů typu testbench. Vestavěný simulátor je vytvoří za nás, avšak pouze v omezené míře. Neumí všechny možné operace.

10.7 Vložení indexovaných signálů

Chceme-li simulovat obvod s vektory, např. ten na Obrázek 19 na str. 80, pak se v seznamu signálů objeví také skupiny, Input a Output Group. Nelze vložit vše, můžeme zvolit buď skupiny, nebo jednotlivé signály, čemuž zde dáme přednost. Náš obrázek ve Vector Waveform File se totiž při simulaci napřed přepíše na kód typu testbench, až ten provede samotný výpočet průběhů. V pin assignments definicích Quartusu mají skupiny SW a LEDR odlišné rozsahy, s čímž si zabudovaný akademický simulátor někdy neporadí, takže se simulace nepovede. V jiných případech lze vložit skupiny.

Skupiny si sestavíme sami a vložíme do nich čítání hodnot, poté spustíme simulaci.



11 Závěr

Výukový materiál vznikl během července až srpna 2019. Jde o původní text určený pro studenty předmětu Logické systémy a procesory, který se přednáší na Katedře řídicí techniky ČVUT-FEL Praha.

I když se prováděly četné korektury textu, mohly přece jenom někde zůstat nejasnosti ve výkladu, případné překlepy či jiné důsledky řádění škodolibých tiskařských šotků. Autor uvítá, pokud mu napíšete, kde se diblíci ukrývají, aby je mohl pochytat.

Autor: Richard Šusta, richard@susta.cz

Domovská stránka dokumentu: http://dcenet.felk.cvut.cz/edu/fpga/navody.aspx

Použití textu vymezuje: GNU Free Documentation License

Copyright 2019: ČVUT Fakulta elektrotechnická,
Katedra řídicí techniky
Technická 2, Praha 6

11.1 Historie verzí dokumentu

Verze 1.1 - září 2019

- doplněný komentář na straně 51
- přidaná tabulka resolved do přílohy B, na straně 71
- korekce drobných překlepů
- přidaná tato historie dokumentu

Verze 1.0 dokončena v srpnu 2019

- vznikla rozšířením předchozího dokumentu "Příkladný úvod do VHDL" z roku 2013, který měl délku 43 stran. Nově doplněné části obsahují nejen podrobnější výklad, ale i další příklady a samostatné úlohy k procvičení. Významně se rozšířil i popis "structural" modelovacího módu a tvorby knihoven.