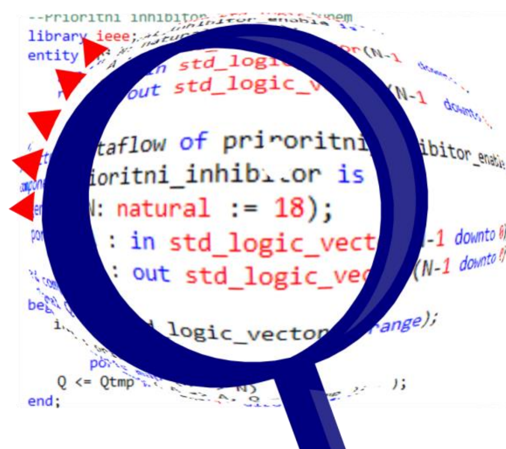


Úvod do návrhu obvodů v jazyce VHDL

II.

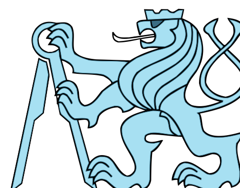
– sekvenční příkazy –

Pracovní verze dopsaná cca ze 2/3!

Opravy se uvádějí na další straně

Richard Šusta

Katedra řídicí techniky
ČVUT-FEL v Praze



Verze beta 0.51 - 6. června 2023

Opravy:

6.6. 2023: Korekce obrázku 14. a 15. na straně 37 a 38

1 ÚVOD 5
1.1 O sekvenčním VHDL 5
2 POPIS KOMBINAČNÍHO OBVODŮ SEKVENČNÍMI PŘÍKAZY 7
2.1 @ Příklad I.: 3-bitový inhibitor 7
2.1.a Prioritní kaskádou multiplexorů 8
2.1.b Multiplexor 11
2.1.c Logické rovnice 12
2.1.d Rozšíření na obecný inhibitor 14
2.1.e Naprogramovaná obecná hrůza 15
2.2 *** Cvičná úloha 1: - Obecný lineární ukazatel 17
2.3 Více o více architekturách 18
2.4 Příkaz process 21
2.5 Příkaz function 23
2.5.a @ Příklad II. - Unární xor jako funkce v knihovně 24
2.6 *** Cvičná uloha 2: Porovnání čísel v Grayově kódu 25
2.7 Příkaz procedure 26
2.8 @Příklad III.: prioritní inhibitor rozlišující tři trojice 26
2.9 *** Cvičná úloha 3: MaxMinSwap pomocí procedure 28
2.10 Příkaz after u <= přiřazení do signálu 29
2.11 Blocking versus non-blocking příkazy 31
3 OBVODY ŘÍZENÉ HODINOVÝM SIGNÁLEM 33
3.1 Obvod typu D-Latch 33
3.2 Klopný obvod DFF - Data Flip-Flop 33
3.3 Synchronní klopný obvod ve VHDL 34
3.4 @ Příklad IV. - Synchronní klopný obvod s dvojím nulováním 35
3.5 @ Přiklad V. - Registr s Q a QN 38
3.6 *** Cvičná úloha 4: Registr s xor Data 40
4 ZÁKLADNÍ SYNCHRONNÍ OBVODY 41
4.1 Synchronní obvod se strukturou automatu 41
4.2 @ Přiklad VI. - Grayův 2bitový čítač 42
4.2.a VHDL kód 43
4.3 Záludnosti synchronních obvodů - zpoždění hodin a rychlé smyčky 44
4.4 *** Cvičná úloha 5: Grayův čítač obecné délky 44
4.5 Posuvné registry 45
4.5.a @ Příklad VII. - Obousměrný 4-bitový posuvný registr 46
4.5.b *** Cvičná úloha 6: Posuvný obecné délky s nulováním a nastavením 46
4.5.c Posuvné registry k sériovému přenosu 47
4.6 Kruhový čítač (posuvný registr) 48
4.6.a Johnsonův čítač 48
4.6.b *** Cvičná úloha 7: Variabilní dělič 49
4.7 @ Příklad VII. - Navigační světlo 50
4.7.a Robustnost řešení 51
4.8 Čítače a děliče 53
4.8.a @ Příklad VIII. Geneze děliče 10 se symetrickým výstupem 54

4.8.b @ Příklad IX. Dělič vysoké frekvence 10 miliony se symetrickým výstupem 59
4.8.c ***Cvičná úloha 8 - Generátor pulzně šířkové modulace 61
4.8.d Univerzální čítač 62
5 KONEČNÉ AUTOMATY ALIAS FSM (FINITE STATE MACHINES) 65
6 PŘÍLOHA A: ŘEŠENÍ CVIČNÝCH ÚLOH 66
6.1 Cvičná úloha 1: Lineární ukazatel 66
6.2 Cvičná úloha 2: Porovnání čísel v Grayově kódu 67
6.3 Cvičná úloha 3: MaxMinSwap pomocí procedure 68
6.4 Cvičná úloha 4: Registr s xor Data 69
6.5 Cvičná úloha 5: Grayův čítač 70
6.6 Cvičná úloha 6: Posuvný obecné délky s nulování a nastavením 71
6.7 Cvičná úloha 7: Variabilní dělič 72
7 SEZNAM ČÍSLOVANÝCH OBRÁZKŮ A TABULEK 73
7.1 Seznam tabulek 74
8 ZÁVĚR 75
8.1 Historie verzí dokumentu 75

1 Úvod

Učebnice je součástí pětice skript, která postupně vznikala od roku 2012 a najdete ji na stránce:

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

0. **Binární prerekvizita** se věnuje kódování celých čísel se znaménkem a bez něho, jejich zápisům v hexadecimálním a BCD tvaru včetně vzájemných převodů. Nastíní i uložení písmen. Shrnuje bazální vědomosti, jejichž bezpečnou znalost předpokládají jak další učebnice, tak přednášky.
1. **Logické obvody na FPGA** vysvětlí základní logické konstrukce, bez nichž lze těžko cokoli efektivně navrhovat. Jak název napovídá, zmíní sice dost obecných částí, ale uvede též zapojení při jejich realizaci na FPGA.
2. **První díl o jazyce VHDL,** Úvod do návrhu obvodů v jazyce VHDL I. – souběžné příkazy**,** se zaměřil na příkazy jeho souběžné kódové domény *(concurrent).*
3. **Druhý díl o VHDL,** který právě čtete, předpokládá aspoň orientační znalost předešlých učebnic ze seznamu nahoře. Věnuje se druhé kódové doméně *sequential*, v níž se návrh tvoří lidem přehlednějším stylem *behavioral*, tedy popisem chování obvodu. Musí však tady psát velmi rozumně, jinak vznikne odstrašující paskvil zapojení. Sekvenční zdrojový kód se totiž nedá již přímo syntetizovat. Překladač ho napřed musí konvertovat na meta-schéma stylu *concurrent* a doplnit všechny pomocné signály, teprve z něho sestaví obvod. Kvůli tomu se napřed probírala první doména, aby se objasnilo, na co se *sequential* příkazy převádějí, a dokázali jsme si představit zapojení, které vzniká z našeho kódu.
   Pozn.: Druhý díl je napsaný z velké části, cca 60 až 70 %.
4. Závěrečná část **Speciální obvody** *se připravuje. Zatím existuje leda její třetina.*

   Zahrne pokročilejší zapojení jako třeba zpětnovazební posuvné registry, konečné automaty a využití pamětí. Přidá i fázové závěsy, nábojové pumpy a ošetření vstupů a výstupů. Rozšíří také znalosti o CMOS hradlech. Při prvotním studiu ji lze vynechat bez ztráty souvislostí.

1.1 O sekvenčním VHDL

Učím návrhy obvodů dloho, ale s užitím VHDL jen od roku 2007. Shlédl jsem již tisíce pokusů o jejich implementace a ověřil jsem si, že začátečníci často tíhnou k jejich nesprávnému „naprogramování".

Klasické programy se vytvoří vhodnými konstrukcemi dle použitého jazyka. Zdrojový kód se, když se přeložil a spustil na procesoru, vždy provádí instrukcemi strojového kódu, které se vykonávají sekvenčně jedna po druhé.

Nehodí se slepě napodobovat konstrukce či techniky programovacích jazyků v popisu obvodu, který je souběžnou strukturou, jíž protékají hodnoty. Většinou žádá jiné postupy ve svém zdrojovém kódu. Někdy lze sice použít jakousi obdobu klasického programu, šikovný překladač ji převede, ale rozhodně ne pokaždé optimálně, zejména u větších a složitějších obvodů.

Maximální zapojení, který se vtěsná do FPGA, není totiž omezeno vyčerpáním všech dostupných logických elementů (LE), základních stavebních prvků. Jeden LE zpravidla zahrnuje, dle typu použitého FPGA, jednu až dvě logické funkce se čtyřmi až šesti vstupy, za nimiž následují jeden až dva klopné obvody. Máme zkušenosti, že se vypoužije nejvýše kolem 80 % obvodu a poté se již vyčerpají dostupné propojky mezi nimi. Blíže se problematika rozebírala v učebnici Logické obvody v kapitole 5.5. A méně optimální návrhy bývají hladovější především na propojky, a tak se hodí tvořit s rozumem.

Nyní si připomeneme dvě základní použití HDL kódu, ať už ve Verilogu nebo ve VHDL či jiném HDL.

- **Simulace obvodu.** Během ní se všechny HDL kódy zpracovávají v simulačním prostředí způsoby blízkými klasickému programu, tedy příkaz po příkazu, akorát se více respektuje paralelní činnost obvodu. Pokud nepíšeme kód, který se syntetizuje, ale tvoříme například pomocný testbench na ověření činnosti obvodů, pak můžeme opravdu napodobovat klasické programování. Stačí, když napíšeme kód, který provádí, co má, je přehledný, a samozřejmě bez chyb, což je hlavní!
- **Syntéza obvodu**. Ve zdrojovém kódu se rovněž držíme přehlednosti, ale navíc volíme takové konstrukce, aby se dobře konvertovaly, protože výsledek rozhodně nepoběží jako strojový kód, ale bude zapojením obvodu. Nebudeme-li se ohlížet na jeho možnosti, lehce syntetizujeme cosi jakž-takž fungujícího, neboť překladač šikovně ohne náš kód díky svým optimalizacím a heuristikám, aby ho „nějak“ vtěsnal do obvodu. A pod slovíčkem „nějak“ se snadno schová jednak pseudo-funkce, která jednou za čas, předem netušíme kdy, poskytne chybné výsledky, a jednak i dost ne-optimální struktura zbytečně vyčerpávající použité prostředky.

Na základě svých zkušeností jsem zvolil netradiční strukturu učebnic, a to od souběžného stylu, abych studenty nasměroval k návrhům obvodů, a ne na jejich naprogramování.

V roce 2019 jsem vytvořil „Návrh obvodů ve VHDL modelovacími styly *dataflow* a *structural*“, avšak nezvolil jsem vhodný název, jelikož obsahoval hodně slov, které začátečníků nic neříkala.

V roce 2023 jsem jučebnici raději přejmenoval na „Úvod do návrhu obvodů v jazyce VHDL I. – souběžné příkazy“, které osobně považuji za základ. Jedině ty se dají přímo implementovat v obvodu. V nich je i „mnohem obtížnější“ napsat mizerné zapojení, jelikož jejich konstrukce nutí do odlišného stylu.

Nyní přidávám nazazující díl, v němž se již používají sekvenční příkazy bližší klasickému programování, což urychluje a usnadňuje návrhy. Řada výukových materiálů začíná od nich, neboť se jimi rychleji získají výsledky, ale v nich se již hbitě „docela snadno“ vytvoří obvodový paskvil. Stačí příliš programovat.

Hodí se stále mít na zřeteli, že sekvenční příkazy nelze přímo implementovat! Překladač je převádí na meta-kód složený ze souběžných (concurrent) konstrukcí a teprve ty optimalizuje na obvod. Známe-li trochu souběžné návrhové styly dataflow a structural, lépe si představíme, co bude výsledkem, našeho popisu.

Vždy dbáme na fakt zdůrazněný v řadě učebnic, že **simulace je leda simulace**. Může vypadat báječně, ale nevynáší finální verdikt. I když spatříme úžasné průběhy, náš návrh může lehce představovat odpad na přepsání. Úloha převodu kódu do zapojení zahrnuje totiž některé části se složitostí SAT (satisfaction) problému, který je NP-úplný (NP-complete)[1] . Ze špatného návrhu se nevytvoří vhodný obvod, ale může nastat exponenciální nárůst jeho složitosti. Z časových důvodů se optimalizace obvykle zastaví na nějakém zdánlivě vhodném řešení, což brzy demonstrujeme v kapitole 2.1.e na straně 15.

[1] Blíže viz Wikipedia: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem a https://en.wikipedia.org/wiki/NP-completeness

2 Popis kombinačního obvodů sekvenčními příkazy

VHDL kód zahrnuje dvě zdrojové domény a v každé se používají odlišné typy příkazů:

- **souběžnou** (concurrent) - kde se píší výhradně příkazy obou stylů dataflow a structural, které známe z první části a dají se snadno implementovat jako zapojení. V žádném případě se tady nesmí objevit sekvenční příkazy, protože zde se neprovádí jejich konverze.
- **sekvenční** (sequential) - v té se naopak pokaždé očekávají jedině sekvenční příkazy a nelze použít souběžné, na ty se sekvenční příkazy budou teprve převádět.

  Začátek a konec sekvenční zdrojové domény ohraničuje některá ze tří konstrukcí:
  - Proces   end process; - nejčastější případ;
  - Function   end function; - definice části obvodu s jedním výstupní veličinou;
  - Procedure   end procedure; - definice části obvodu s více výstupními hodnotami.

  Žádná z nich se ale při syntéze obvodu nevolá ve smyslu klasického (pod)programu! Zapojení vytvořené z funkce či procedury se pokaždé vkládá celé, jinými slovy, použijeme-li pětkrát jednu funkci, tak se vloží pět kopií obvodu, který realizuje operaci popsanou jejími příkazy.

Každé „správné" pravidlo má svou výjimku a tou je VHDL příkaz **<=** souběžného přiřazení, který se smí sice použít v obou doménách, ale **pozor** — v sekvenční doméně získává odlišné vlastnosti. Nejzávažnější rozdíl se týká zpětného čtení jím přiřazení hodnoty a poprvé ho demonstrujeme v kapitole 2.1.a na straně 8. Důvody rozebereme pak v kapitole 2.9 na straně 28.

Do sekvenčních příkazů uvedeme příkladem různě řešeného jednoduchého kombinačního obvodu, kde vždy srovnáme popisy v obou doménách zdrojového kódu. Až poté provedeme detailnější vysvětlení.

2.1 @ Příklad I.: 3-bitový inhibitor

Vytvořte obvod, který propustí pouze nejvyšší bit ve stavu logické '1', a současně hlásí, že došlo k potlačení některého nižšího bitu. Jeho funkce se podobá „radio button"prvku v programech či webových stránkách, a v obvodech se používá třeba při obsluze přerušení, kdy se dál pošle pouze nejvýše prioritní vstup.

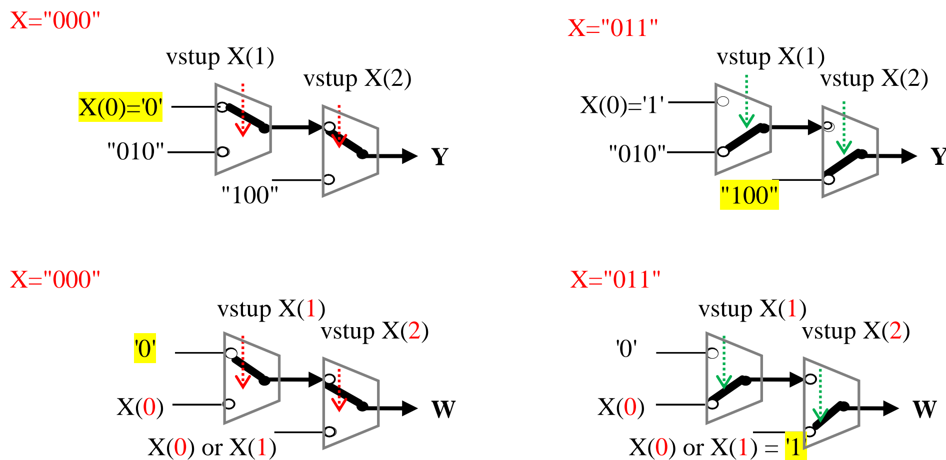
Jeho obdobu znáte z 3. cvičného příkladu v první části, který jste v kapitole 4.3 řešili sami pro 8 vstupů. Tady máme jen 3, ale přidali jsme si indikaci W, třeba od waiting, potlačení nějakého nižšího vstupu.

Vstup	Výstup	
X(2 downto 0)	Y(2 downto 0)	W
"000"	"000"	'0'
"001"	"001"	'0'
"010"	"010"	'0'
"011"	"010"	'1'
"100"	"100"	'0'
"101"	"100"	'1'
"110"	"100"	'1'
"111"	"100"	'1'

Tabulka 1 - Pravdivostní tabulka 3-bitového prioritního inhibitoru

2.1.a Prioritní inhibitor kaskádou multiplexorů

V řešení 3. cvičné úlohy z prvního dílu učebnice se doporučoval souběžný příkaz when else. Jeho sekvenční obdobou je příkaz if then elsif. Oba modelují obvod kaskádou dvouvstupových multiplexorů, tedy analogií dvoupólových přepínačů. Pokud je vyšší přepínač přepnutý, pak se ignorují výsledky nižších voleb, na obrázku dole vlevo, zatímco vpravo je nejvyšší priorita. Všimněte si, že obvod obsahuje kompletní zapojení a pouze vybíráme, jaký výsledek se právě posílá na výstup.



Obrázek 1 - Modelování prioritního inhibitoru kaskádou dvouvstupových multiplexorů

Kaskády W a Y mají na vstupech jak logické funkce, tak konstanty. Vektor X zahrnuje tři prvky, ale nejnižší multiplexor kaskády by přepínal mezi '1' při X(0)='1' a logickou '0' pro X(0)='0', a tak se nahradil X(0).

Souběžný VHDL kód s použitím příkazu when else by vypadal třeba takto: (*Knihovna numeric_std zde zatím není potřeba, ale využijeme ji později.)*

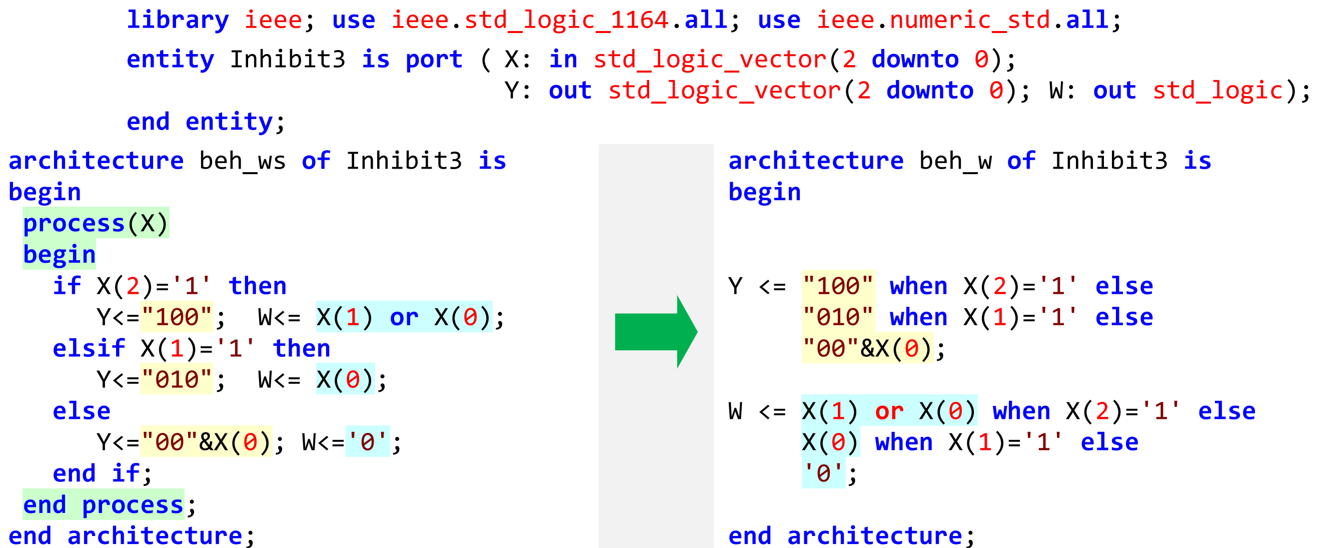
```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity Inhibit3 is
port (X: in std_logic_vector(2 downto 0);
      Y: out std_logic_vector(2 downto 0);
     W: out std_logic);
end entity;

architecture beh_w of Inhibit3 is
begin
   Y <= "100" when X(2)='1' else
        "010" when X(1)='1' else
        "00"&X(0);

   W <= X(1) or X(0) when X(2)='1' else
        X(0) when X(1)='1' else
        '0';
end architecture;
```

Jelikož when else přiřazuje pouze jednomu signálu, vložili jsme dvě kaskády multiplexorů. Vidíme zde i operátor spojení do vektoru `"00"&X(0)`, tedy tříprvkový vektor se složkami `0|0|X(0)`.
Nyní využijeme sekvenční if then elsif, tedy obdobu k souběžnému when else. Vložíme si implementaci pomocí něho coby další architekturu[2].

[2] V první části jsme se zmínili, že k entitě lze definovat několik architektur, ale vždy jsme používali výhradně jednu. V druhé části budeme více experimentovat s různými styly zápisu, k čemuž využijeme vícenásobné architektury. V kapitole 2.3 na str. 16 si ukážeme, jak se zvolí požadovaná architektura. Zatím si můžete nadbytečné architektury zakomentovat, aby v kódu existovala pouze jedna aktivní.

Napíšeme obě architektury vedle sebe kvůli porovnání. Sekvenční if then elsif obdoba k souběžnému příkazu when else zpřehlednila zápis, protože za then lze napsat několik příkazů i jakékoli vnořené sekvenční konstrukce, z nichž zatím známe jen if then else. Překladač vše rozloží na nezávislé kaskády.

Náš sekvenční kód vlevo se převede na souběžnou strukturu vpravo, která se namodeluje v metaschématu během prvního kroku překladu a výsledek umí zobrazit RTL Viewer.



```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity Inhibit3 is port ( X: in std_logic_vector(2 downto 0);
                          Y: out std_logic_vector(2 downto 0); W: out std_logic);
end entity;
```

```
architecture beh_ws of Inhibit3 is
begin
 process(X)
 begin
   if X(2)='1' then
      Y<="100";  W<= X(1) or X(0);
   elsif X(1)='1' then
      Y<="010";  W<= X(0);
   else
      Y<="00"&X(0); W<='0';
   end if;
 end process;
end architecture;
```

```
architecture beh_w of Inhibit3 is
begin


Y <= "100" when X(2)='1' else
     "010" when X(1)='1' else
     "00"&X(0);

W <= X(1) or X(0) when X(2)='1' else
     X(0) when X(1)='1' else
     '0';

end architecture;
```

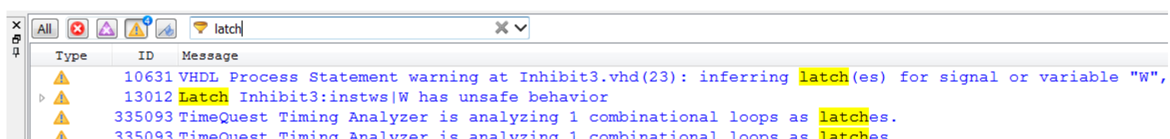
Sekvenční doménu zdrojového kódu uvozuje process(X), v němž **X ale není vstupní parametr** jako ve funkci klasického programování. Jde pouze o specifikaci signálu, jehož změna vyvolá změnu výstupních signálů. O podrobnějších vlastnostech sekce process projednáme podrobně v kapitole 2.4 na straně 21, až probereme všechny potřebné pojmy k objasnění jeho funkce.

Všimněte si, že if then elsif začíná od bitu X(2), tedy od nejvyšší priority, a stejně jako u when else ovlivňuje výstup nižší podmínka tehdy a jedině tehdy, když není aktivní žádná vyšší. Stačí tedy pokračovat testem elsif X(1)='1'. Nemá smysl přidat podmínku i na vyšší prvek, ve stylu elsif X(2)='0' and X(1)='1' then , neboť při X(2)='1' výstupy určuje vyšší multiplexor v kaskádě, viz předešlý Obrázek 1.

Při psaní kombinačních obvodů dbáme na to, že příkaz if then elsif se bude během první fáze překladu konvertovat na jednu či více kaskád dvouvstupových multiplexorů, a tak nutně potřebuje definované přiřazení nové hodnoty na všech svých cestách. Opomineme-li, vzniknou nám prvky typu latch (úrovňové sekvenční obvody) velmi striktně zakázané v FPGA, jelikož vedou na nestabilní chování.

```
process(X)-- the process now results in an erroneous circuit
 begin
   if X(2)='1' then Y<="100";  W<= X(1) or X(0);
   elsif X(1)='1' then Y<="010";  W<= X(0);
   else Y<="00"&X(0); --W<='0';-- the missing assignment ! :-(
   end if;
 end process;
```

Poznámka: V první dílu jsme se o latches *jenom letmo zmínili v kapitole „6.6.b Fatální chyba: Opakované přiřazení signálu ve* for generate*", jelikož souběžný popis nutí ke konstrukcím, jimiž se* latches *nevytvoří tak snadno na rozdíl od sekvenčních příkazů, u nichž stačí nepatrné opomenutí. Pojednáme o nich podrobně v kapitole 0 na straně 33. Zatím si jen pamatujte, že ve zprávách překladače se nesmí objevit hláška obsahující slovo* latch *nebo* loop*, což si snadno ověříme pomocí filtru zpráv.*



Obrázek 2 - Loop a Latch ve zprávách o překladu

Příkaz if then elsif dovolí i jinou konstrukci, využijeme-li nové možnosti, že v sekvenční doméně kódu lze souběžným přiřazením <= vícenásobně zadat hodnoty témuž signálu. Jde o výhodu, oproti souběžné doméně kódu, v níž se tohle zakazuje, jak jsme zdůraznili v prvním dílu v kapitole 2.3.

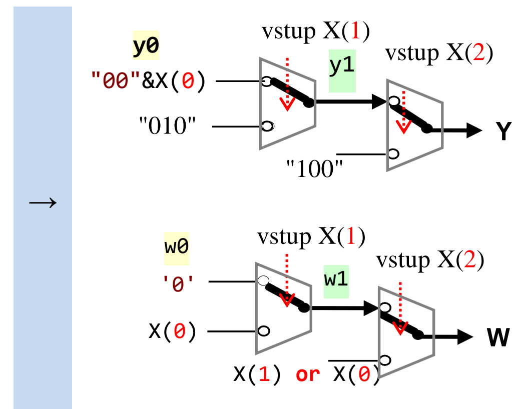
Sekvenční kód používající vícenásobné přiřazení hodnoty může vypadat třeba takto:

```
architecture beh_wsm of Inhibit3 is
begin
   process(X)
   begin
       Y <="00"&X(0); W<='0';
      if X(1)='1' then Y <="010"; W <=X(0);  end if;
      if X(2)='1' then Y <="100"; W <=X(1) or X(0); end if;
   end process;
end architecture;
```

Opakované zápisy do signálů nelze ale implementovat, a tak je překladač musí nahradit. Zavede si místo nich pomocné signály, čím může vytvořit něco v podobného kódu[3]:

```
architecture beh_wm of Inhibit3 is
signal y0, y1 : std_logic_vector(Y'RANGE);
signal w0, w1 : std_logic;
begin
   y0<="00"&X(0);
   y1<="010" when X(1)='1' else y0;
   Y<= "100" when X(2)='1' else y1;

   w0<='0';
   w1<=X(0) when X(1)='1'  else w0;
   W<= X(1) or X(0) when X(2)='1' else w1;

end architecture;
```

→



Obrázek 3 - Naznačení překladu přepisovaných signálů

Pořadí priorit je tentokrát opačné, začínáme od nejnižší a postupujeme k vyšší, dle pravidla následující přepíše předchozí, a tak jeho akce překryje předchozí zápisy.

Demonstrovali jsme právě významný fakt opakovaného přepisu signálu.

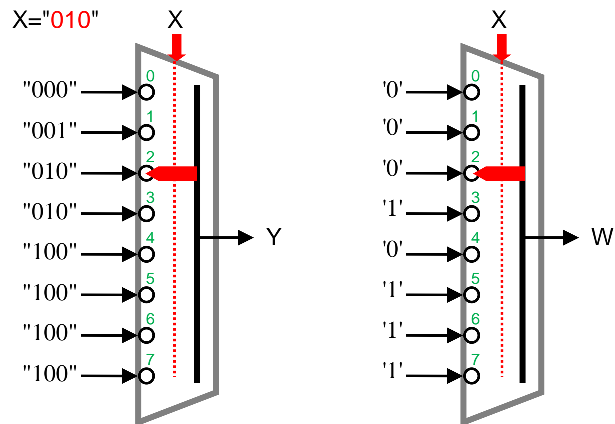
- Když se prvním příkazem W<='0'; přiřadila hodnota v architektuře beh_wsm, ve skutečnosti se operace převedla na w0<='0'; s vnitřním signálem překladače w0, jehož název nám není dostupný. U W<=X(0); v dalším příkazu if X(1)='1' then se zas přiřazení konvertuje na zápis do signálu w1. Čtení hodnoty W ale vrací pokaždé jen finální výsledek přiřazený až posledním příkazem W<= X(1) or X(0) when X(2)='1' else w1;
- V souběžné doméně lze číst hodnotu zapsanou do signálu, ale v té se všechny příkazy vykonávají najednou.
- **Sekvenční doména** se teprve bude převádět na souběžné konstrukce. Signálům v ní lze sice přiřadit vícekrát hodnoty, ale mezivýsledky nejsou dostupné. Nechceme-li vytvořit nesrozumitelný popis s obtížně předvídatelnou funkcí, pak **nečtěme hodnotu signálu, jemuž se na nějaké sekvenční cestě už přiřadilo něco nového.**
- V sekvenční doméně lze však definovat typy variable, které vrací mezivýsledky. Uvedeme je v kapitole 2.1.c.

Obvod jsme nezkomplikovali, neboť obě architektury vedou v prvním kroku překladu na shodný výsledek, akorát if then elsif architektuty beh_ws buduje kaskádu dvouvstupových multiplexorů počínaje vstupem s **nejvyšší** prioritou, zatímco opakované if then příkazy v beh_wsm ji řetězí od vstupu s **nejnižší** prioritou. Námitku vůči architektuře beh_wsm lze tak vznést jedině z hlediska její názornosti. Použijeme-li několik if then příkazů za sebou, není na první pohled zřetelné, že popisujeme prioritní úlohu, zatímco zápis s if then elsif na ni ukazuje naprosto zřetelně.

[3] *Identifikátory vytvářené překladačem obsahují uživateli nepovolený znak ~, aby se zaručila jejich unikátnost.*

2.1.b Multiplexor

Naše úloha je sice prioritní, ale natolik jednoduchá, že se dá bez většího přemýšlení napsat i multiplexorem, což je přímé namodelování pravdivostní tabulky Tabulka 1 uvedené na straně 7. V souběžné doméně popíšeme inhibitor kódem specifikujícím vložení dvou multiplexorů, pro Y a W:



Obrázek 4 - Modelování inhibitoru multiplexorem

```
architecture beh_c of Inhibit3 is
begin
   with X select
       Y <= "100" when "100" | "101" | "110" | "111",
         "010" when "010" | "011",
         "00"&X(0) when others;
   with X select
       W <= '1' when "110" | "101" | "011" | "111",
        '0' when others;
end architecture;
```

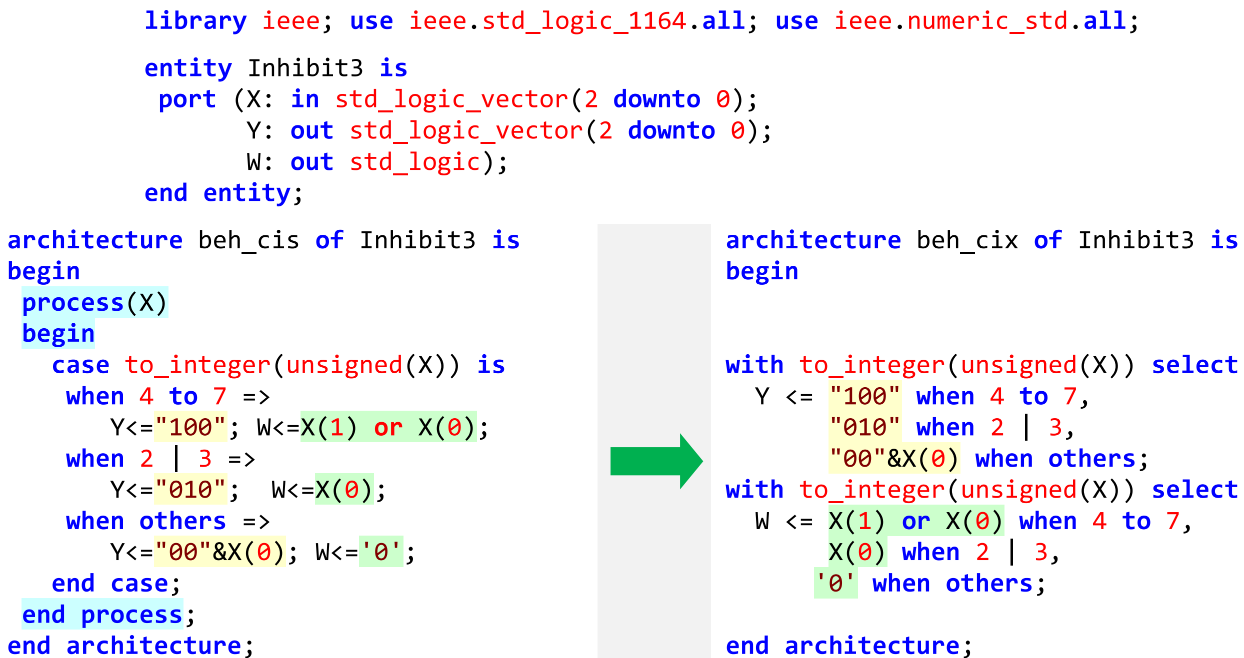
Vysvětlivky ke kódu:

- Vstupy adres X="000" a X="001" jsme sloučili do others a nejnižšímu bitu Y přiřazujeme X(0).
- Použití others je zcela nezbytné kvůli vstupu X, který je definovaný jako tříčlenný vektor, ale s hodnotami danými 9-úrovňovou std_logic, viz první díl. X může tak nabývat 9*9*9 =729 hodnot.
- V seznamu podmínek when jsme nepoužili when "100" to "111", což by Quartus sice přijal, ale dostali bychom chybové hlášení v ModelSim, jelikož operace je ve VHDL definovaná pouze ke skalárním hodnotám. Zápis rozsahu by tak znamenal nevhodnou konstrukci závislou na překladači.

Chceme-li rozsah použít správně syntakticky, musíme v prvním with select převést X na integer.

```
architecture beh_ci of Inhibit3 is
begin
   with to_integer(unsigned(X)) select
    Y <= "100" when 4 to 7,  -- 100" | "101" | "110" | "111"
         "010" when 2 | 3,
         "00"&X(0) when others;

   with X select
    W <= '1' when "110" | "101" | "011" | "111",
         '0' when others;
end architecture;
```

Multiplexor popisovaný with select v souběžné doméně má svou přímou sekvenční analogii v konstrukci case. Opět uvedeme, co mu odpovídá v souběžné části jako další architekturu, jejíž kód by mohl vypadat takto:



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

entity Inhibit3 is
 port (X: in std_logic_vector(2 downto 0);
       Y: out std_logic_vector(2 downto 0);
       W: out std_logic);
end entity;
```

```
architecture beh_cis of Inhibit3 is
begin
 process(X)
 begin
   case to_integer(unsigned(X)) is
    when 4 to 7 =>
       Y<="100"; W<=X(1) or X(0);
    when 2 | 3 =>
       Y<="010";  W<=X(0);
    when others =>
       Y<="00"&X(0); W<='0';
   end case;
 end process;
end architecture;
```

```
architecture beh_cix of Inhibit3 is
begin


with to_integer(unsigned(X)) select
  Y <= "100" when 4 to 7,
       "010" when 2 | 3,
       "00"&X(0) when others;
with to_integer(unsigned(X)) select
  W <= X(1) or X(0) when 4 to 7,
       X(0) when 2 | 3,
      '0' when others;

end architecture;
```

Zatímco with select dovoluje měnit pouze jeden signál, tak příkaz case nabízí výhodu napsání skupiny operací, klidně i vložení dalšího case či if then else.

Opět si musíme dávat pozor, píšeme-li kombinační obvod, aby se na všech cestách přiřazovaly hodnoty, jinak se opět vloží nechtěný paměťový prvek typu latch. Důvod, proč se tohle děje, si necháme na později. Zatím jen znovu zdůrazníme, že **latch-škůdce se nesmí vytvořit logickými elementy FPGA obvodů**.

Zde ještě připomínáme, že i operace s indexy se většinou rovněž převádí na multiplexory během prvního kroku překladu, a tak jsme mohli také psát:

```
architecture beh_horror of Inhibit3 is
type mux_t is array (0 to 7) of std_logic_vector(2 downto 0);
constant muxy : mux_t :=("000","001","010","010","100","100","100","100");
constant muxw : std_logic_vector(0 to 7):="00010111";
begin
   Y <= muxy(to_integer(unsigned(X)));
   W <= muxw(to_integer(unsigned(X)));
end architecture;
```

Zde bude úplně jedno, zda umístíme přiřazení Y a W do sekvenční či souběžné domény zdrojového kódu. V obou budou obě vypadat stejně, v sekvenční se kolem nich vyskytne jen process(X) end process;

Budiž obvod! Popíšeme-li jeho zapojení výčtovým stylem a aspoň u našeho jednoduchého obvodu dostaneme též optimální výsledek. (Pozor, obecně to však neplatí, například už u sčítačky!) Výsledná architektura bude však vskutku hororová, jak ostatně nese ve svém názvu. Nevyužívá možnosti jazyka, což bychom ji mohli ještě blahosklonně odpustit, ale nejvíc ze všeho postrádá názornost a tuhle herezi již nelze kódu prominout. Uvidíte-li takový zápis, nepoznáte z něho na první pohled, co se vlastně provádí.

2.1.c Logické rovnice

Logické rovnice se píší stejně v obou doménách zdrojového kódu a dávají také možnost jak vytvořit variabilní obvod s různou šířkou vektorů vstupů a výstupů. VHDL jazyk zahrnuje sice cykly, ale nelze jimi přidávat části příkazů, pouze celé příkazy. Lze sice využít několik if then příkazů za sebou, ale dobře sestavené logické rovnice poskytují efektivnější výsledky, protože napovídají i strukturu výsledku.

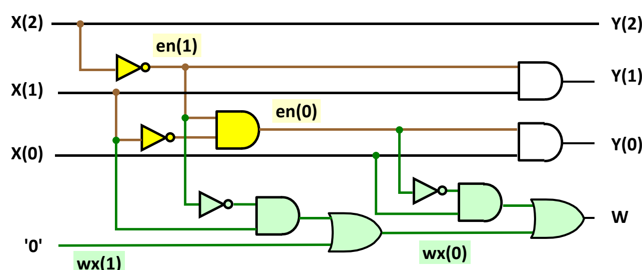
Napíšeme-li jimi náš inhibitor, pak první aproximaci řešení vytvoříme snadno v souběžné doméně.
V sekvenční doméně by zde jen přibylo process end process; a rovnice by se nezměnily.

```
architecture beh_a of Inhibit3 is
begin
   Y(2) <= X(2); Y(1) <= not X(2) and X(1); Y(0) <= not X(2) and not X(1) and X(0);
   W <= (X(2) and X(1)) or (X(2) and X(0)) or (X(1) and X(0));
end architecture;
```

Zápis však nijak nenaznačuje možnost rozšíření v případě širších vektorů X a Y. Upravíme je na rekurzi, kvůli níž si vytvoříme si pomocné vektory, abychom dodrželi pravidlo jediného zápisu do signálu.

```
 architecture beh_ag of Inhibit3 is
signal en,wx:std_logic_vector(1 downto 0);
begin
  Y(2) <= X(2); en(1)<=not X(2); wx(1)<='0'; -- initialization
  ---
  Y(1) <= en(1) and X(1);                    -- recursion
  en(0)<= en(1) and not X(1); wx(0)<= wx(1) or (not en(1) and X(1));
  ---
  Y(0) <= en(0) and X(0);                    -- final step
  W<=wx(0) or (not en(0) and X(0));
end architecture;
```

Pomocný vektor en (enable) uvolňuje posílání vstupu X(i) na Y(i) výstup. Prvek en(i) bude v '1' tak dlouho, dokud se nepropustí první X(i)='1', pak se en(j) s nižšími indexy (j<i) již bude '0' a méně prioritní Y nastaví na '0' výstupy and hradel Y(1) a Y(0). Druhý signál, vektor wx, naopak bude akumulovat na hradle or informaci, že už došlo k zablokování nějakého Y(j), tedy en(j) bylo již v '0' a X(j)='1'.



V sekvenční doméně zdrojového kód můžeme využít opakované přepisování, **avšak nikoli signálů**!!! Důvod jsme již naznačili v orámované poznámce u Obrázek 3 na straně 10. Využijeme však možnosti definovat si variable, která na rozdíl od signálu vrací při své čtení přiřazené mezihodnoty.

```
architecture beh_ags_OK of Inhibit3 is
begin
   process(X)
   variable en, wx:std_logic; -- If en and wx were signals, the circuit would not work !!!
   begin
     Y(2) <= X(2); en:=not X(2); wx:='0';
     Y(1) <= en and X(1); wx:=wx or (not en and X(1)); en:=en and not X(1);
     Y(0) <= en and X(0); W<=wx or (not en and X(0));
   end process;
end architecture;
```

Platí zásada, že v **procesech upřednostňujeme variable**. Signály využijeme jedině ke spojení obvodu popsaného uvnitř procesu s okolím. Ze signálů si jen čteme vstupní hodnoty a v závěru jim přiřadíme výstupní hodnoty, neboť signály představují jedinou rozumnou možnost jak předat výsledky dál.
U variable **záleží na pořadí příkazů**. Všimněte si, že **wx se aktualizuje před en** [4].

Opakované zápisy do variable se během překladu opět transformují s použitím definic pomocných signálů, ale tentokát se při jejich čtení vracejí mezihodnoty, čímž se myslí hodnota do nich naposledy uložená.

[4] Ve cvičném příkladu 4 z prvního dílu, vyřešeném v kapitole 8.4, se zmiňovalo, že lze prohodit pořadí souběžných operací bez změny funkce. V sekvenční části to nelze, tedy analogicky s klasickými programy, třeba v jazyce C.

Přechozí kód dovoluje již napsat příkaz for loop, sekvenční obdobu souběžného for generate probíraného v závěru prvního dílu učebnice. Zde se sice v závěru provedou nějaká přiřazení navíc, ale vše přebytečné se zminimalizuje, a tak napíšeme smyčku optimalizovanou hlavně na počet řádek:

```
architecture beh_forlopp of Inhibit3 is
begin
   process(X)
   variable en, wx : std_logic;
   begin
       en:='1'; wx:='0'; -- correct initialization inside begin end process
   igen:for i in 2 downto 0 loop
       Y(i) <= en and X(i);
       wx := wx or (not en and X(i));
       en:= en and not X(i);
     end loop;
     W<=wx;
   end process;
end architecture;
```

Zde zdůrazníme, že **inicializace proměnných se musí provádést příkazy mezi** begin a end process. Přímo u definic se inicializace sice rovněž používají, ale především kvůli simulaci, jelikož vyjadřují jedině stav po zapnutí napájení. Poté již nemají vliv na činnost a proces se dále chová, jako kdyby vůbec neexistovaly. Důvod vyplyne z jeho vlastností, jimž věnujeme samostatnou pozdější kapitolu.

Poznámka: Pravidlo se týká jen variable. *Konstanty samozřejmě inicializuje přímo u jejich definic.*

2.1.d Rozšíření na obecný inhibitor

Náš pracně získaný kód zapsaný pomocí logických rovnic lze již snadno rozšířit na univerzální obvod, u něhož si generic parametrem HBIT (index nejvyššího bitu) volíme šířku X a Y vektorů. Obvod můžeme popsat následujícím kódem, v němž využijeme i pasivní proces v entitě, podrobně popsaný v prvním dílu, který se provádí výhradně v době překladu a slouží k testování korektních hodnot parametrů generic:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity InhibitG is
   generic(  HBIT:integer :=7);
   port(X: in std_logic_vector(HBIT downto 0);
       Y: out std_logic_vector(HBIT downto 0);
       W: out std_logic);
   begin -- passive process. It is not synthetized, but executed only in the compilation
     assert HBIT>1 report "Required HBIT>1" severity failure;
end entity;

architecture beh_forlopp of InhibitG is
begin
   process(X)
   variable en, wx: std_logic;
   begin
     en:='1'; wx:='0'; -- correct initializations inside begin end process
ilp: for i in Y'RANGE loop
       Y(i) <= en and X(i);
       wx := wx or (not en and X(i)); en:= en and not X(i);
     end loop;
     W<=wx;
   end process;
end architecture;
```

Program 1 - Generic inhibitor

Zahrnutí úvodního kroku s indexem HBIT do cyklu nemá jakýkoli vliv na výsledný obvod, v obou případech se správně minimalizuje.

Pro zajímavost uvedeme ještě soubežnou analogii, na níž se bude Program 1 převádět. Vznikne zapojení analogické kódu s for generate opakovaným vkládáním příkazů. V obvodu to jinak ani nelze.

```
architecture beh_forgen of InhibitG is
signal en, wx : std_logic_vector(HBIT-1 downto 0);
begin
     Y(HBIT) <= X(HBIT); en(HBIT-1)<=not X(HBIT); wx(HBIT-1)<='0'; --initialization
  igen:for i in HBIT-1 downto 1 generate
        Y(i) <= en(i) and X(i);                    --recursive loop
        wx(i-1)<=wx(i) or (not en(i) and X(i));
        en(i-1)<= en(i) and not X(i);
     end generate;
     W<=wx(0); Y(0) <= en(0) and X(0); -- the last step
end architecture;
```

Souběžný kód potřeboval cyklus rozdělit na inicializaci, vlastní tělo a závěrečný krok, a přidat ještě definice en a wx vektorů kvůli splnění jediného přiřazení do signálu. V předchozím sekvenčním kódu se pomocné prvky vytvořily automaticky a nadbytečné zápisy se zrušily minimalizací, což nám ulehčilo návrh.

2.1.e Naprogramovaná obecná hrůza

Přímým převodem klasických programů v C či Java lze vytvořit řadu hrůz, z nichž vzniká příliš složité RTL Schéma během úvodní konverze. Veškerá námaha se pak přesune do dalších optimalizačních kroků, které obecně představují složité úlohy. Někdy se ani nenajde minimální řešení, jen se stanoví nějaké pseudo-optimální. Z komplikovaného vstupního popisu mohou tak vzniknou nepěkné výsledky.

Odsuzovali jsme přímé programové konverze, ale neuškodí si jednu udělat, abychom viděli důvody. Zkušenější programátor by na začátku mohl testovací C kód vytvořit třeba takto:

```
int main()
{  typedef unsigned char byte;
   byte X, Y, W;
   for (X = 0; X < 255; X++) // debug input
   {  /************************************/
      Y = 0; W = 0;
      if (X != 0)
      {  byte mask = 0x80;
         do
         {  if ((X&mask) != 0) {Y = mask; W = (~mask & X) ? 1 : 0;
                    break; }
            mask = mask>>1;
         } while (mask != 0)
      } /***********************************/
      printf("%X %d; ", Y, W);
   }
   return 0;
}
```

Lze podle něho sice vytvořit VHDL popis obvodu, ale s exponenciálním nárůstem složitosti. Důvody:

- Použití mask k získání hodnoty bitu je ryze programátorská konstrukce. V jazycích C a ani Java ani jinou možnost nemáme, ale v obvodu ano, u nich jde o primitivní operaci, vyvedeme si vodič.
- V obvodu nelze běhat ve smyčce. Ta se konvertuje opakovaným vkládáním celého těla cyklu, takže musí v době překladu existovat **jasné maximum počtu iterací** nezávislé na vstupních datech, což zde splněno, ale obecně se while smyčky transformují na zapojení hůře než zcela jasné for loop.
- Vstupní test na X==0 sice zrychlí běh kódu, ale jen v případě nulového X, tedy v 0,4 % případů z 256 možných hodnot X, takže zpomalí ostatní výpočty. V obvodu bývají testy na rovnost navíc často náročné na spotřebu logiky. Vyžadují úplný minterm. A přeskočením části kódu se obvod nijak nezmenší. Vždy se v něm zapojí všechny možné cesty smyčkou a také se naráz vyhodnocují, pouze se vybírá výsledek, který se pošle dál.

Zapracujeme-li námitky do C kódu a využijeme doporučený for cyklus. Místo masky si vytvoříme pomocné funkce, jimiž substituujeme chybějící testy bitů, které sice obsahuje VHDL, ale ne C a Java.

```
typedef unsigned char byte;
byte bit(int index) { return index<=0 ? 1 : (byte)(1u << index); }
int testbit(byte X, int index) { return (X & bit(index))!=0; }
int main()
{  byte X, Y, W;
   for (X = 0; X < 255; X++) // debug input
   {  /*************************************/
      Y = 0; W = 0;
      int flag = 1;
      for (int i = 7; i>=0; i--)
      { if (testbit(X, i))  {  if (flag) { Y = bit(i); flag = 0; }
                               else { W = 1; break; }
                            }
      } /*********************************/
      printf("%X %d; ", Y, W);
   }
   return 0;
}
```

Tělo mezi /***/ komentáři vytvoří již přijatelnější popis obvodu, který vykazuje pomalejší nárůst složitosti se zvětšování HBIT:

```
architecture beh_clumsy_program of InhibitG is
begin
   process(X)
   variable flag:boolean;
   begin
      Y<=(others=>'0'); W<='0'; flag:=true;
igen:for i in X'RANGE loop
         if X(i)='1' then
            if flag then Y(i)<='1'; flag:=false; else W<='1'; exit;
            end if;
         end if;
      end loop;
   end process;
end architecture;
```

VHDL kód ale nijak nenaznačuje vyřešení prioritní úlohy řetězem logických operací, což vede na komplikovanější úvodní meta-schéma RTL Viewer. To se sice dále optimalizuje, ale minimum se nalezne jen u menších obvodů. Složitější zjednodušování se totiž zastaví na nějakém náhodně nalezeném pseudo-optimu, což dokazuje i tabulka porovnání `beh_clumsy_program` a architektury `beh_forlopp` ze str. 14, která má téměř lineární růst složitosti.

	Parametr HBIT=	3	7	11	15	23	31	63	127
Počet použitých LE [5] (Logických elementů)	beh_forlopp	4	13	21	29	45	63	129	280
	beh_clumsy_program [6]	4	13	22	30	47	71	161	348

Náš relativně jednoduchý výukový příklad tak demonstroval, několik důležitých faktů, které uvedeme v orámovaném souhrnu:

---

[5] Uvedené počty logických elementů LE se mohou lehce měnit dle nastavení konkrétních podmínek překladače a nemusí nutně pokaždé vyjít stejně, neboť hledání skončí na nějakém optimu závislém i na rozmístění do prvků použitého FPGA, což pochopitelně ovlivní i na dalších obvodech, které do něj vkládáme v celkovém zapojení.

[6] U komplikovanějších popisů by nastaly problémy už dříve. Obvod stvořený převodem prvního C programu s mask a do while roste exponenciálně a optimalizuje se jen v případě malých HBIT. Při HBIT=11 má již 32 LE, při HBIT=23 již 95 LE a na HBIT=31 pak 150 LE.

- Píšeme-li malý a jednoduchý obvod, lze ho prakticky vytvořit jakkoli. Dbáme jen na přehlednost popisu a definici správné funkce. Komplikovanějším kódem sice přesuneme hlavní úsilí do dalších optimalizačních kroků, avšak u menších struktur se zpravidla podaří i tak najít minimální výsledek.
- Řešíme-li složitější úlohu, pak se při psaní VHDL kódu hodí přemýšlet o výsledné implementaci. Když naznačíme překladači i vhodnou strukturu obvodu, dostaneme příznivější zapojení.
- Lze sice napodobit techniky z programovacích jazyků, ovšem pouze jejich omezenou množinu, ale i tak nemusí řešení vést na optimální výsledek, jak demonstroval předchozí příklad.

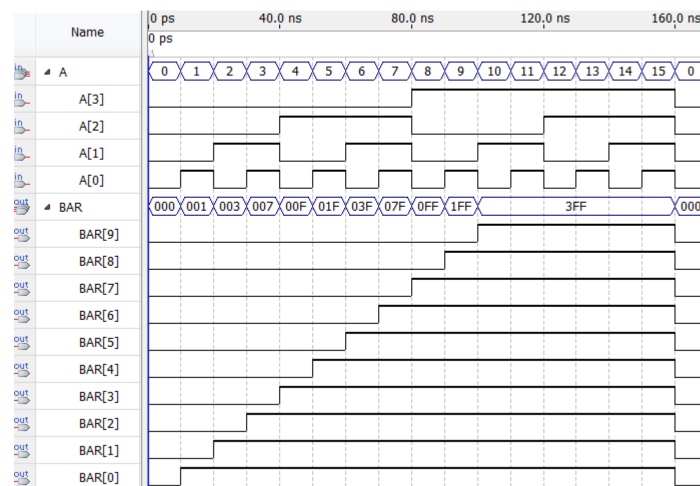
2.2 *** Cvičná úloha 1: - Obecný lineární ukazatel

Zadání: Navrhněte dekodér pro lineární ukazatel hodnoty (angl. bar indicator). Výstup ve formě vektoru BAR(HBIT_BAR downto 0) má tolik bitů v logické ´1´, kolik určuje A(HBIT_A donwto 0) brané jako číslo bez znaménka.

Zkuste vymyslet variabilní verzi, jejíž entita by mohla vypadat následovně (jen by chtěla doplnit o pasivní proces, v němž se otestují parametry):

```
entity unsigned2barG is
   generic(HBIT_A:integer :=3;
           HBIT_BAR:integer :=9);
  port (A: in std_logic_vector(HBIT_A downto 0);
        BAR: out std_logic_vector(HBIT_BAR downto 0));
end entity;
```

Činnost při volbě hodnot HBIT_A=3 a HBIT_BAR=9 naznačuje záznam simulátoru:



Úloha se již řešila v prvním dílu, ale používaly se vždy pevné velikosti vstupních vektorů. V kapitole 3 prvního dílu se modelovala příkazem with select a v kapitole 3.4 se její větší verze vytvořila čtením z pole. Obě předchozí řešení ale nelze vhodně rozšiřovat. V souběžné doméně zdrojového kódu by se variabilní řešení dalo napsat, ale vyšlo by dost nepřehledně, a tak se raději opominulo.

Nápověda: Využijte sekvenční zdrojovou doménu a proměnnou variable shodnou typem s BAR. Můžete do ní postupně nasouvat bity v '1'. Počet doplněných '1' bude záviset od váhy jednotkového bitu ve vstupu A. K přidání '1' nemusíte ani využít další vnořený for cyklus, úplně postačí operátor & sjednocení dvou vektorů, z nichž pravý bude část šikovné konstanty obsahující samé '1'.

Výsledný proces vyjde poměrně jednoduše a ve vzorovém řešení vzadu obsahuje jen 18 řádek. Zkuste ho ale napřed vykoumat sami, než nalistujete dozadu.

2.3 Více o více architekturách

V předchozí části jsme používali více architektur. Ukažme si jejich volbu na jednoduchém výukovém příkladu obvodu xgate, který vytvoříme třemi architekturami, a to se zcela odlišnou funkcí, abychom si ověřili, že nám vše správně funguje.

```
library ieee; use ieee.std_logic_1164.all;
entity xgate is port (A,B: in std_logic; X: out std_logic);
end entity;
architecture gxor of xgate is begin  X<=A xor B; end;
architecture gor of xgate is  begin  X<=A or B;  end;
architecture gand of xgate is begin  X<=A and B; end;
```

Změna užité architektury se vždy provádí před překladem zdrojového kódu a přepíšeme pouze její název, což znamená flexibilnější úpravu než zakomentování nechtěných řádek. K její volbě musíme však nezbytně využít příkaz map, který se v prvním dílu učebnice probíral až v kapitole 6 nazvané „VHDL stylem Structural“, a tak jsme se ke zjednodušení omezili na jedinou architekturu u každé entity.

Poznámka: Schéma BDF schéma (Block Diagram Schematic File) a VWF (Vector Waveform File) simulace nenabízí možnost volby, vždy se použije poslední definovaná architektura, v našem případě gand.

Existují dva způsoby zadání užité architektury. Napřed si ukážeme její přímočařejší specifikaci přímo během vytváření instance příkazem map.

```
library ieee; use ieee.std_logic_1164.all; library work;
entity xgate_test is
   port(A, B : in std_logic; X: out std_logic_vector(1 to 6));
end entity;
architecture arch0 of xgate_test is
component xgate is port (A,B: in std_logic; X: out std_logic);
end component;
begin
-- the usage of keyword notations (cz:jmenne asociace)
inst1 : entity work.xgate(gxor) port map(A => A, B=>B, X=>X(1));
-- the alternative usage of positional notations (cz:pozicni asociace)
inst2 : entity work.xgate(gor) port map(A,B,X(2));
inst3 : entity work.xgate(gand) port map(A,B,X(3));
-- no architecture specification - the last one is used, i.e. gand
inst4 : xgate port map(A,B,X(4));
inst5 : component xgate port map(A,B,X(5));
inst6 : entity work.xgate port map(A,B,X(6));
end architecture;
```

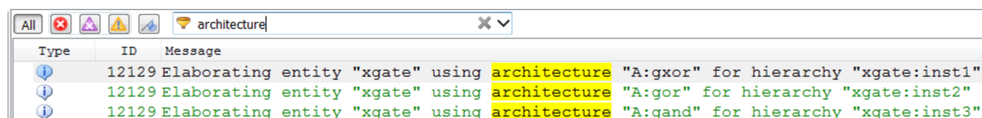
Všimněme si zde napřed dolních map definic, inst4 až inst6, v nichž se nespecifikuje architektura, takže se použije vždy poslední ve zdrojovém kódu, tedy architektura gand.

- Řádka s inst4 odpovídá jednoduchému zápisu, který jsme používali v první části.
- Následující definice inst5 ji rozšiřuje o uvedení nepovinné specifikace component, kterou překladač doplnil automaticky u inst4 coby nasměrování, že následující název xgate patří mezi vložené component reference v právě překládaném souboru.
- Vytvoření inst6 instance obvodu xgate odkazuje přímo na entitu, a tak se musí uvést plná cestou k ní. Bude se hledat v knihovně work, do níž se překládají soubory projektu, a z té se vybere entita xgate. *Poznámka: Kdyby se komentováním vyřadily z překladu řádky inst1 až inst5 instancí a zůstala by jen inst6 instance, pak by se nemusela ani vkládat component hlavička xgate. Kvůli přehlednosti a redukci omylů se radí přidat komponenty, na něž odkazujeme, čímž rychle zjistíme i typy jejich parametrů.*

Srovnáme-li inst1 až inst3 s poslední inst6, pak se od ní liší jenom přidáním závorek se specifikací architektury, která se použije. Právě kvůli tomu se referuje přímo na soubor s popisem entity, jelikož překladač musí v něm vybrat předepsanou architekturu.

V příkladu používáme pouze port map, ale za uvedením názvu entity můžeme pokračovat jak generic map, tak port map. V téhle části se nic nemění. Dle libosti lze používat jak jmenné asociace, tak poziční asociace. Rozdíl mezi nimi jsme diskutovali v první části v kapitole 6, v níž jsme uvedli, že jmenné představují bezpečnější způsob, ale v menších souborech, v němž vše leží poblíž sebe, se běžně používají kratší poziční asociace i v profesionálních kódech.

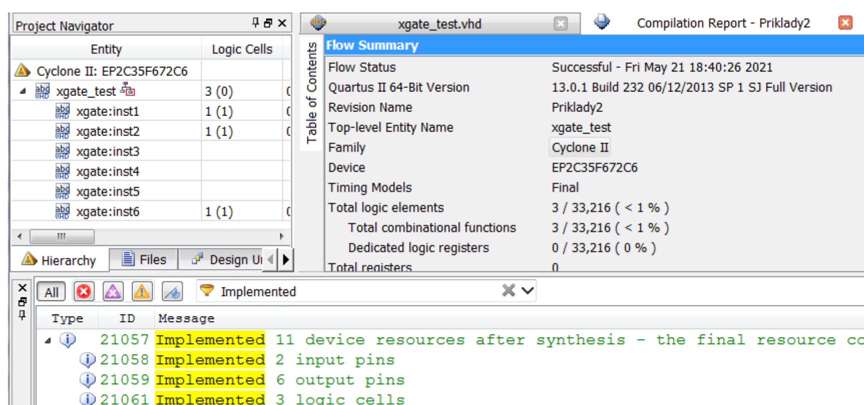
Soubor si můžeme přeložit a překontrolovat volbu vyhledání "architecture" v okně zpráv:



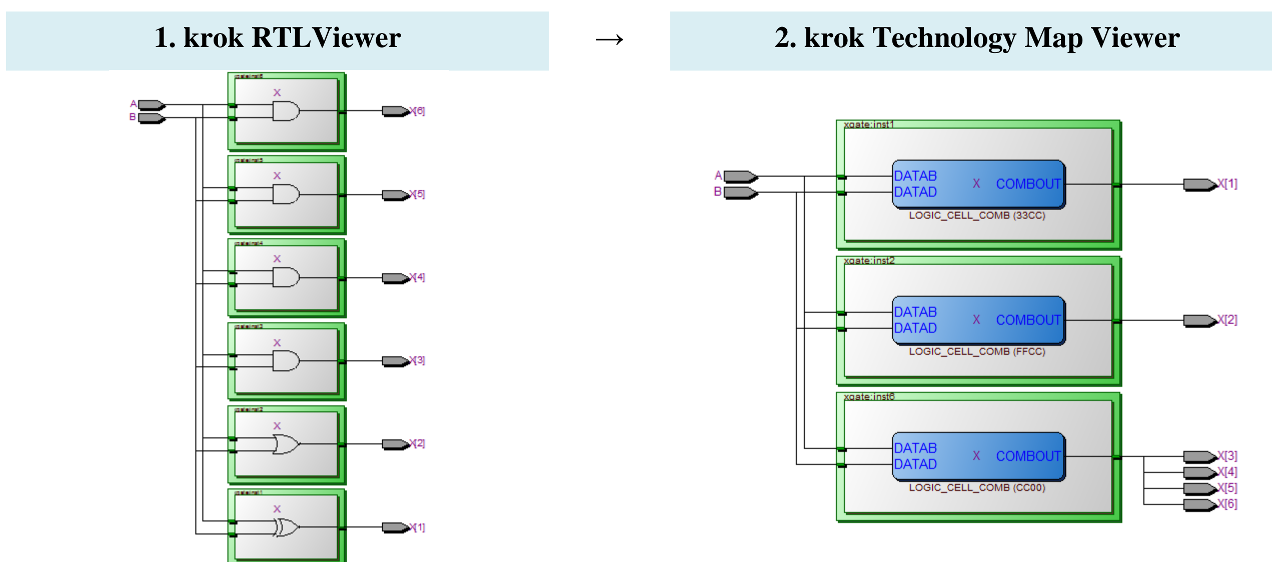
Ve zprávách narazíme i na zvláštnost, a to informační hlášku překladače:

Info (21057): Implemented 11 device resources after synthesis - the final resource count might be different

Odkazuje se na podivný počet použitých elementů, kterých by mělo být nejméně 6, ale máme 3.



Nástroj RTL Viewer známe již z prvního dílu. Zobrazí nám složitější obvod, ale ukazuje jen úvodní dataflow meta-schéma vytvořené ze zdrojového kódu. Záhadu objasní až Technology Map Viewer, výsledek dalšího kroku překladu, během něhož se sloučila čtyři hradla AND provádějící zcela totožnou funkci.



Existuje i alternativní metoda volby použité architektury v map příkazu pomocí speciálního příkazu config, jímž popíšeme konfiguraci, jaká architektura se pro danou instanci použije. Soustředíme v něm změny na

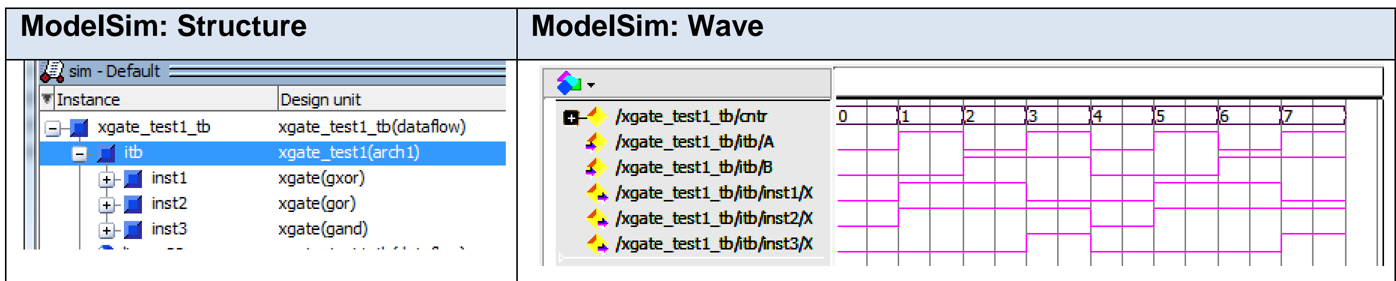
jedno místo a lze definovat i víc různých konfigurací, což dovolí flexibilně měnit strukturu testů. Konfiguraci umístíme ho do samostatného VHDL souboru xgate_test1_cfg.vhd:

```
-- file: xgate_test1.vhd ---------------------------------------

library ieee; use ieee.std_logic_1164.all; library work;
entity xgate_test1 is
   port(A, B : in  std_logic; X1, X2, X3: out std_logic);
end entity;
architecture arch1 of xgate_test1 is
   component xgate is
     port (A,B: in std_logic; X: out std_logic);
   end component;
begin
     -- the usage of keyword notations (cz:jmenne asociace)
inst1 : xgate port map(A => A, B=>B, X=>X1);

       -- the alternative usage of positional notations (cz:pozicni asociace)
inst2 : xgate port map(A,B,X2);
inst3 : xgate port map(A,B,X3);
end architecture;

-- file: xgate_test1_cfg.vhd -----------------------------------

library work; use work.all;
configuration xgate_test1_cfg of xgate_test1 is
    for arch1
      for inst1 : xgate  use entity work.xgate(gxor);
      end for;
      for inst2 : xgate  use entity work.xgate(gor);
      end for;
      for inst3 : xgate  use entity work.xgate(gand);
      end for;
    end for;
end configuration;
```

Poznámka kvůli úplnosti výkladu: Quartus si konfiguraci najde a správně použije, avšak budeme-li vytvářet instanci od xgate_test1 *v ModelSim, tak se v něm nepoužije automaticky, ale musíme její uplatnění specifikovat v architektuře testbench za vloženým příkazem* component*:*

```
-- testbench file: xgate_test1_tb.vhd -----------------------------------

library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all; library work;
entity xgate_test1_tb is end entity;
architecture dataflow of xgate_test1_tb is
   component xgate_test1 is
     port (A,B: in std_logic; X1, X2: out std_logic);
   end component;
   for itb: xgate_test1
       use configuration work.xgate_test1_cfg;
   signal cntr : unsigned(3 downto 0):=(others=>'0'); -- here, the initialization is necessary!!!
   signal X1, X2: std_logic;
   signal BA: std_logic_vector(cntr'RANGE);
begin
   cntr<= cntr+1 after 20 ns; -- we explain after keyword in chapter 2.10 on page 29
   BA<=std_logic_vector(cntr);
  itb:  xgate_test1 port map(A=>BA(0), B=>BA(1), X1=>X1, X2=>X2);

   assert cntr<8 report "End of simulation" severity failure;
end;
```

Přesvědčíme se, zda se zvolily správné architektury v okně Structure prostředí ModelSim, které ukazuje strukturu simulovaného kódu a jména architektur použitých při vytváření instancí simulovaných obvodů:



2.4 Příkaz process

Přesná syntaxe příkazu vypadá takto, kde části v [ ] lze vynechat:

```
[optional_label:] process [(optional sensitivity list)] [is]
    [sequential_declarations]
begin
   sequential statements
end process [optional_label];
```

- *optional_label* identifikátor usnadní orientaci hlavně v simulaci, v syntéze nemá větší význam.
- is klíčové slovo lze přidat od verze VHDL-93, nicméně v praxi se zcela běžně vynechává.
- ***sequential_declarations*** obsahují lokální deklarace prvků dostupné jen v rámci procesu. Seznam může být i prázdný a bude platit i u funkcí a procedur:

(zelená)	➢ type, subtype, constant, variable deklarace známé již z prvního dílu učebnice. ➢ function, procedure - sekvenční domény kódu definující části obvodů k lokálnímu vládání. ➢ use klausule dovoluje přidat další knihovny lokálně dostupné jen zde, jsou-li potřeba. ➢ group a attribute deklarace se často aplikují pospolu. Pomocí group lze specifikovat skupinu objektů, jimž všem se pak pomocí attribute dodá přídavná vlastnost. Z prvního dílu známe předdefinované atributy (vlastnosti) signálů jako RANGE, LENGTH apod. Lze zavést i vlastní zvané "user defined attributes". Jde však o speciální téma, které zmiňujeme jen kvůli úplnosti.
(žlutá)	▪ alias změní pojmenování prvku (opět známe z prvního dílu z části o package). Užití je sice normou povolené i zde, ale **řada nástrojů syntézy obvodu ho tady nepodporuje**. ▪ FILE - soubor lze užít výhradně v kódu určeném **k simulaci**. Nelze ho syntetizovat!
(červená)	❖ V žádném případě **se zde nesmí objevit** deklarace signal nebo component. ❖ **Nelze** zde vytvořit shared variable, ta patří do deklarací architektury a vytvoří sdílený element podobně jako signal, ale mající vlastnosti variable. V kódech určených k syntéze obvodů se nedoporučuje, respektive ho řada profesionálních firem spíš v nich striktně **zakazuje** kvůli obtížné syntéze. Možnost se do VHDL přidala především kvůli simulaci.

Tabulka 2 - Deklarace v process, function a procedure

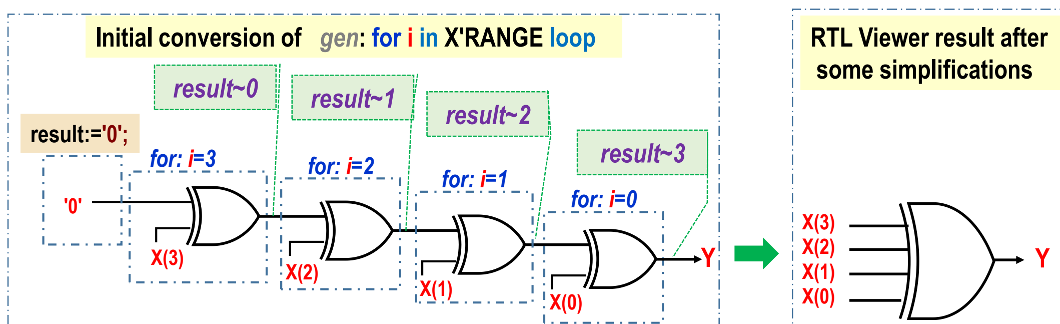
Prvek sensitivity list jsme si nechali na konec výkladu. Udává seznam signálů, jejichž změna vyvolá změnu nějakého výstupního signálu. V kombinačních obvodech odpovídá obvykle všem vstupním signálům, ale mimo ně může obsahovat třeba pouze jediný signál z mnoha vstupů. Třeba u obvodů řízených hodinovým signálem, se často uvede jenom jeho název, jelikož výstupy získají nové hodnoty jen po změně hodin.

Kvůli demonstraci chování procesu si vytvoříme obvod tak zvaného **unárního** xor, které po aplikaci na vektor provede xor všech jeho složek. Jazyk VHDL od své verze VHDL-2008 zná již unární logické ope-

rátory, v něm lze psát něco ve stylu xor X, kde X je vektor. Výsledkem je vícevstupové xor všech složek, analogické operace existují ke všem binárním logických operátorů.

Předpokládejme však, že z nějakého důvodu, třeba kvůli simulátoru, píšeme ve starší verzi VHDL-93, která ještě nezná unární operátory, a tak si vytvoříme naše vlastní unární xor:

```
library ieee; use ieee.std_logic_1164.all;
entity unary_xor is
    generic(HBIT:integer:=3);
    port (X: in std_logic_vector(HBIT downto 0):=(others=>'0');
          Y: out std_logic:='0');
end entity;
architecture beh of unary_xor is
begin
px: process (X) is              -- here, we have also appended optional 'is' that is usally omitted
    variable result:std_logic:='0';
    begin
      result:='0'; -- If we commented out this line, an unstable circuit would be created!!!
 gen: for i in X'RANGE loop result:=result xor X(i);
      end loop;
      Y<=result;
    end process;
end architecture;
```



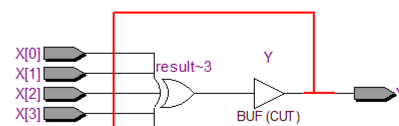
Obrázek 5 - Překlad unárního XOR

Kód smyčky for se překládá opakovaným vkládáním jejího těla, jinak to v obvodu ani nelze. Přepisování hodnoty result se nahrazuje zápisy do pomocných dočasných signálů, jejichž identifikátory obsahují symbol ~ uživateli nepovolený, čímž se zaručí jejich unikátnost.

Definovali jsme result jako variable, jejíž čtení vrací poslední přiřazenou mezihodnotu. Vytvoří se tak kaskáda dvouvstupových hradel xor, která se vzápětí zjednoduší aplikací asociativity logických operací na výsledek Y<=X(3) xor X(2) xor X(1) xor X(0); popisující vícevstupové xor.

Všimněte si, že ve výsledném zapojení se neuplatnily inicializace u deklarací proměnných a signálů, které jsme z cvičných důvodů napsali všude, kde se jen dalo, dokonce i v entitě. Ty se přidávají kvůli simulaci. V obvodu nelze inicializovat vodič, pouze zdroj, který do něho posílá signál. Překladač se sice vynasnaží nám vyhovět, ale řadu inicializací odmítne jako neproveditelné či konfliktní s prioritnějšími nastaveními.

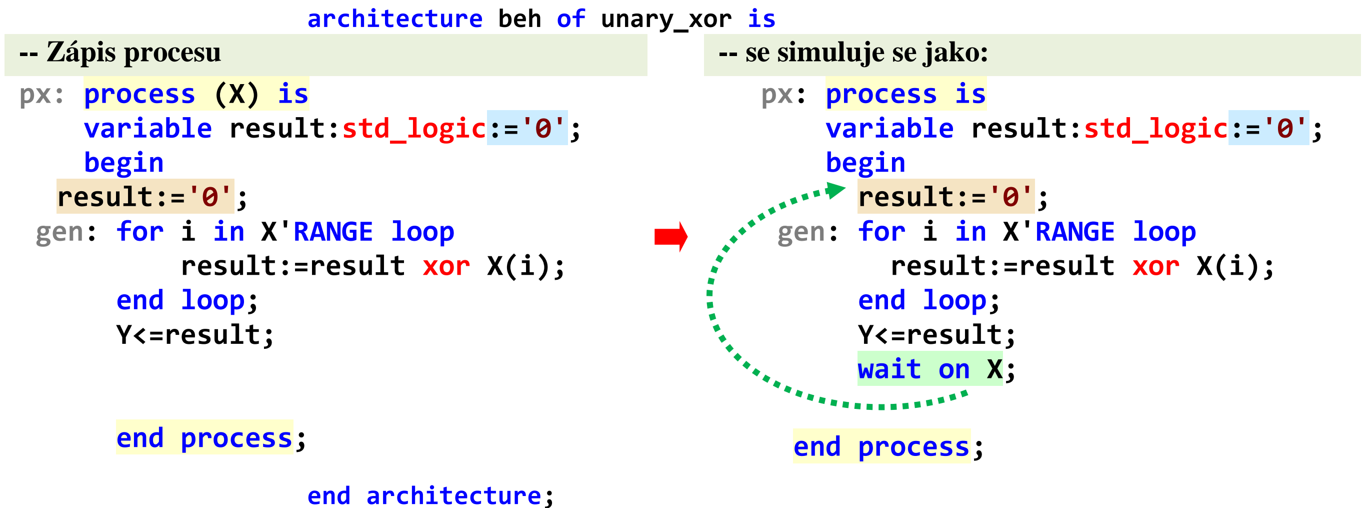
Výhradně inicializace result:='0' uvnitř begin end těla procesu ovlivnila výsledné zapojení. Kdyby se tenhle příkaz vynechal a zůstala by jen inicializace u deklarace **variable result:std_logic:='0';** pak by se vytvořil obvod s nestabilní smyčkou, která by se občas rozkmitala jako oscilátor, viz obrázek vpravo, neboť absencí zadání výchozí hodnoty result jsme předepsali překladači, aby se poslední hodnota přiřazená proměnné result znovu použila jako vstup.



Obrázek 6 - Výsledek opomenutí inicializace result:='0'; v kódu procesu

V simulačním prostředí se proces mající sensitivity list emuluje jako nekonečná smyčka, na jejímž konci existuje příkaz wait on X; , který znamená čekání na změnu signálu. Místo X zde může rovněž být seznam signálů oddělený čárkami, stejně jako v sensitivity list.

Můžeme tedy sensitivity list pokládat za přímý ekvivalent příkazu wait on, ale přemístěného do hlavičky. Zápis se tím zpřehlednil především u předlouhých kódů, protože není nutné listovat až na konec procesu, abychom zjistili, na čem závisí výstupy.



Obrázek 7 - Process se sensitivity list a jeho simulace

Quartus nedovoluje použití příkazu wait on. Kód na obrázku vpravo, ač je syntakticky zcela správný, se provede jen v simulátoru. Z obrázku však vyplývá, že **sensitivity list by měl zahrnovat všechny potřebné signály**, protože simulátor bude pokaždé čekat jen na změnu signálů, kterou jsme mu předepsali. Vynecháme-li něco, může probíhat odlišně od obvodu.

Quartus si chybějící signály zpravidla doplní, ovšem bez záruky. Pokud vypátrá, že nějaký chybí, uvede varování typu: *Signal "X" is read inside the Process Statement but isn't in the Process Statement's sensitivity list*. Doporučuje se minimálně před testováním obvodu si vyhledat slovo "sensitivity" ve zprávách překladače.

2.5 Příkaz function

Funkce obsahuje pouze vstupní parametry a vrací jedinou hodnotu. Má následující syntaxi, kde [] značí, že lze element vynechat a znak | možnost volby shodnou s regulárními výrazy buď, nebo:

```
[pure | impure ]   function function_name (parameter_list) return type_name is
                     [sequential_declarations]
                     begin
                       sequential statements
                     end [function] [function_name];
```

Nepovinné označení pure je výchozí volbou a specifikuje, že funkce vrací stejný výsledek pro totožné vstupní hodnoty. Klíčové slovo impure se použije, pokud funkce má vedlejší efekt a ke stejným vstupům může vrátit i odlišné hodnoty, například popisuje-li náhodný generátor. Uvedení klíčového slova function za koncovým end není sice vyžadováno všemi překladači, ale doporučuje se.

Deklarace ***sequential_declarations*** se shodují s deklaracemi uvnitř procesu, které jsem již rozebrali v Tabulka 2 na straně 21.

Příkazy sequential statements musí na všech svých možných cestách nezbytně zahrnout nejméně jeden příkaz return, jímž se určí návratová hodnota.

2.5.a @ Příklad II. - Unární xor jako funkce v knihovně

Na straně 22 jsme při objasňovaní tajů procesu použili příklad doplnění unárního xor, pokud používáme verzi VHDL, která ho ještě neobsahuje.

Můžeme si přidat i jinou architekturu, v níž si ho definujeme jako funkci:

```
architecture behfn of unary_xor is
   function uxor(X:std_logic_vector) return std_logic is
   variable result:std_logic:='0';  -- Contrary to a process, the initialization here is fully sufficient!
   begin
   gen: for i in X'RANGE loop  result:=result xor X(i);
        end loop;
        return result;
   end function;
begin
    Y<=uxor(X);
end architecture;
```

- Funkce, ani procedury, které uvedeme dále, se v syntéze **nikdy nevolají**, ale jejich kód se vloží do místa, v němž se použijí. **Inicializace přímo u proměnných** se v nich berou vždy v úvahu, jelikož překladač je automaticky konvertuje na inicializace přímo v kódu. Připomínáme, že **u procesu** se musí příkaz inicializace proměnné vložit do kódu, jak demonstroval Obrázek 6 na straně 22.
- Všimněte si, že jsme funkci záměrně použili v souběžné doméně zdrojového kódu. V sekvenční by se samozřejmě mohla také volat. Vždyť se pokaždé vkládá vložením části obvodu.
- V případě vektorů se neuvádějí rozsahy u vstupních parametrů a návratové hodnoty, pouze názvy typů. Vše se doplní až během vložení celé funkce při jejím použití.
- Funkce musí na všech svých možných cestách vracet nějakou hodnotu.
- Dokonce i konstantu lze inicializovat funkcí, pokud její výsledek lze jednoznačně stanovit v době překladu. Jazyk C podobné inicializace nepovolí, ale v něm funkce není obvodem.
- Funkce lze přetěžovat, a tak s výhodou využijeme možnosti definovat si vlastní balíček (package), jehož tvorba se vysvětlovala v závěru prvního dílu.

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

package unary_logic is  -- Library header
   function uxor(X:std_logic_vector) return std_logic;
   function uxor(X:unsigned) return std_logic;
   function uxor(X:signed) return std_logic;
end package;

package body unary_logic is

   function uxor(X:std_logic_vector) return std_logic is
   variable result:std_logic:='0';  -- Contrary to a process, the initialization is here fully sufficient!!
   begin
      gen: for i in X'RANGE loop  result:=result xor X(i); end loop;
           return result;
   end function;

   function uxor(X:unsigned) return std_logic is
   begin
           return uxor(std_logic_vector(X));
   end function;

   function uxor(X:signed) return std_logic is
   begin
           return uxor(std_logic_vector(X));
   end function;

end unary_logic;
```

Vytvořený balíček umožní použití unárního xor kdekoliv, stačí, když vložíme use odkaz na něj, buď globální před entitou či jen lokální v architektuře, procesu, funkci nebo proceduře. Samozřejmě časem ho můžeme vylepšit i o další logické funkce. Použití baličku si ukážeme v nové architektuře a tentokrát spolu s ní uvedeme i kopii entity ze strany 22.

```
library ieee; use ieee.std_logic_1164.all;
entity unary_xor is
    generic(HBIT:integer:=3);
    port (X: in std_logic_vector(HBIT downto 0):=(others=>'0');
          Y: out std_logic:='0');
end entity;

architecture behl of unary_xor is
use work.unary_logic.all;   -- local usage of our package inside architecture
begin
    Y<=uxor(X);
end architecture;
```

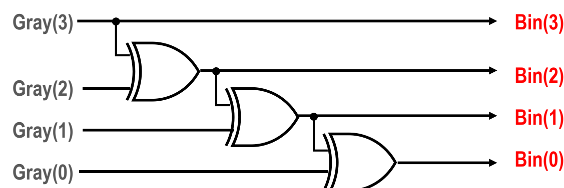
2.6 *** Cvičná uloha 2: Porovnání čísel v Grayově kódu

Mám dva výstupy X a Y, které nemusí mít totožnou délku, a udávají hodnoty v zrcadlově binárním Grayově (RCB - reflected binary code)[7]. Může se třeba jednat o výstupy ze snímačů polohy.

Porovnejte je po převodu na odpovídající čísla integer a poté vraťte GreaterThan rovné '1', je-li první vstup větší než druhý. Obvod bude určený entity ve tvaru:

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

entity GrayCodeGreaterThan is
   generic(HBITX:integer :=3; HBITY:integer:=4);
  port (X: in std_logic_vector(HBITX downto 0);
        Y: in std_logic_vector(HBITY downto 0);
     GreaterThan: out std_logic -- X greater then Y
      );
end entity;
```

Doplňte k ní pasivní proces testující vstupní parametry a taky architekturu s využitím funkce převodu Grayova kódu na integer, což lze provést snadno, stačí popsat obvod, který u vstupu se čtyřmi prvky může vypadat třeba takto:



Obrázek 8 - Převod z Grayova RBC na binární číslo

Více prvků znamená jenom prodloužení kaskády o další hradla xor.

Návod:

Vytvořte obvod provádějící operaci naznačenou na Obrázek 8 jako funkci a využijte ji k porovnání čísel.

[7] Popis najdete třeba na https://en.wikipedia.org/wiki/Gray_code

2.7 Příkaz procedure

VHDL procedure reprezentuje část obvodu, která ovlivňuje více hodnot. Její syntaxe:

```
procedure identifier [ (formal_parameter_list) ] is
   [sequential_declarations]
  begin
     sequential statement(s)
  end [procedure] [identifier];
```

- *sequential_declarations* se tady stejně jako u funkce shodují s deklaracemi uvnitř procesu, které jsme již uvedli v Tabulka 2 na straně 21.
- sequential statement(s) část může sice obsahovat příkaz return; ale bez návratové hodnoty.
- `formal_parameter_list` udává výčet formálních parametrů ve tvaru podobném jako u entity. Identifikátory stejného typu lze (stejně jako u entity) uvést oddělené čárkami a za jejich typem je středník s výjimkou posledního před konvovou ")" závorkou.
  - V seznamu lze zadat proměnné a signály, jimž se navíc přidávají vstupní a výstupní módy, ale jen in, out a inout. (Čtvrtý mód buffer se zde nesmí použít). Stejně jako u entity je **výchozím módem in**, pokud není nic specifikováno.
  - U vektorových typů se podobně jako u funkcí nezadávají rozsahy, protože popisujeme pouze vkládanou část obvodu. Jde o rozdíl oproti entity, v níž se naopak musí vždy rozsahy uvádět, neboť ta přímo definuje vstupy a výstupy celého obvodu, u nichž je nutné znát jejich velikost.
  - V simulační části lze dále použít i typ FILE, jemuž se však nesmí přiřadit mód.

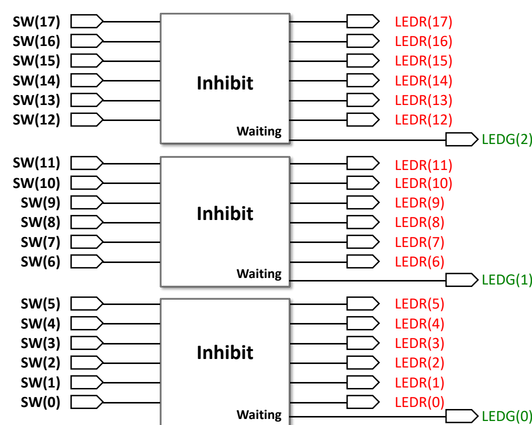
Užití procesu si vysvětlíme tradičně na příkladu vytvořeném tak, aby se ukázaly jeho zvláštnosti.

2.8 @Příklad III.: prioritní inhibitor rozlišující tři trojice

V prvním dílu učebnice jste v 9. cvičné úloze řešili prioritní inhibitor rozlišující tři trojice.

Přepínače SW(5 downto 0) tvořily 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).
Výstup pošleme na červené led diody LEDR(17 downto 0) vývojové desky DE2, respektive DE2-115.

Navíc si přidáme i LEDG(2 downto 0), které budou udávat, že se blokuje nějaký méně prioritní vstup.



Kód napíšeme, podobně jako v prvním dílu, ve formátu, v němž půjde později snadno upravit i na 6 trojic, kdy z každé se propustí nejvýš jediný signál v '1'. SW(2 downto 0) by tvořily první skupinu, SW(5 downto 3) druhou, atd. Výstup na LEDG by se pak samozřejmě rozšířil na 6 elementů.

Mohli bychom tady postupovat analogicky k prvnímu dílu, v němž se k řešení 9. cvičného příkladu používá souběžný for generate, pomocí kterého vkládá opakovaně příkaz port map. Zde bychom jenom využili již vytvořený InhibitG (viz podkapitola 2.1.d na straně 14).

Chceme si však procvičit proces, a tak předpokládejme, že jsme InhibitG dosud nevytvořili, a tak si neuděláme jako obvod, ale ve formě procesu, který vzápětí použijeme.

```vhdl
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);
           LEDG : out std_logic_vector(2 downto 0));
end;

architecture dataflow of demo_priority_inhibitor3x6 is
begin --architecture
   process(SW)
      procedure Inhibit(X: in std_logic_vector;
                        signal Y: out std_logic_vector;   -- output is a signal
                        W: out std_logic        -- output is a variable (default)
                        ) is
        variable yx : std_logic_vector(Y'RANGE); -- local definitions of process
        variable en : std_logic:='1';
        variable wx : std_logic:='0';
        begin --procedure body
   igen :  for i in X'RANGE loop
             yx(i) := en and X(i);
             wx := wx or (not en and X(i)); en:= en and not X(i);
          end loop;
         Y<=yx;
         W:=wx;
      end procedure;
      variable LEDGvar :std_logic_vector(LEDG'RANGE);-- we must keep W process output as a variable!
   begin -- process
       iloop: for i in 0 to 2 loop
      --  the usage of keyword notation (cz:jmenne asociace)
         Inhibit(X=>SW(6*i+5 downto 6*i),Y=>LEDR(6*i+5 downto 6*i),W=>LEDGvar(i));
      -- the alternative usage of positional notation (cz:pozicni asociace)
       --   Inhibit(SW(6*i+5 downto 6*i),LEDR(6*i+5 downto 6*i),LEDGvar(i));
      end loop;
      LEDG<=LEDGvar;
   end process;
end architecture;
```

- Před rozborem kódu si připomeneme, že VHDL procedure, podobně jako function, se nikdy nevolá ve stylu klasických programů, ale celý její kód se pokaždé vkládá do místa, v němž se na ni odkázalo, akorát se formální parametry zadané v hlavičce procedure nahradí skutečně použitými identifikátory.
- Mohli jsme procedure definovat v deklarací architecture (před jejím begin), poté by zůstal dostupný i dalším process blokům uvnitř architecture, ale zvolili jsme lokální definici uvnitř process.
- Kvůli demonstraci všech vlastností jsme výstup Y procedure Inhibit záměrně předepsali jako signal, zatímco druhý W jsme nechali na variable (výchozí stav, není-li nic zadané). Všimněte si, že do Y ukládáme hodnotu pomocí souběžného přiřazení, tedy jako do signálu, zatímco do W zapisujeme **:=** příkazem okamžitého přiřazení hodnoty.
- Samotný odkaz na procedure se podobá příkazu port map, podobně jako v něm můžeme zvolit jmenné nebo poziční asociace. Jelikož W je variable, smíme naši procedure použít jedině v sekvenční doméně kódu, neboť jinde nelze definovat variable. Nemůžeme dát přímo LEDG, ale vytvořili jsme si pomocný signál LEDGvar, jehož hodnotu posíláme souběžným přiřazením do LEDG.
- Vstupní parametry s módem in lze nahrazovat jak signály tak proměnnými či konstantami, neboť zmíněné typy se chovají během čtení svých hodnot stejně.

Pokud bychom chtěli procedure Inhibit použít přímo v souběžné části taky můžeme, ale pak oba výstupní formální parametry zadáme jako signal. Ukažme si použití jako novou architekturu:

```
architecture dataflow2 of demo_priority_inhibitor3x6 is
  procedure Inhibit(X: in std_logic_vector;
                    signal Y: out std_logic_vector;   -- output is a signal
                    signal W: out std_logic          -- output is now a signal
                    ) is
     variable yx : std_logic_vector(Y'RANGE); -- local definitions of process
     variable en : std_logic:='1';
     variable wx : std_logic:='0';
  begin --procedure body
 igen:  for i in X'RANGE loop
          yx(i) := en and X(i); wx := wx or (not en and X(i)); en:= en and not X(i);
        end loop;
        Y<=yx;  W<=wx;
  end procedure;
begin --architecture
   iloop:  for i in 0 to 2 generate
              Inhibit(SW(6*i+5 downto 6*i),LEDR(6*i+5 downto 6*i),LEDG(i));
           end generate;
end architecture;
```

Naše procedure se také může přidat do balíčku (package), takže ji půjde aplikovat kdekoli.

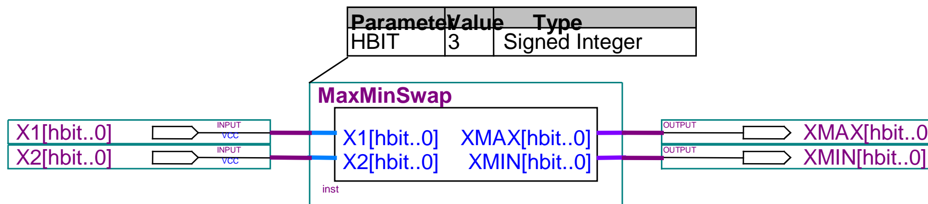
Lze se ptát, zda máme raději upřednostnit entity, či obvody tvořit pomocí procedur či funkcí. Složitější se lépe vytvoří jako entity, ve které může kooperovat i více VHDL procesů. U menších si jako kriterium zvolíme především přehlednost kódu. Popíšeme obvod prostředky, které se nám momentálně hodí, ale srozumitelně nejen i ostatním, ale i nám, až se ke kódu vrátíme za delší dobu:-)

2.9 *** Cvičná úloha 3: MaxMinSwap pomocí procedure

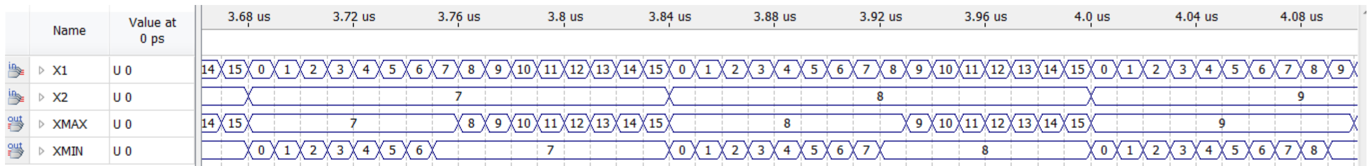
Zkuste si vytvořit obvod, který prohodí vstupy X1 a X2 tak, aby ten s vyšší hodnotou, branou jako číslo unsigned, se poslal na výstup XMAX, zatímco druhý na výstup XMIN.

Obvod by mohl mít následující schematickou značku:

-



Část simulace vypadá takto:

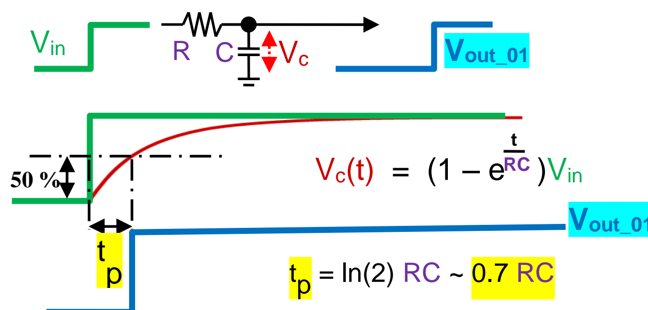


Úkoly:

1. Ač by příklad šel řešit přímo souběžnými příkazy, ale vytvořte ho pomocí definice procedury a vyzkoušejte ji simulací.
2. Vytvořte si vlastní balíček, třeba v souboru my_logic.vhd a dejte proceduru do něj. Přidejte si další architekturu, tentokrát s použitím balíčku.

2.10 Příkaz after u <= přiřazení do signálu

V předchozích kapitolách jsme několikrát připomněli rozdíly mezi signal a variable, ale zatím jsme neuvedli důvody jejich odlišného chování, které paradoxně plynou z RC článku.



Obrázek 9 - Zpoždění na RC článku

Nechť jsme v pozitivní logice, v níž logická '0' odpovídá napětí 0 V, zatímco logická '1' se téměř rovná napájecímu napětí. Na V_{in} vstup RC článku přišel signál z logického členu, který se změnil z '0' na '1'. Kondenzátor C se začne nabíjet na napětí V_c (t) popsané diferenciální rovnicí, v níž t označuje čas.

Z hlediska logiky nás zajímá okamžik, kdy napětí V_c dosáhne rozhodovací úrovně kolem 50 % a vyhodnotí se už jako logická '1'. Řešením rovnice lze stanovit, že se tak stane za čas t_p rovný 0.7 RC [8].

RC článek nám tak zpozdil signál a stejně tak ho pozdrží i při přechodu vstupu z '1' do '0', jen se kondenzátor bude místo nabíjení vybíjet.

Každý vodič se chová jako vedení s rozprostřenými parametry a signály se po něm šíří pomaleji než rychlost světla kvůli nabíjení jeho kapacit. Na plošných spojích se přenos zpomaluje zhruba na 0.7 až 0.85 c, uvnitř obvodů může dojít i k línějšímu šíření signálů. Vysvětlení problematiky přesahuje rámec této publikace a zájemci mohou najít víc vyhledáním „transmission line“, přičemž z hlediska plošných spojů je nejdůležitější „lossless transmission line“ [9].

V simulaci, a výhradně v ní, lze definovat zpoždění na vodiči příkazem:

```
y_signal <= transport x_expression after time_value;
```

kde *time_value* se skládá z číselné hodnoty, za níž se uvádí **mezerou oddělená** některá jednotka času:

Jednotka VHDL	Název	Velikost
fs	femtoseconds	10^{-15} seconds
ps	picoseconds	10^{-12} seconds
ns	nanoseconds	10^{-9} seconds
us	microseconds	10^{-6} seconds
ms	miliseconds	10^{-3} seconds
sec	seconds	
min	minutes	60 seconds
hr	hours	3600 seconds

Tabulka 3 - Jednotky času ve VHDL

Jako *time_value* lze užít i výraz, který dává datový typ time. S časovými hodnotami lze v kódech určených k simulaci zacházet jako s reálnými čísly, viz třeba ukázka jejich použití:

---

[8] Poznámka: Hodnota násobitele je sice ln(2)=0.6931471805599..., ale většinou známe hodnoty odporů R a kapacit C leda tak s přesností 10 %, zpravidla spíš s ještě horší, a tak zaokrouhlením na 7/10 nic nezkazíme.

[9] V době psaní této publikace existovalo například na https://practicalee.com/transmission-lines/ velmi krásné vysvětlení chování logických signálů na vedení, a to včetně názorné YouTube animace, kterou vřele doporučujeme shlédnout.

```
architecture hokus_pokus of some_testbench is
constant PERIODS : time := 400 ns;
constant TWAIT : time := 20.5 ms;
begin
   process
     variable TD1, TD2, TD3 : time;
   begin
     TD1 := 2*TWAIT-PERIODS; -- TD1 = 40999.6 us
     TD2 := 4*TWAIT+5*PERIODS-10.75 us;  -- TD2 = 81991.250 us
     TD3 := TD2/10 - 99125 ns; -- 8.1 ms
     wait; -- infinit wait -> end simulation of this process
   end process;
end architecture;
```

Uvnitř obvodů se spojů měří v mikrometrech a neprojeví se významněji. Hlavním faktorem bude průchod přes hradla, která v roli zdrojů signálů vykazují vnitřní odpor. Jejich vstupy mají parazitní kapacity, které se sčítají, budí-li zdroj současně více vstupů hradel najednou, a tak zpoždění není malé.

Hradlo získalo díky své setrvačnosti (vyvolané čekáním na nabití kapacity, viz učebnice Logické obvody na FPGA) významnou vlastnost setrvačného (inerciálního) zpoždění (inertial). Odmítne (reject) propustit kratší pulzy než doba jeho zpoždění.

Zpoždění na hradle lze popsat příkazem

```
y_signal <= x_expression after time_delay;
```

který je zcela ekvivalentní se zápisem:

```
y_signal <= reject time_delay inertial x_expression after time_delay;
```

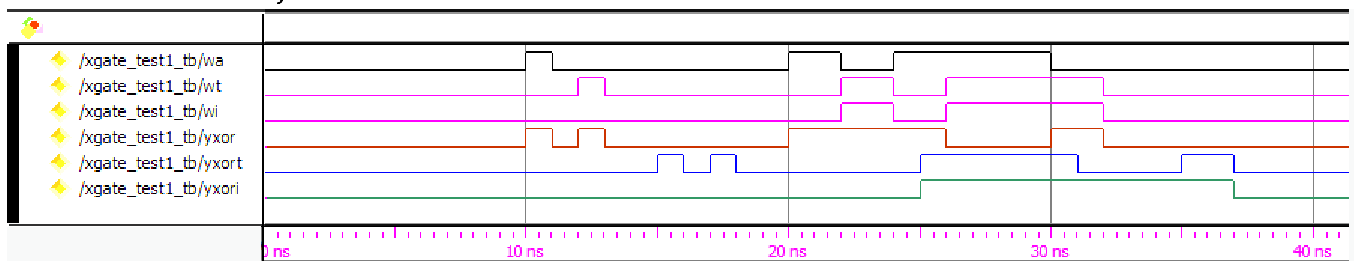
Části reject a inertial se doplnily automaticky (výchozí stav), ale můžeme je přidat i sami, potřebujeme-li použít odlišný reject time_delay.

Srovnání s chováním předchozího příkazu

```
y_signal <= transport x_expression after time_delay;
```

si demonstrujeme na výstupu simulačního programu:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all; library work;
entity some_testbench is
end entity;
architecture demo_after of some_testbench is
signal wa, wt, wi, yxor, yxort, yxori : std_logic:='0';
begin
   wa <=   '0', '1' after 10 ns, '0' after 11 ns,
           '1' after 20 ns, '0' after 22 ns, '1' after 24 ns, '0' after 30 ns;
  wt <= transport wa after 2 ns;
  wi <= wa after 2 ns;
  yxor <= wa xor wt;
   yxort <=  transport wa xor wt  after 5 ns;
   yxori <= wa xor wt after 5 ns;
end architecture;
```



- **wa** signál obsahuje několik zpoždění od sebe oddělených čárkami a s časovými hodnotami, které tvoří **vzestupnou** řadu (nutná podmínka). Signálu se přiřazují konstanty, tedy hodnoty stabilní od počátku času, takže se chovají stejně v obou typech zpoždění.

  Poznámka: Doplnění případného klíčového slova transport se povoluje i v řadě, avšak jen před jejím prvním členem a ovlivnilo by výhradně ten, všechny další za oddělovacími čárkami budou už inertial.

- **wt** se transportně zpozdí na vodiči, takže projdou pulzy jakékoli délky.
- **wi** má výchozí inerciálním chování se zadaným zpožděním 2 ns, doplní se rovněž i výchozí reject 2 ns, takže se zadrží dva pulzy o délce 1 ns v simulačních časech 10 ns a 12 ns.
- **yxor** se vytváří bez udání zpoždění z předchozích signálů operací wa **xor** wt.
- **yxort** se stejnou operací má transportní zpoždění a dává výstup stejný jako yxor, jen o 5 ns zpožděný.
- **yxori** s totožnou operaci jako yxor, ale tentokrát s inerciálním zpožděním o 5 ns, svůj výstup nejen pozdrží, ale současně se v něm potlačí vše kratší než udaná doba zpoždění.

Příkaz yxor <= wa xor wt; bez uvedeného zpoždění, bude překladačem chápaný jako přiřazení: yxor <= wa xor wt after δ; kde δ je jakési interní nekonečně malé zpoždění (δ nelze ve VHDL napsat). Každé přiřazení <= bez after se bere jako s after δ zpožděním, což souhlasí i s realitou. Pošleme-li něco na vstup vodiče (hradla), nová hodnota ve stejném okamžiku není na jeho výstupu, ale teprve k němu putuje.

Na závěr můžeme ještě vznést otázku, jak bude dlouhé zpoždění na hradle? Odpověď není jednoznačná. Hodnota závisí na typu technologie, jejím napájecím napětí a teplotě. Literatura uvádí různé hodnoty.

Přibližně lze říci, že jeden invertor v dnešní 32 nm CMOS technologii zpozdí zhruba mezi 150 až 200 ps. V jemnější technologii 16 nm se nejčastěji uvádějí hodnoty kolem 100 až 130 ps. Ve vědeckých článcích věnovaných 7 nm nejrychlejší technologii FinFET, 3D obdoba CMOS technologie, lze nalézt i 50 ps.

Zpoždění na hradlech má relativně nízké hodnoty a můžeme ho mnohdy zanedbat, ovšem s výjimkou několika případů, z nichž dva hlavní jsou:

- Zpoždění vytváří hazardy v kombinačních obvodech, což přiblížíme hned v kapitole **Chyba! Nenalezen zdroj odkazů.**.
- Musíme dbát na správné časování synchronní obvodů, jimž věnujeme celou kapitolu 3 začínající na stránce 33.

2.11 Blocking versus non-blocking příkazy

V předešlém textu jsme naznačili, že sekvenční přiřazení := má blocking charakter (cz:blokující, sekvenční???), zatímco souběžné přiřazení <= je non-blocking. Oba termíny by čtenáři měli znát z imperativních programovacích jazyků, ale asi nepochybíme, pokud je přiblížíme.

Klasické programovací jazyky (typu C nebo Java) se ve zdrojovém kódu zapisují příkazy typu blocking, což znamená, že se čeká na dokončení celého příkazu, a teprve po něm se provede další. Podobně se volají i funkce.

Operační systémy dovolují i tak zvané non-blocking (cz:asynchronní, neblokující?) volání, kdy se pouze inicializuje operace, ale nečeká se na její dokončení. Zde můžeme uvést třeba přijímání síťových socketů, které obvykle provádí v non-blocking módu. Program jen zahájí čtení socketů, ale nečeká, až nějaký dorazí, pokračuje jinými úkony. Operační systém pak ohlásí nějakým příznakem, že se již něco přijalo, případně v přerušení aktivuje čtecí operaci. Stejným způsobem lze třeba i otevřít soubor. Dáme jenom pokyn, aby se operace inicializovala, ale nečekáme na její dokončení.

V programech se non-blocking operace realizují vlákny, která mohou běžet na různých jádrech procesoru, takže se i fyzicky vykonávají souběžně. I samotné pipeline (cz:zřetězené zpracování) instrukcí strojového kódu v procesoru vykazuje řadu charakteristik souběžného zpracování společně s technologií hyper-threading sdílející prostředky jednoho jádra dvojicí proudů instrukcí.

Dovolíme si tvrdit, že klasickým počítačům je vlastní blocking chování, avšak kvůli zrychlení se v nich různými triky napodobuje souběžné non-blocking zpracování.

U obvodů narazíme na přesně opačnou situaci. Každý jejich prvek běží souběžně, a tak celé zapojení má charakter non-blocking, avšak různými fígly se v něm často vytváří blocking chování, a to především kvůli zjednodušení návrhu či eliminaci rušení. Jedním z řady triků budou i synchronní obvody, jimž věnujeme další kapitolu, a také konečné automaty, které podrobně rozebereme v druhé polovině.

Sekvenční zdrojová doména používá příkazy, které emulují blocking charakter, a to včetně := přiřazení. V syntéze se však neprovádějí, všechny je nahrazuje obvod, na který se převedly. Konverze na něj se však realizovala, jako kdyby se příslušná část sekvenčního kódu, tedy jeden process, function nebo procedure, vykonala ve stojícím čase. Vždyť je popisem struktury obvodu, nikoli programem! Pozor, tady závisí na jejich pořadí.

Všechny příkazy souběžné (concurrent) domény mají zase non-blocking charakter. Provádí se všechny najednou a už v prvním díle jsme si několikrát ukázali, že prakticky nezáleží na pořadí, v jakém se uvedou ve zdrojovém kódu. Všechny běží současně.

Přiřazení <= do signálu si v sekvenční doméně zdrojového kódu pořád zachovává svůj souběžný charakter. Představte si <= jako tobogán. Na jeho začátek strkáme nová '0'/'1' data, která kloužou rourou ke konci. Dorazí k němu za nekonečně krátkou dobu, avšak při převodu jednoho bloku sekvenční části se vždy předpokládá stojící čas. Kdykoli se během její konverze na obvod podíváme na konec tobogánu, vidíme tam pořád původní '0'/'1', tedy to, co tam bylo na začátku, což přesně odpovídá obvodové realitě.

Věnovali jsme blocking a non-blocking více pozornosti, aby posluchači snáze přijali fakt souběžného chování všech prvků obvodu. Sekvenční doména zdrojového kódu jim hodně pomáhá, ale pořád se hodí za ní vidět obvod, který po konverzi bude souběžným prvkem.

3 Obvody řízené hodinovým signálem

Hodí se, aby návrháři znali principy obvodů řízených hodinami, neboť pak nemusí v hlavě nosit encyklopedii zásad jejich použití, ale potřebné pravidla si odvodí sami. Jejich výklad se provedl v učebnici Logické obvody na FPGA v kapitole 7. Octujeme z ní část jazykové poznámky podrobně vysvětlené v její podkapitole 7.1, a to anglickou terminologii podle jejího užití ve vývojových nástrojích obvodů:

- latch v nich vymezuje jen asynchronní klopné obvody, jako RS latch a D-latch. Jejich přesné české ekvivalenty jsou „RS a D úrovňové klopné obvody".
- *edge-triggered* latch - čili hranou řízený latch zahrnuje celou kategorii klopných obvodů, které změní svůj výstup jen při příchodu nějaké hrany hodinového signálu.
- *flip-flop* - určuje v návrhových prostředích nejčastější *edge-triggered* latch zapojení. To se označuje zavedenou zkratkou DFF, data flip-flop. Podrobně se v Logických obvodech vysvětlil v kapitole 7.4.

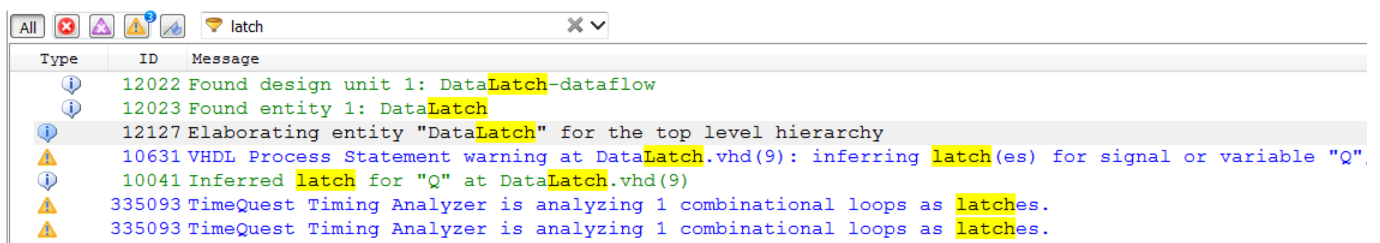
Další text obohatíme o nová „ryze" česká slova latch a flip-flop, s nimiž lze elegantněji psát „flip-flop se překlopil" místo „klopný obvod se překlopil". Pojem „klopný obvod" degradujeme na jejich synonymum.

3.1 Obvod typu D-Latch

V FPGA se samotný D-Latch považuje za parazitní nechtěný prvek. Poměrně snadno se vytvoří neúplným přiřazením, o němž jsme se zmínili na straně 9, kde jsme ho nazvali nevhodným návrhem pro FPGA.

```
library ieee; use ieee.std_logic_1164.all;
entity DataLatch is
   port ( D, ENA : in std_logic; Q : out std_logic);
end entity;

architecture dataflow of DataLatch is
begin
   process (D, ENA)
   begin
      if ENA='1' then Q<=D; end if;
   end process;
end dataflow;
```

V kódu přiřazujeme výstupu Q hodnotu jen při ENA v logické '1', což znamená, že Q si musí při ENA='0' držet svou poslední hodnotu,[10] což lze jen v D-Latch vytvořeným z NAND hradel. Vypíší se varování:



Obrázek 10 - Zprávy o vytvoření nežádoucího Latch

3.2 Klopný obvod DFF - Data Flip-Flop

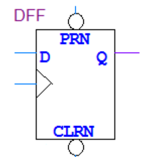
DFF obvod vzorkuje hodnoty na vstupu D při na hraně hodinového signálu a podrží si poslední vzorek na svém výstupu Q. Eliminuje tím hlavní nevýhodu D-latch, který po celou dobu ENA v logické '1' propouštěl vše ze vstupu D na výstup Q. Navrhly se již desítky různých zapojení DFF s odlišnými vlastnostmi, ale drtivá většina používaných DFF bývá citlivá výhradně jenom na jeden typ hrany.

[10] Požadavek na udržení poslední hodnoty vyplývá z deterministického chování VHDL specifikace. Veličiny si musí zachovat hodnoty, které jsme jim přiřadili. Nestojíme o stochastické změny chování obvodu.☺

V FPGA se asynchronní vstupy DFF používají výhradně k inicializaci po zapnutí napájení.

Pokud inicializace není potřeba, nepotřebný asynchronní vstup se připojí na '1', zatímco ostatní nulovací ACLRN vstupy se napojí na signál, často odvozený od výstupu RC článku, neboť kondenzátor má nulové napětí, bylo-li napájení vypnuté na delší dobu[11]. Viz „Logické obvody na FPGA“ kapitoly 7.4.1 a 7.4.2.

Obecná značka DFF

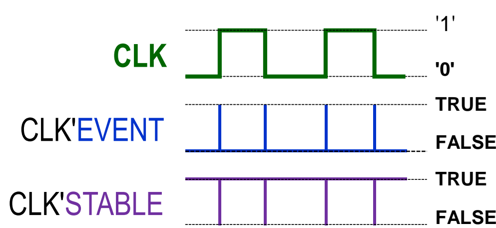


Pozor! Obecné symbolické značky DFF sice zahrnují jak asynchronní nulování, tak přednastavení, ale o použití rozhodují typy DFF v FPGA, v němž vytváříme obvod. Mají-li jen vstup CLRN, pak se každý **DFF musí inicializovat jen na hodnotu konstanty** známé v době překladu. Kdyby se inicializoval na různé hodnoty, třeba podle stavu vnějšího signálu (například od přepínače na desce), překladač by proměnnou inicializaci nutně řešil přidáním nedovoleného Latch, který se vysvětloval v učebnici Logické obvody.
A rozhodně nechceme, aby se kazi-obvodový latch objevil v FPGA!

3.3 Synchronní klopný obvod ve VHDL

Vložení klopného obvodu specifikujeme příkazem, že si přejeme změnu signálu až po náběžné/sestupné hraně, k čemuž využijeme atributy, které známe již z prvního dílu.

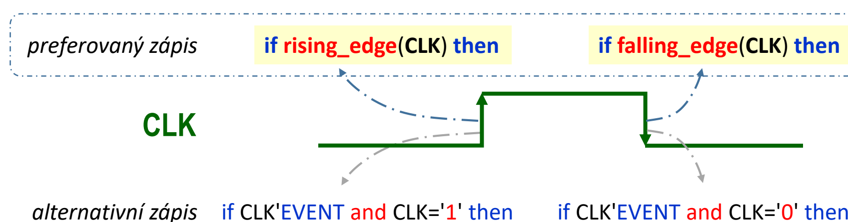
Všechny signály mají předdefinovaný atribut EVENT, coby příznak jejich změny. Jeho negací je atribut STABLE. Oba atributy ale chybí u proměnných (variable).



Nechť máme jakýkoli signál, třeba nazvaný CLK.

- Atribut CLK'EVENT bude rovněž signálem, ale typu boolean, a nabývá TRUE pouze na náběžné hraně signálu CLK.
- Atribut CLK'STABLE bude jeho negací.

Atributy sice existují ke všem signálů, ale můžeme jako hodiny použít pouze signály std_logic. Chceme-li předepsat, že veličina se v obvodu změní až po hraně zvoleného hodinového signálu, využijeme k tomu funkce definované v základní knihovně ieee.std_logic_1164 (na níž odkazuje každý VHDL kód).



Obrázek 11 - Funkce rising_edge() a falling_edge()

Smysl alternativního zápisu dole lze vyjádřit slovy, že právě došlo ke změně signály (CLK'EVENT) a současný stav CLK se rovná '1', tedy nastala náběžná hrana, nebo nyní je '0', tudíž přišla závěrná hrana CLK.

Preferované funkce jsou výhodnější i v simulaci. Obsahují kromě podmínky na EVENT (podobné alternativním zápisů v obrázku) i další přídavné testy, jimiž se zajistí, že náběžné hrana se detekuje jen při přechodu '0'->'1', a sestupná pouze při '1'->'0'´, jelikož v 9-hodnotové std_logic mohou během simulace nastat i jiné hrany signálu jako třeba 'U'->'1' a 'U'->'0', apod., které pak vnášejí další reakce zamlžující vyhodnocení výsledků. Navrhujeme-li obvod určený pouze k syntéze, bude úplně jedno, zda se použije preferované funkce rising_edge a falling_edge, či alternativní konstrukce s EVENT. Dostaneme totožný obvod.

[11] Právě kvůli inicializacím běžně založeným na RC-článku se zařízení, u něhož se potřebuje inicializace, musí nechat vypnuté delší dobu před novým zapnutím. Zde možno zmínit návody k modemům, či potřebu na chvíli vyjmout baterku z mobilu.

3.4 @ Příklad IV. - Synchronní klopný obvod s dvojím nulováním

Použití detekce hrany ve VHDL si ukážeme na různých architekturách vytvořených k entitě:

```
library ieee; use ieee.std_logic_1164.all;
entity DataFlipFlop is
  port ( Data, CLOCK, ENABLE, ACLRN : in std_logic;
         SCLEAR : in std_logic; -- clear on CLK rising edge
         Q : out std_logic);
end entity;
```

kde

- Data, CLOCK, ENABLE a ACLRN - vstupy korespondují s D, CLK, ENA a ACLRN popsaného DFF;
- SCLEAR vstup nuluje výstup Q až po náběžné hraně signálu CLOCK, tzv. synchronní nulování;
- Q - označuje výstup DFF.

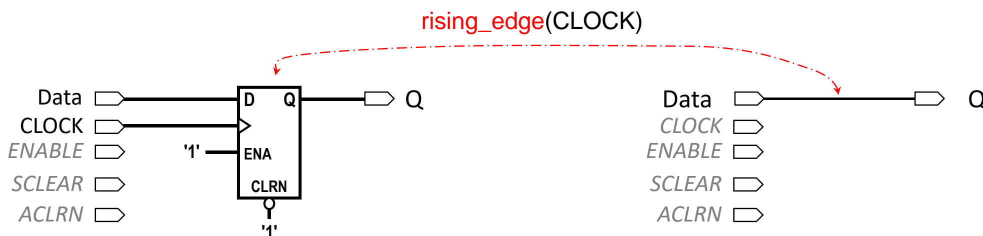
Obvod budeme postupně budovat, a tak zpočátku nevyužijeme všechny vstupy.

Architektura rtl_dq uvedená vpravo vytvoří pouze přiřazení Data vstupu výstupu Q, ostatní vstupy zůstanou nezapojené. V sensitivity listu procesu odkážeme jenom na vstup Data, neboť na něm závisí Q.

Architektura rtl_re vlevo obklopila příkaz Q<=Data; podmínkou if rising_edge(CLOCK) then, která specifikuje, že aktualizace Q nastane až po náběžné hraně. Do propojení se tady vložilo DFF. V sensitivity listu procesu máme výhradně CLOCK, protože jedině po jeho změně nastane aktualizace hodnoty Q.

```
architecture rtl_re of DataFlipFlop is
begin
  process(CLOCK)
  begin
    if rising_edge(CLOCK) then
      Q<=Data;
    end if;
  end process;
end architecture;
```

```
architecture rtl_dq of DataFlipFlop is
begin
  process(Data)
  begin

    Q<=Data;

  end process;
end architecture;
```

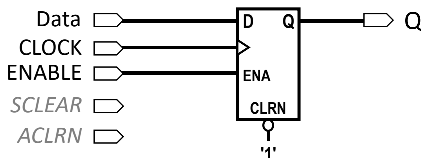


V logickém elementu použitého FPGA předpokládáme DFFE pouze s asynchronním nulováním ACLRN, takže překladač automaticky propojí na '1' (v symbolickém schématu značené Vcc) jeho nepoužité vstupy.

Připojení vstupu ENABLE lze udělat přidáním podmínky za rising_edge, nebo vložením příkazu if.

```
architecture rtl_ren of DataFlipFlop is
begin
  process(CLOCK)
  begin
    if rising_edge(CLOCK) and ENABLE='1' then

      Q<=Data;

    end if;
  end process;
end architecture;
```

```
architecture rtl_ren1 of DataFlipFlop is
begin
  process(CLOCK)
  begin
    if rising_edge(CLOCK) then
      if ENABLE='1' then
        Q<=Data;
      end if;
    end if;
  end process;
end architecture;
```



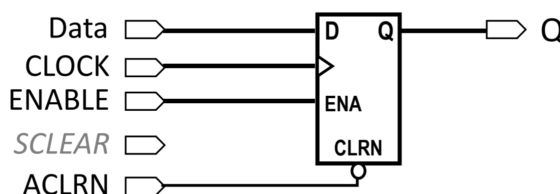
Oba způsoby dávají ekvivalentní výsledek. Závisí na nás, který z nich se nám zdá přehlednější.

V pravé architektuře máme **příkaz if s neúplným přiřazením**, který se tady (na rozdíl od kombinačních obvodů) nachází uvnitř podmínky detekce náběžné hrany (tedy specifikace vložení obvodu DFF), který si **přiřazené hodnoty zapamatuje automaticky**. Neúplná přiřazení se běžně vkládají do části s detekcí hrany hodin. Specifikujeme v nich jen nové hodnoty, staré zůstanou automaticky uchované v DFF [12].

Chceme-li přidat asynchronní nulovací vstup, pak třeba dodržet podmínku, že asynchronní chování má vyšší prioritu než CLOCK, protože při ACLRN='0' bude Q vždy v '0'. Test na ACLRN musí bezpodmínečně proběhnout před detekcí náběžné hrany. V opačném případě obvod nelze fyzicky realizovat.

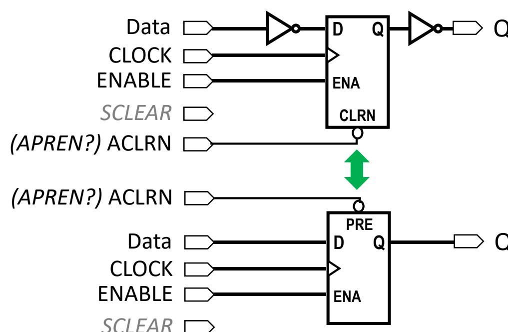
Sensitivity list procesu nyní obsahuje i ACLRN, které ovlivňuje nezávisle na hodinách.

```
architecture rtl_renec of DataFlipFlop is
begin
  process(CLOCK, ACLRN)
  begin
    if ACLRN='0' then Q<='0';
    elsif rising_edge(CLOCK) then
      if ENABLE='1' then Q<=Data;
      end if;
    end if;
  end process;
end architecture;
```



Můžeme zkusit i opačnou inicializaci, pak by se ovšem vstup ACLRN měl jmenovat spíš APREN. Máme-li v používaném FPGA pouze DFF jen s asynchronním nulováním, pak inicializace na '0' znamená efektivnější postup, ale na '1' se také vytvoří přidání invertorů, což lze považovat za pořád přijatelné řešení kvůli minimálnímu nárůstu složitosti.

```
architecture rtl_renep of DataFlipFlop is
begin
  process(CLOCK, ACLRN)
  begin
    if ACLRN='0' then Q<='1';
    elsif rising_edge(CLOCK) then
      if ENABLE='1' then Q<=Data;
      end if;
    end if;
  end process;
end architecture;
```



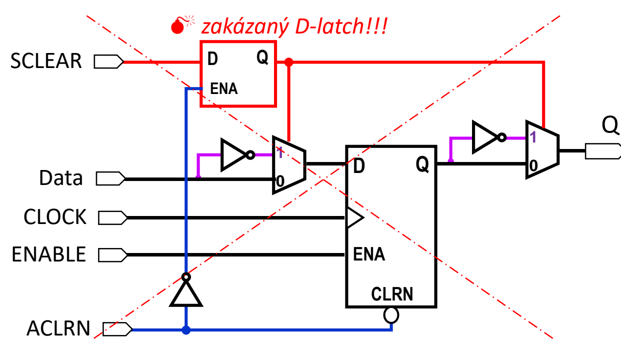
Všimněte si, že **Q se asynchronně inicializuje vždy výhradně na konstantu známou v době překladu**. Píšeme-li kód určený k syntéze a v FPGA nemáme DFF s dvojicí asynchronních vstupů dovolujících inicializaci jak na '0', tak na '1' (ty se v nich málokdy implementují), pak asynchronní inicializace Q na nekonstantní hodnotu, třeba na signál SCLEAR (Q<=SCLEAR;) zcela znehodnotí návrh!!!

[12] Kombinační obvod nemá paměťový prvek, kvůli tomu je kombinační (jeho hodnota dle definice závisí výhradně na vstupech). Užije-li se v nich neúplné přiřazení, které si žádá zapamatování hodnoty, pak se přidá latch (striktně zakázaný v FPGA), čímž se kombinační obvod vlastně zdeformuje na sekvenční.

Nahrazení řádku asynchronní inicializace na hodnotu signálu

```
if ACLRN='0' then Q<=SCLEAR;
```

vede na obvod, v němž se musí přidat D-latch, aby se v něm zapamatovala poslední hodnota SCLEAR na konci pulzu ACLRN='0'. Výstupem D-latch se pak přepíná mezi přidáním invertorů přímou cestou. A D-latch se nesmí použít v FPGA.

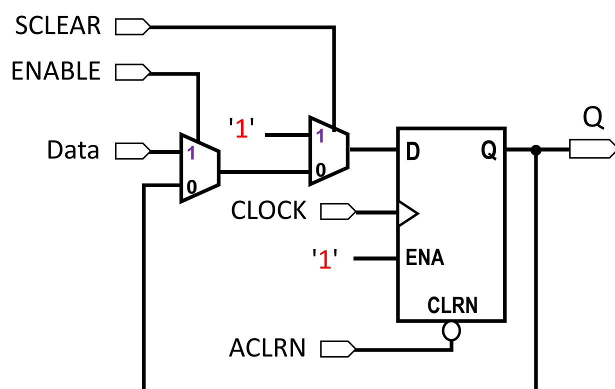


Obrázek 12 - Důsledek nevhodné asynchronní inicializace DFF

Příklad současně ukazuje, že VHDL kód se opravdu musí psát s rozmyslem, protože i taková zdánlivě drobná změna vede na nepoužitelnost výsledku určeného k syntéze obvodu.

Ve finální verzi VHDL popisu doplníme SCLEAR pracovní nulování obvodu při náběžné hraně bez ohledu na stav ENABLE a Data.

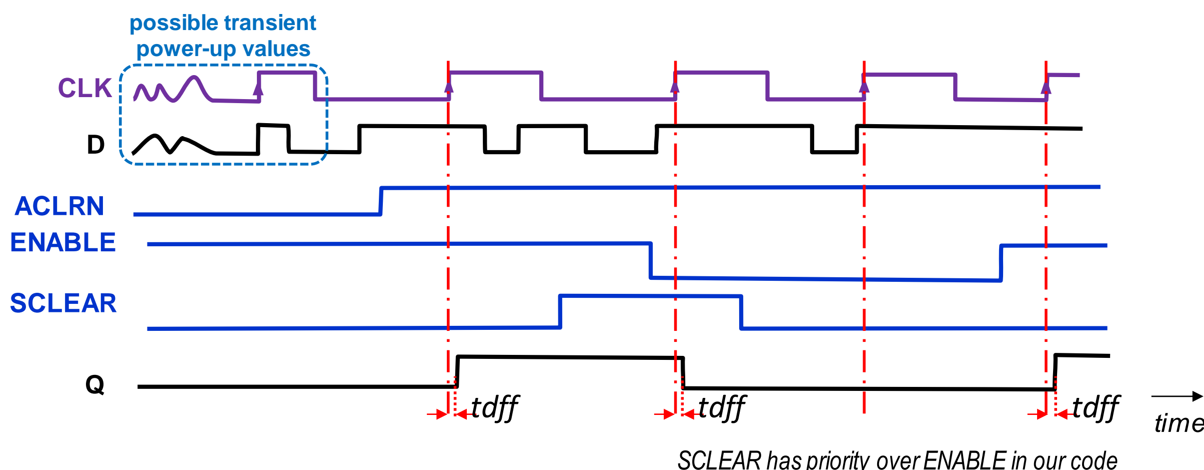
```
architecture rtl_final of DataFlipFlop is
begin
  process(CLOCK, ACLRN)
  begin
    if ACLRN='0' then Q<='0';
    elsif rising_edge(CLOCK) then
      if SCLEAR='1' then Q<='0';
      elsif ENABLE='1' then Q<=Data;
      end if;
    end if;
  end process;
end architecture;
```



Obrázek 13 - Finální synchronní klopný obvod s dvojím nulováním

Překladač nevyužije vstup ENA obvodu DFFE, jelikož jsme předepsali reakce na náběžnou hranu při vstupu SCLEAR='1' bez ohledu na Data a ENABLE. Vstupu ENABLE='0' přepne svůj multiplexor v kaskádě (překlad podmínky if then elsif), aby se nahrával výstup Q, což se probíralo v učebnici Logické obvody.

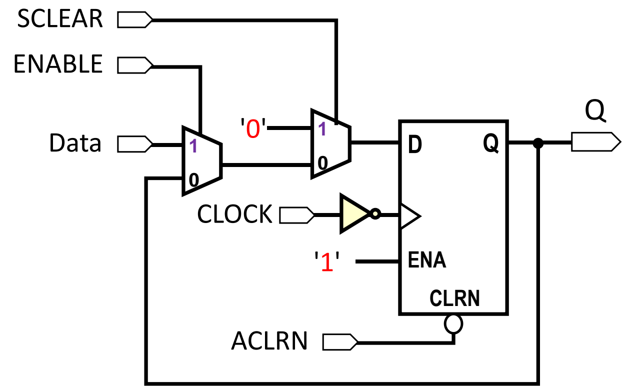
Odezva DFFE na nějaké vstupní signály může vypadat následovně:

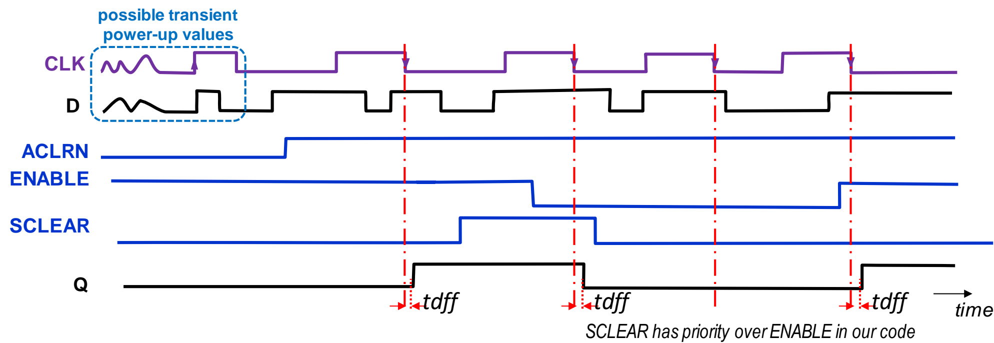


Obrázek 14 - Ukázka reakcí synchronního klopného obvodu s dvojím nulováním

Pokud bychom chtěli změnu na sestupnou hranu, stačí nám jen nahradit rising_edge voláním falling_edge. Ve schématu se pak přidá leda invertor před vstup hodin.

```
architecture rtl_finalf of DataFlipFlop is
begin
  process(CLOCK, ACLRN)
  begin
    if ACLRN='0' then Q<='0';
    elsif falling_edge(CLOCK) then
      if SCLEAR='1' then Q<='0';
      elsif ENABLE='1' then Q<=Data;
      end if;
    end if;
  end process;
end architecture;
```





Obrázek 15 - Synchronní klopný obvod s dvojím nulováním reagující na sestupnou hranu

3.5 @ Příklad V. - Registr s Q a QN

Vytvoříme si nyní 1-bitový registr, tedy synchronní klopný obvod, avšak s výstupem Q a jeho negací QN, kterou nemá námi předpokládaný DFF v FPGA.

Vstupy si tentokrát zredukujeme na Data a CLOCK, avšak přidáme si další výstup QN.

```
library ieee; use ieee.std_logic_1164.all;
entity DataFlipFlop is
  port ( Data, CLOCK : in std_logic; Q, QN : out std_logic);
end entity;
```

Zkusíme si napřed vytvořit negovaný výstup pouhým dalším přiřazením:

```
architecture rtl1 of DffQN is
begin
  process(CLOCK)
  begin
    if rising_edge(CLOCK) then Q<=Data; QN<=not Data; end if;
  end process;
end architecture;
```

Kód je zcela správný, ale dva <= souběžné (neblokující) příkazy znamenají vložení dvou DFF.

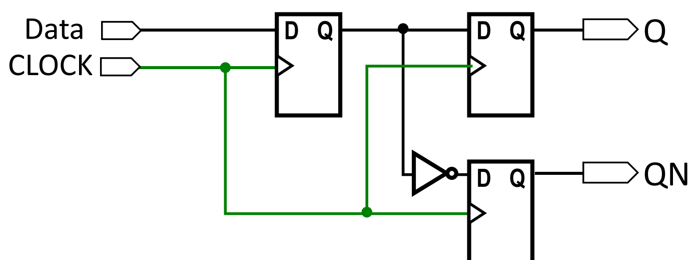
V podmínce detekce hrany se každé <= přiřazení převede vložením DFF.

Vstupy našeho DFFE, neuvedené kvůli zjednodušení, budeme předpokládat připojené na '1'.

Obrázek 16 - Dva DFF s Q a QN

Řešení není špatné, žádáme-li právě tohle zapojení. Přejeme-li si jenom jeden obvod DFF, pak musíme upravit kód. Z výukových účelů si zkusíme několik experimentů. Definujeme pomocný signál ds (od data storage), a to navzdory doporučení v kapitole 2.1.c na str. 12, že v procesech se mají upřednostňovat proměnné, a tak ještě víc zkomplikujeme obvod. Kvůli simulaci má ds inicializaci u své definice.

```
architecture rtl2 of DffQN is
signal ds:std_logic:='0';
begin
  process(CLOCK)
  begin
    if rising_edge(CLOCK) then
      ds<=Data; Q<=ds; QN<=not ds;
    end if;
  end process;
end architecture;
```

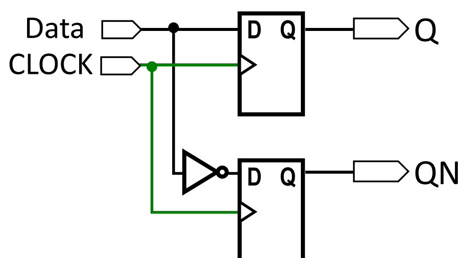


Obrázek 17 - DFF s Q a QN varianta nadbytečná -

Použili jsme tři přiřazení <= a každé z nich vloží vždy nového DFF, neboť se má provést až po aktivní hraně hodinového signálu.

Změníme-li ds na proměnnou[13], zápis do ní se již pokaždé nekonvertuje na vložení registru, ale jen při nutnosti pamatovat si momentální hodnotu proměnné. Zde se ds vždy přiřadí nová.

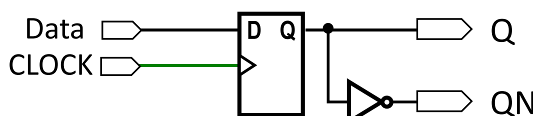
```
architecture rtl3 of DffQN is
begin
  process(CLOCK)
  variable ds:std_logic:='0';
  begin
    if rising_edge(CLOCK) then
      ds:=Data;
      Q<=ds; QN<=not ds;
    end if;
  end process;
end architecture;
```



Blokující přiřazení := uvnitř podmínce detekce hrany se implementuje vložením DFF jedině tehdy, pokud kód vyžaduje zapamatování momentální hodnoty proměnné.

Jeden DFF obvod dostaneme až po přesunu finálního přiřazení mimo podmínku detekce náběžné hrany, pak již definujeme pouhé propojky, do kterých se nevkládají DFF.

```
architecture rtl4 of DffQN is
begin
  process(CLOCK)
  variable ds:std_logic:='0';
  begin
    if rising_edge(CLOCK) then ds:=Data;
    end if;
    Q<=ds; QN<=not ds;
  end process;
end architecture;
```



Obrázek 18 - Optimální DFF s Q a QN

Lze přidat i další vstupy ACLRN a SCLEAR a dopsat jejich podmínky postupem z předchozí kapitoly.

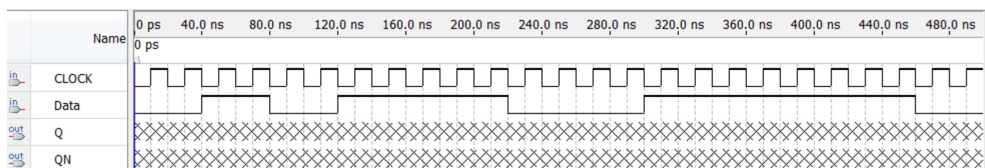
[13] Připomínáme, že **procesu se pracovní inicializace proměnných vkládají vždy do kódu**. U jejich definic se projeví leda v simulaci a v obvodu jen jako stav po zapnutí napájení, ale překladač je tady někdy odmítne jako neproveditelné.

3.6 *** Cvičná úloha 4: Registr s xor Data

Říkali jsme si, že architektura rtl3 (viz nahoře) nevloží další DFF, jelikož není potřeba pamatovat si hodnotu ds proměnné. Dokážete nakreslit, aniž byste využili překladač, jaké zapojení popisuje následující VHDL architektura, v níž se již ds hodnota musí nutně zapamatovat?

```
architecture rtl3_xor of DffQN is
begin
  process(CLOCK)
  variable ds:std_logic:='0'; -- power-up initialization only
  begin
    if rising_edge(CLOCK) then
      ds:=Data and not ds; Q<=ds; QN<=not ds;
    end if;
  end process;
end architecture;
```

Dokážete stanovit odezvu Q a QN na signály CLOCK a DATA uvedené dole i bez použití simulátoru?



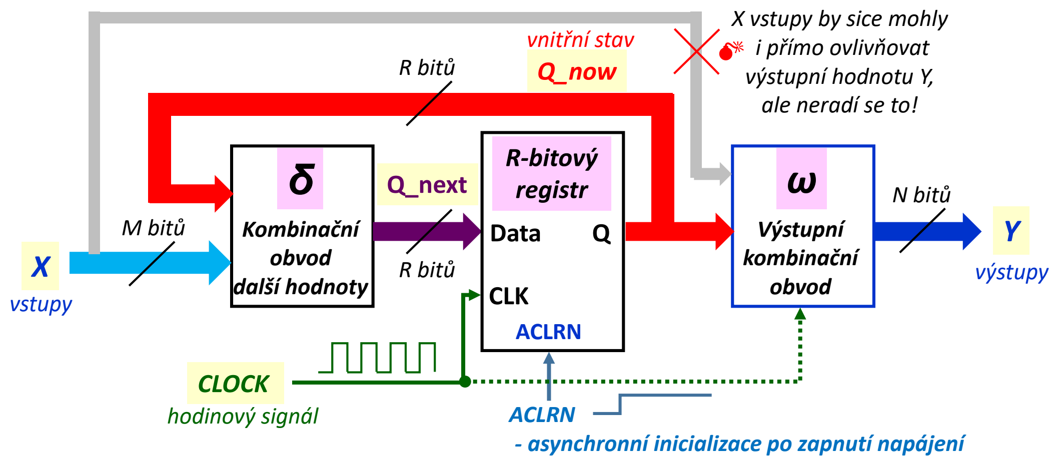
Návod: Nakreslete si schéma obvodu a pak ho rozkreslete na dvě jednodušší pomocí Shannonovy expanze[14] podle hodnoty vstupu Data. Z nich pak již snadno odvodíte chování obvodu.

[14] Expanze je známa též jako Boole's expansion theorem, viz https://en.wikipedia.org/wiki/Boole%27s_expansion_theorem

4 Základní synchronní obvody

4.1 Synchronní obvod se strukturou automatu

Značná část synchronních úloh se dá vyřešit strukturou podobnou konečného automatu, v angličtině nazývaného Finite State Machine (FSM), kterou naznačuje obrázek dole.



Obrázek 19 - Synchronní obvod se strukturou FSM (konečného automatu)

K pojmu FSM se především váže návrhová strategie na principu stavů, o níž se zmíníme později. Jednoduché obvody (třeba posuvné registry, čítače a další), sice používají uspořádání FSM, ale jeho bloky se výhodněji popíší přímo logickými výrazy[15]. Obvodová struktura na bázi FSM zahrnuje tři základní části:

- **Registr** obsahuje uloženou vnitřní hodnotu, tedy binární číslo o šířce R-bitů, kde R>0. Označíme ho třeba Q_now. V terminologii automatů se nazývá stavem, ale zatím v něm budeme vidět obyčejné binární číslo. U čítačů bude třeba jejich načítanou hodnotou.
  Hodnota registru se změní po aktivní hraně hodinového signálu (buď vždy vzestupné, či vždy sestupné), kdy se do něj nahraje připravená hodnota Q_next na jeho datovém vstupu.
- **ACLRN vstup** (asynchronní nulování registru po zapnutí napájení) se přidává jen k inicializaci, je-li nutná. Nesní se nepoužívat jako pracovní, viz diskuze v učebnici Logické obvody.
- **X vstupy** představují hodnoty, které se do obvodu posílají zvnějšku. Mají bitovou šířku M>=0. Je-li M=0, mluvíme někdy o autonomním obvodu, který pracuje bez možností svého ovlivnění.
- **Kombinační obvod δ** další hodnoty registru, funkce δ(Z, Q_now) vytváří ze vstupu X a současné hodnoty registru Q_now jeho novou hodnotu Q_next, která se přivádí na datový vstup registru.
- **Výstupní kombinační obvod ω** je funkcí ω(Q_now), která specifikuje výstup Y. Hodně obvodů obsahuje prosté přiřazení Y<=Q_now, ale lze užít i komplikovanější funkce. Výstupní funkce může používat i hodiny, ale většinou se obejde bez nich.
- **Vazba z X na Y** je sice možná, pak bude funkce ω(Q_now, X), ale ze vstupů X se pak přenášejí na Y nejen hazardy probírané v učebnici Logické obvody, ale hrozí i generace krátkých pulzů na výstupu Y po změně X v nevhodném časovém okamžiku. **V obvodech se radí ji nepoužívat**.
  Hazardy představují i vážný problém **ω** funkce, i když se v ní nepoužije vazba od X. Zatímco výstup registru je vždy bez nich, za ω se již mohou objevit.

[15] Velmi pokročilá poznámka: Ve stručnosti lze zhruba říct, že strukturou uvedenou na Obrázek 19 lze řešit úlohy, jejich činnost se dá vyjádřit orientovaným grafem přechodů. Například čítač 0,1,2, 0,1,2,... atd., můžeme popsat grafem se třemi vrcholy. Podobné úlohy též specifikuje regulární gramatika/výraz (viz https://en.wikipedia.org/wiki/Regular_grammar).

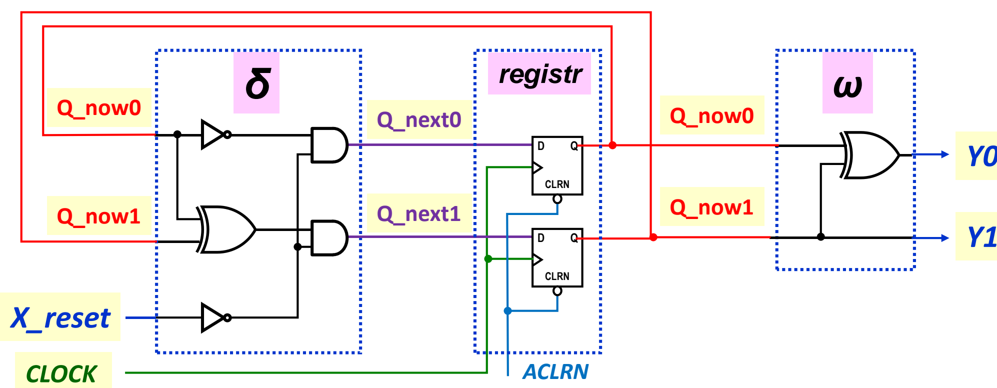
4.2 @ Příklad VI. - Grayův 2bitový čítač

Obrázek 20 ukazuje čítač v Grayově kódu, který se tak běžně realizuje překódování binárního čítače v případě více bitů. 2bitový lze řešit jinak, což si ukážeme v kapitole 4.6.a na str. 48. Kvůli demonstraci jsme předchozí strukturu použili i na něj. Kombinační funkce δ a ω navrhneme z tabulky jeho činnosti:

Ustálené hodnoty po náběžné hraně hodinového signálu						
ω(Q_now)		***Registr***		***δ(Q_now,X_reset)***		***Vstup***
Y1	Y0	Q_now1	Qnow0	Q_next1	Q_next0	X_reset
0	0	0	0	0	1	0
0	1	0	1	1	0	0
1	1	1	0	1	1	0
1	0	1	1	0	0	0
0	0	0	0	0	1	0
0	1	0	1	0	0	1
0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	1	0	1	1	0	0

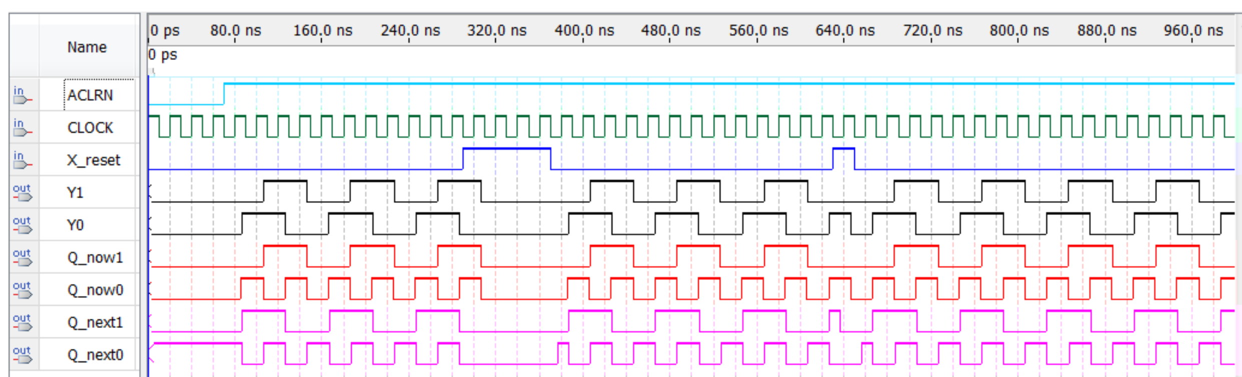
Tabulka 4 - Funkce Grayova čítače s překódováním výstupu

Kombinační obvod δ další hodnoty zde funguje jako sčítačka +1 s přidaným nulováním, je-li X_reset='1'. Výstupní funkce ω zase provádí překódování na Grayův kód. Hodnota Q_next se nahraje do registru až po hraně CLOCK, na obrázku dole náběžné, což znázorňuje i barevné stínování řádků tabulky. Po změně Q_now se v kombinačním obvodu δ(Q_now,X_reset) stanoví nová hodnota Q_next.



Obrázek 20 - Logické schéma Grayova čítače s překódováním výstupu

Průběh ukazuje i ukázka simulace na obrázku dole, která se úmyslně provedla s respektováním zpoždění obvodů, zde především DFF. Po náběžné hraně hodin CLOCK se mění Q_now a s ním i Q_next a Y.



Obrázek 21 - Časová simulace Grayova čítače

Všimněte si, že X_reset se mění mimo náběžnou hranu CLOCK, aby se dodržel požadovaný předstih a přesah dat na datovém vstupu DFF, což jsme již rozebírali v učebnici Logické obvody.

4.2.a VHDL kód

Ve VHDL kódu popíšeme náš příklad rychleji než kreslení schématu, jak naznačuje obrázek dole, v němž jsme graficky orámovali příslušné funkce:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity GrayCntr2 is
port (CLOCK, X_reset, ACLRN: in std_logic;
      Y: out std_logic_vector(1 downto 0));
end entity;
architecture rtl of GrayCntr2 is
begin
   process(CLOCK, ACLRN)
   constant ZERO:unsigned(Y'RANGE):=(others=>'0');
   variable rg:unsigned(Y'RANGE):=ZERO;
   begin
      if ACLRN='0' then rg:=ZERO;
      elsif rising_edge(CLOCK) then
         if X_reset='1' then rg:=ZERO;
         else rg:=rg+1;
         end if;
      end if;
      Y(1)<=std_logic(rg(1));
      Y(0)<=std_logic(rg(1) xor rg(0));
   end process;
end architecture;
```

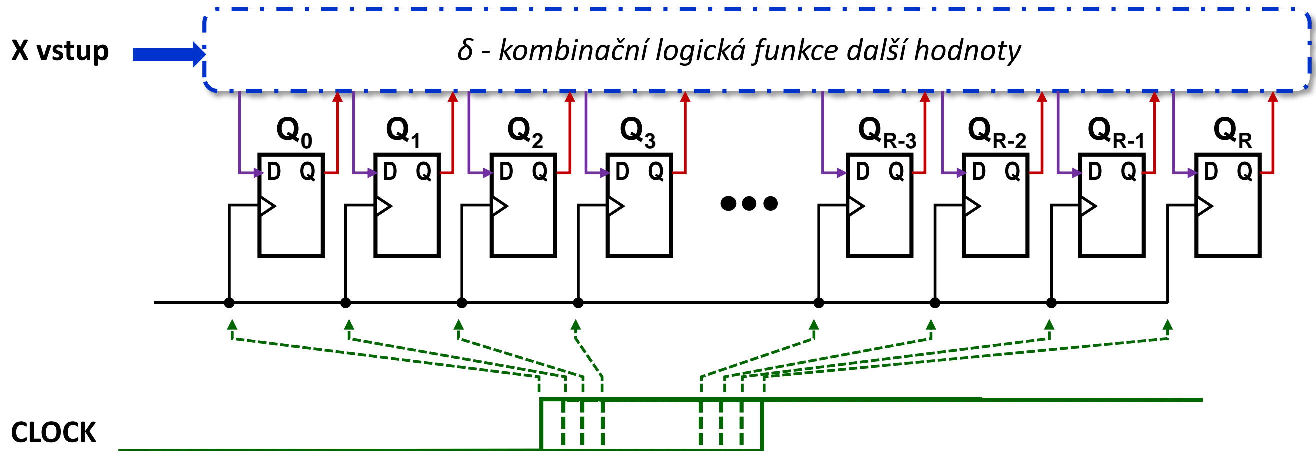

Obrázek 22 - VHDL kód s funkcemi δ a ω

- Náš VHDL kód začíná knihovnami a běžnou entitou, která definuje vstupy a výstupy.
- Synchronní obvod popíšeme v architektuře v sekci procesu. V jeho sensitivity listu bude vždy hodinový signál, zde CLOCK, a v případě asynchronního nulování se rovněž přidá i jeho vstup, zde signál ACLRN, protože se testuje před detekcí náběžné hrany. (Za ní ani nemůžeme, má vyšší prioritu.) Pokud by se mimo synchronní sekci používaly i nějaké jiné signály, též se musí vložit[16].
- Signál X_reset chybí v sensitivity list procesu, protože jeho změna nevyvolá nový výstup. Ovlivní ho až po náběžné hraně hodin. (Připomínáme, že sensitivity list neobsahuje všechny vstupní parametry, neboť nemá nic společného s hlavičkou funkce klasického programování.)
- Registr si žádá definici pomocné proměnné rg. Přehlednost jsme zvýšili i konstantou ZERO. Nulujeme jí rg hned v definici, ale výhradně kvůli simulátoru. Syntéza obvodu by ji zde v případě procesu použila leda jako výchozí stav po zapnutí napájení, viz text u Obrázek 5 na straně 22. V našem příkladu inicializujeme v kódu signálem ACLRN.
- Samotné vložení registru provedeme přidáním detekce hrany hodin. Nepotřebujeme-li ACLRN, pak sekci začneme příkazem: `if rissing_edge(CLOCK) then` ...
- V synchronní sekci se specifikují operace funkce δ, která stanoví další hodnotu registru. K její realizaci jsme využili sčítání a nechali na Quartusu, aby si sám navrhnul logické rovnice sčítačky +1.
- Za synchronní sekcí vidíme ω výstupní funkci, zde se nachází, pokud nepotřebuje hodiny. Hodnotu v registru překóduje na požadované výstupy pouhou kombinační logickou funkcí.

[16] *Poznámka pro úplnost: Proměnné definované v procesu se samozřejmě nedávají do sensitivity listu - v něm ještě nejsou definované. Jejich hodnoty se navíc změní jen vnějšími vlivy.*

4.3 Záludnosti synchronních obvodů - zpoždění hodin a rychlé smyčky

Uvedená struktura vypadá velmi jednoduše a vnucuje domněnku, že stačí vytvořit vhodné funkce a snadno vyřešíme úlohu. Ve skutečnosti návrhové prostředí musí pečlivě hlídat distribuci hodin a příliš rychlé smyčky, jak nastiňuje obrázek dole.



Obrázek 23 - Skluz (**slack**) v distribuci hodin

Hodinový signál CLOCK se uvnitř obvodu šíří po interních vodičích a při nevhodném uspořádání by se mohlo přihodit, že některé klopné obvody obdrží aktivní hranu hodin dříve než jiné. Na obrázku se aktivní hrana CLOCK šíří od DFF označeném jako Q0, k němuž dorazí nejdřív. Naposledy (na uvedeném obrázku) pronikne ke QR.

Q0 tak získá novou hodnotu jako první ze všech, která se ihned přenese na vstup kombinačního obvodu δ. Ten začne okamžitě vytvářet nový výstup Q_next celému registru, tedy nejen DFF datovému vstupu s indexem Q0, ale i všem ostatním.

Když na DFF s indexem QR konečně dorazí aktivní hrana hodin, na jeho datovém vstupu může být nějaká neustálená mezihodnota (daná kombinací Q výstupů DFF již překlopených, a dosud ne), která se do něho nahraje místo požadované. Náš obvod, zdánlivě správný, nebude fungovat.

Jak se tohle řeší? Pokud napíšeme vhodný popis ve VHDL, návrhové prostředí se pokusí najít optimální rozložení klopných obvodů, aby všechny dostávaly hodiny přibližně ve stejný okamžik, třeba s užitím jejich kruhového uspořádání, nebo přidáním zpožďujících členů. V závěru proběhne časová analýza, která ověří, dodržení dob předstihu a přesahu na vstupech DFF vůči hodinám. Nezdaří-li se kontrola správnosti, musíme se zamyslet nad návrhem a zkusit ho modifikovat.

Terminologie:

- Zpoždění hodin má anglický termín „**slack**" (zde ve významu deficit, skluz dodávky).
- Hodinová doména, **clock domain**, je skupina obvodů závislých na jednom hodinovém signálu[17].

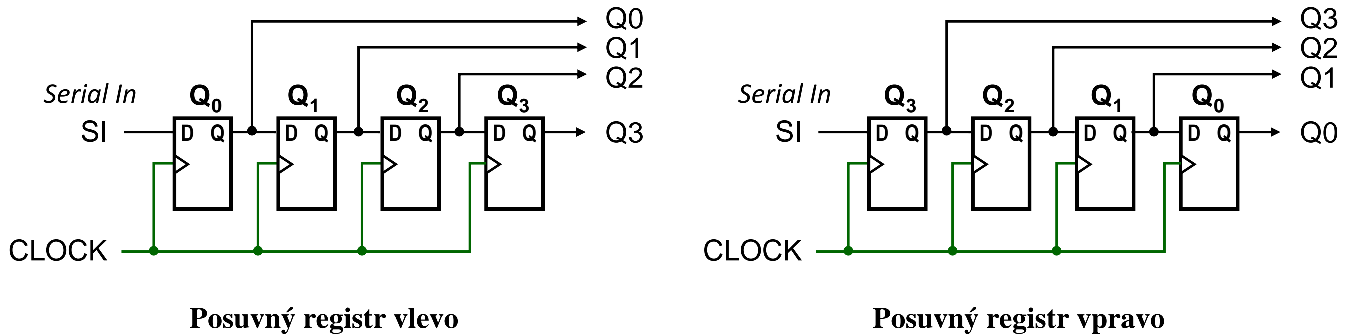
4.4 *** Cvičná úloha 5: Grayův čítač obecné délky

Zadání: Zkuste si dle vzoru předchozího příklad napsat obecný Grayův čítač, který má na výstupu bitovou délku zadanou pomocí parametru typy generic. Nezapomeňte na kontrolu správnosti hodnot parametru pomocí pasivního procesu v sekci entity. A napište ho celý z hlavy jen s využitím VHDL šablony. Pozn. Obrázek 22 je opravdu obrázek. Nejde okopírovat !

[17] Zatím budeme pracovat v jediné hodinové doméně. Přenesení nějakého signálu z jedné hodinové domény do druhé tvoří další náročnou otázku, kterou si necháme na později. Řeší se synchronizátory či handshake obvody. Hloubaví čtenáři se mohou zatím podívat třeba na http://www.fpga-faq.com/FAQ_Pages/cdc_wp.pdf

4.5 Posuvné registry

Posuvné registry patří mezi nejjednodušší obvody. Jejich funkce δ další hodnoty se tvoří pouhým propojením a maximální přípustná frekvence nezávisí na jejich bitové délce, neboť u nich se snadno vyřeší problémy s distribucí hodin, jelikož jejich vstup klopného závisí jen na předchozím v řadě.



Obrázek 24 - 4-bitový posuvný registr

Směr posunu se specifikuje jen v případě, když jsme zavedli číslování výstupních bitů. Nikdy se neurčuje podle směru kresby na symbolickém schématu, ale dle organizace binárních čísel. Posun doleva je daný tím, že bit s nižší váhou se posunuje na vyšší (analogický C bitové operaci <<), tedy výsledek násobení 2, zatímco opačný směr (analogický C bitové operaci >> s číslem bez znaménka), odpovídá dělení 2.

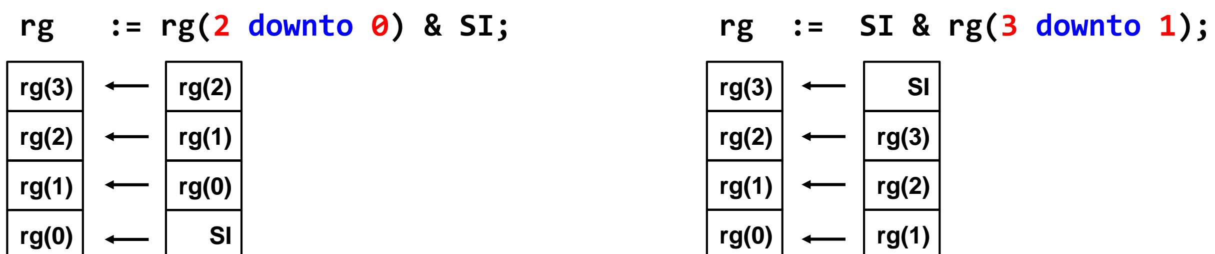
```
library ieee; use ieee.std_logic_1164.all;
entity Shift4 is
      port (CLOCK, SI: in std_logic;
            Q: out std_logic_vector(3 downto 0));
end entity;
architecture rtl of Shift4 is
begin
   process(CLOCK)
      variable rg:std_logic_vector(Q'RANGE);
      begin
         if rising_edge(CLOCK) then
```

-- shift left	-- shift right
`rg:= rg(2 downto 0) & SI;`	`rg:= SI & rg(3 downto 1);`

```
         end if;
         Q<=rg;
   end process;
end architecture;
```

Obrázek 25 - 4-bitový posuvný registr

VHDL kód obou verzí se liší v jediném řádku, a to sestavení nové hodnoty vnitřního registru. V kód je použitý VHDL operátor & sjednocení řetězců, jehož funkci naznačuje i obrázek dole.



Obrázek 26 - Operátory & v posuvném registru

4.5.a @ Příklad VII. - Obousměrný 4-bitový posuvný registr

Pokud chceme vytvořit jeden obousměrný posuvný registr, stačí do VHDL přidat další vstup na řízení směru, třeba RightDir, který při '1' zapne posun doprava, zatímco při '0' se bude posouvat doleva, k čemuž využijeme již probrané příkazy. Zavedeme zde i dva vstupy, `LeftIn` při směru doleva, zatímco `RightIn` při posunu doprava. Doplníme rovněž asynchronní nulování.

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

entity Shift4bidir is
  port (CLOCK, ACLRN : in std_logic;              --  clock and asyn. clear
        RightDir : in std_logic;                   -- if RightDir='1', right shift
        LeftIn, RightIn : in std_logic;            -- serial-in for left and right shift
        Q: out std_logic_vector(3 downto 0));      -- output
end entity;

architecture rtl of Shift4bidir is
begin
  process(CLOCK)
  variable rg:std_logic_vector(Q'RANGE);
  begin
    if ACLRN='0' then rg:=(others=>'0');
    elsif rising_edge(CLOCK) then
      if RightDir='1' then
        rg:= RightIn & rg(3 downto 1);
      else
        rg:= rg(2 downto 0) & LeftIn;
      end if;
    end if;
    Q<=rg;
  end process;
end architecture;
```

4.5.b *** Cvičná úloha 6: Posuvný obecné délky s nulováním a nastavením

Zadání: Okopírujte si předchozí kód a změňte ho na posuvný registr obecné délky. Přidejte asynchronní i synchronní nulování a také datový vstup stejné délky jako vnitřní registr, z něhož se může nastavit při náběžné hraně hodin. Jeho entita by mohla vypadat takto:

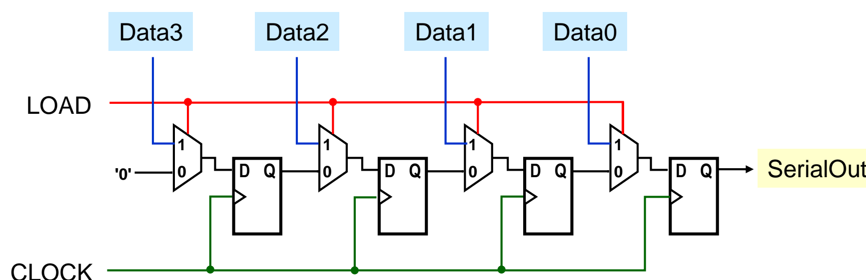
```
library ieee; use ieee.std_logic_1164.all;
entity ShiftBidir is
  generic(MAXBIT:integer:=7);
  port(CLOCK, ACLRN : in std_logic;                  -- clock and asynchronous clear
       Reset : in std_logic;                         -- if Reset='1' then clear shift register
       SLoad : in std_logic;      -- if SLoad='1' and Reset='0 then load value from Data input
       Data: in std_logic_vector(MAXBIT downto 0);   --data input for SLoad
       RightDir : in std_logic;   -- if SLoad='0' and Reset='0 then on RightDir='1' right shift, else left shift
       LeftIn, RightIn : in std_logic;               -- serial-in for left and right shifts
       Q: out std_logic_vector(MAXBIT downto 0));    -- output
end entity;
```

Zadání lze snadno vyřešit jednoduchými úpravami. V synchronní sekci se hodí kaskáda multiplexorů if-elsif ...

4.5.c Posuvné registry k sériovému přenosu

Posuvné registry nabízejí vynikající vlastnosti ohledně rychlosti, nezávislé na jejich délce, a snadné řešení správné distribuce hodinových signálů, což jsme diskutovali v kapitole 4.3 na straně 44, dovolují dosáhnout rychlosti přenosu v řádu gigahertzů. Nejčastěji se používají k sériové komunikaci, které patří dnes mezi nejrozšířenější přenosy dat. Používají je třeba všechny sériové sběrnice od PCIe až po DVI a DisplayPort video přenosy a mnohá další zařízení.

Převodník na sériový přenos získáme, pokud do posuvného registru přidáme možnost nahrát data. Lze zvolit jak posuvný registr doprava (obrázek dole) či doleva, opět rozdíl bude jen v očíslování bitů. Závisí na nás, zda si přejeme vysílat od nejnižšího k nejvyššímu bitu, či naopak.



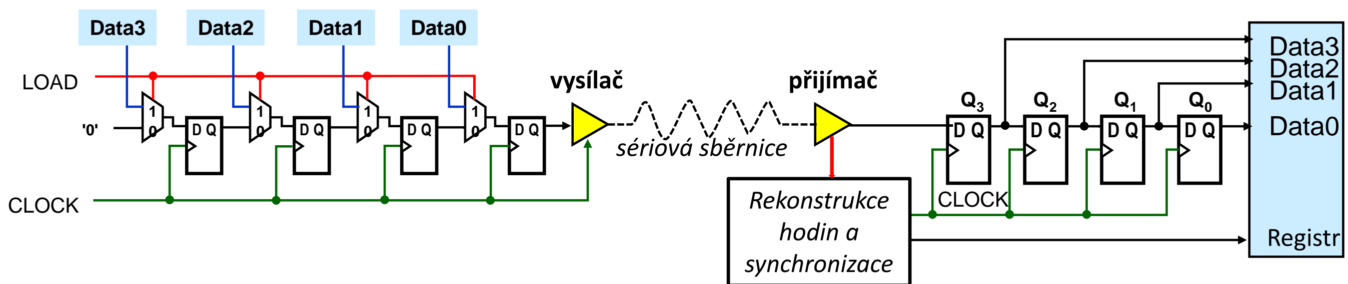
Obrázek 27 - 4-bitový převodník paralelní z paralelního přenosu na sériový

VHDL kód získáme pouze drobnou úpravou předchozího příkladu. Provedeme jen změny entity, do níž vložíme požadované vstupy. Dále upravíme pouze definici funkce δ, která je uvnitř synchronní sekce, a upravíme výstupní funkce ω určující, co se pošle z procesu do dalších obvodů.

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

entity Shift4Load is
   port (CLOCK, Load : in std_logic;          -- clock and synchr. load
         Data : in std_logic_vector(3 downto 0);      -- data in
         SerialOut: out std_logic);           -- output
end entity;

architecture rtl of Shift4Load is
begin
   process(CLOCK)
   variable rg:std_logic_vector(Data'RANGE);
   begin
     if rising_edge(CLOCK) then
        if Load='1' then rg:=Data;
        else
          rg:= '0' & rg(3 downto 1);
        end if;
     end if;
     SerialOut<=rg(0);
   end process;
end architecture;
```

Princip sériové komunikace pak naznačuje Obrázek 28 na straně 48. Sběrnice používá vysílač, dost často s formátem přenosu, který šikovným obvodem dovolí i rekonstruovat frekvenci, takže není nutné pak přenášet i hodinový signál, což opět zrychlí přenos, jelikož se nemusí hlídat případný posun fáze hodin mezi přijímačem a vysílačem.

Nejjednodušším způsobem přenosu je třeba RS232 protokol, který dnes ustupuje do pozadí, ale má jednoduchý princip založený na start stop bitech, takže se stále dá najít v jednodušších aplikacích. Pokročilejší technikou je třeba diferenční Manchester kódování na principu posunu fáze, které používá i internet.

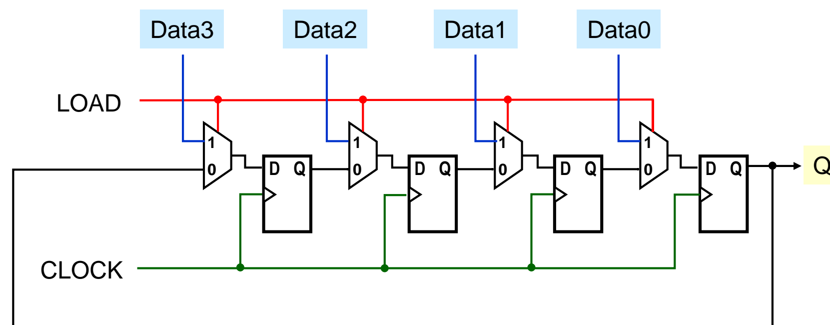


Obrázek 28 - Princip sériové komunikace při 4-bitovém přenosu

Téma vlastní realizace sériové sběrnice spadá do oblasti telekomunikací a přesahuje rámec této publikace. Vlastní vysílače a přijímače není třeba navrhovat. Existují jejich četná již hotová a ověřená řešení jako přídavné součastky, některá FPGA je i obsahují v sobě, takže stačí připojit posuvné registry.

4.6 Kruhový čítač (posuvný registr)

Pokud spojíme výstup posuvného registru s jeho vstupem, dostaneme kruhový čítač (ring counter).



Obrázek 29 - Kruhový čítač

V kódu uvedeném na předchozí stránce nahradíme jen příkaz za else

```
rg:= '0' & rg(3 downto 1);
```

za nový popis

```
rg:= rg(0) & rg(3 downto 1);
```

Podobný obvod donekonečna vysílá jednou nahraná data, a tak se hodí jako generátor různorodých sekvencí. Délka může samozřejmě být libovolná.

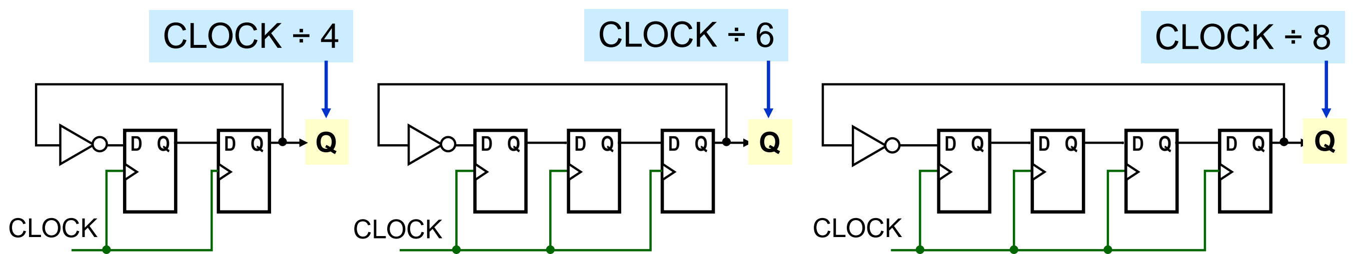
4.6.a Johnsonův čítač

Pokud do zpětné vazby vložíme invertor, dostaneme další zajímavý obvod, kterému se říká Johnsonův čítač. Anglické názvy jsou i twisted ring counter, switch-tail ring counter, walking ring counter, Möbius counter.

Dvoubitový čítač dává stejný výstup jako 2-bitový Grayův čítač "00", "10", "11", "01", *"00", "10",...* . Tříbitový dává jen 6 výstupních hodnot: "000", "100", "110", "111", "011", "001", *"000", "100",...* Čtyřbitový jich bude mít jen 8: "0000","1000","1100","1110","1111","0111","0011","0001",*"0000","1000,...*

Počet hodnot výstupů je tedy vždy rovný dvojnásobku jeho délky. Hlavní výhodou Johnsonova čítač je jeho rychlost, která vyplývá ze založení na posuvném registru, a tak se nasazuje jako první stupeň při dělení hodně vysokých frekvencí anebo případně jejich čítání.

5-bitový čítač nabývá deseti hodnot, takže deseti dělí vstupní frekvenci. Nemusí se nijak inicializovat po zapnutí napájení, jelikož se z jakéhokoli počátečního stavu vždy dostane do správné sekvence.

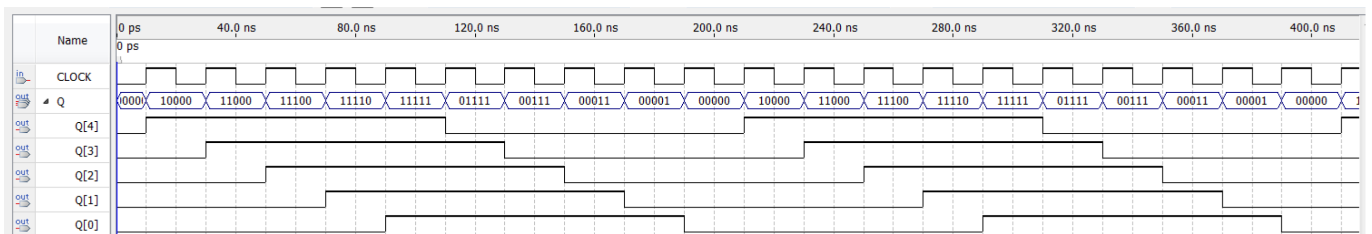


Obrázek 30 - Johnsonův dělič frekvence - 4, 6 a 8

VHDL kód napíšeme velmi snadno a klidně i s generic sekcí, čím dostaneme univerzální obvod. Kvůli tomu si raději vyvedeme všechny bity. Podobá se předchozímu příkladu s nepatrnou úpravou funkce δ další hodnoty registru rg.

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

entity Johnson is
   generic(BIT_LENGTH:integer:=5);
   port(CLOCK : in std_logic;        -- clock
        Q: out std_logic_vector(BIT_LENGTH-1 downto 0)); -- output
end entity;

architecture rtl of Johnson is
begin
   process(CLOCK)
   variable rg:std_logic_vector(Q'RANGE);
   begin
      if rising_edge(CLOCK) then
           rg:= not rg(0) & rg(rg'HIGH downto 1);
      end if;
      Q<=rg;
   end process;
end architecture;
```

Simulace prokáže další zajímavou vlastnost. Chceme-li Johnsonův čítač využívat jako dělič velmi vysoké frekvence, pak jeho výstup Q můžeme vyvést z libovolného bitu vnitřního registru. Na všech bude vstupní frekvence dělená 2*BIT_LENGHT. V simulaci se použilo CLOCK 50 MHz, a tak na všech výstupech je 5 MHz, a navíc se symetrickým výstupem, tedy se střídou signálu 1:1, čili 50 % DLC (Duty Cycle).



Obrázek 31 - Výstupy Johnsonova 5-bitového čítače

4.6.b *** Cvičná úloha 7: Variabilní dělič

Zadání: Navrhněte dělič frekvence, u něhož lze přepínat dělicí poměr vstupem IXDIV.

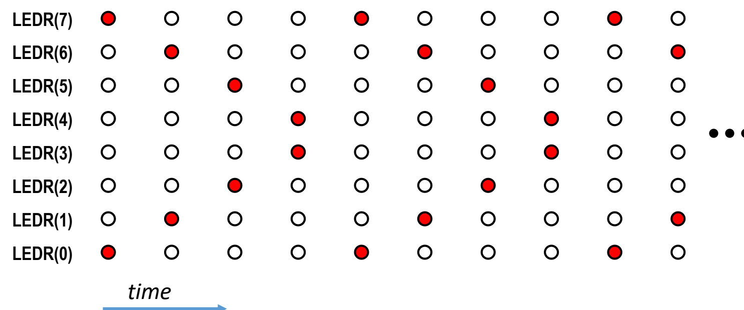
Hodnota IXDIV	"00"	"01"	"10"	"11"
Dělicí poměr	6	8	10	12

Návod: Využijte jeden Johnsonův čítač vhodné délky, u něhož budete měnit, odkud se vede zpětná vazba přes invertor.

Otázka: Může se při změně dělicího poměru objevit někdy pulz, ať v '0' či v '1', který bude kratší, než by odpovídalo průběhu při hodnotě IXDIV, která je nižší z obou měněných?

4.7 @ Příklad VII. - Navigační světlo

Zadání: Na osmi diodách vytvořte navigační světlo k navádění do středového bodu, což se běžně používá. Svislá se hodí k výškovému navádění, třeba létajících objektů, vodorovná zase k přesnému směrování do úzkého prostoru, jako lodí do průlivu. Pohybující se světla naznačí směr lépe než statické šipky.

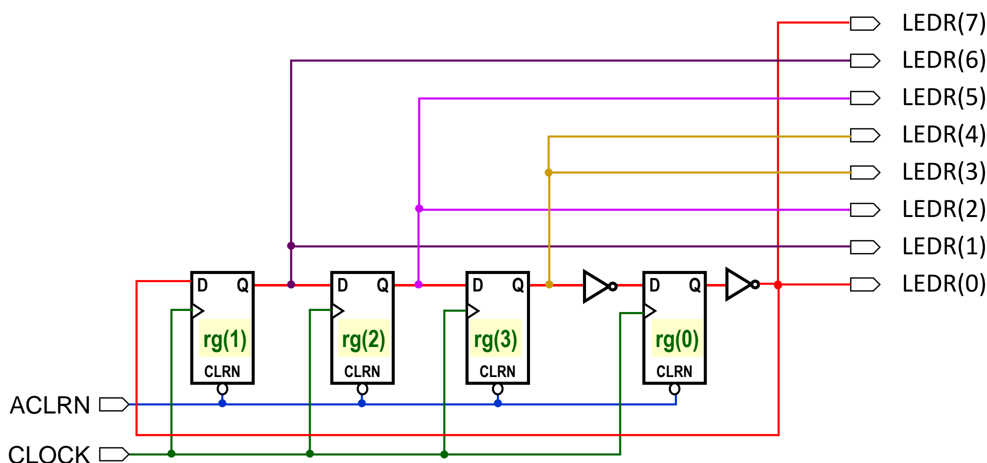


Obrázek 32 - Navigační světlo

Pokud se podíváme na průběh, v prvním přiblížení lze vytvořit světlo pomocí kruhového posuvného registru, a nepotřebujeme ani dva, stačí nám jeden, jehož výstupy rozvedeme na obě čtveřice.

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity Shift4Sign is
  port (CLOCK, ACLRN: in std_logic;
        LEDR: out std_logic_vector(7 downto 0));
end entity;
architecture rtl of Shift4Sign is
begin
  process(CLOCK, ACLRN)
  variable rg:std_logic_vector(3 downto 0);
  begin
      if ACLRN='0' then  rg:="0001";
      elsif rising_edge(CLOCK) then   rg:= rg(2 downto 0) & rg(3);
      end if;
      LEDR(3 downto 0)<=rg(3 downto 0);
iloop: for i in 0 to 3 loop LEDR(7-i)<=rg(i);
      end loop;       -- We used for-loop because we cannot change range directions in vectors,
         -- e.g. LEDR(7 downto 4)<=rg(0 to 3); results in an error!
  end process;
end architecture;
```

Kód vytvoří schéma na dolní obrázku.



Obrázek 33 - Navigační světlo za předpokladu užití DFF jen s asynchronním nulováním

Řešení se opírá o kruhový posuvný registr, do něhož se asynchronní inicializací nahraje výchozí hodnota. Předpokládali jsme, že FPGA obsahuje pouze klopné obvody typu DFF, které mají výhradně asynchronním nulování (nejčastější případ). Před a za registr rg(0), u něhož se v kódu předepisuje nastavení do '1', se kvůli tomu vloží invertory.

Ve výstupní funkci ω (část po skončení synchronní sekce) se spodní bity LEDR přiřadí snadno příkazem: `LEDR(3 downto 0)<=rg(3 downto 0);` Na horní není ale možné použít *LEDR(7 downto 4)<=**rg(0 to 3)**;*

VHDL nedovoluje obrátit směr v rozsahu vektoru, aby se vyloučila chybná použití. Pokud si opravdu přejeme něco podobného, pak si lehce napíšeme cyklus:

```
iloop: for i in 0 to 3 loop LEDR(7-i)<=rg(i); end loop;
```

který se bude realizovat jeho pouhými propojkami a nepřidá žádnou logiku.

4.7.a Robustnost řešení

Předchozí řešení by se dalo použít leda tak do levné hračky či jiného nekritického objektu. Nehodí se na družici nebo na marťanské vozítko. Ve skutečnosti by se nemělo využívat ani jako pobřežní maják.

Co je na něm špatně? **Není robustní!** Pomocný registr rg obsahuje 4-bitovou hodnotu, takže může nabývat celkem 16 stavů, z nichž využíváme pouze čtyři, a to "1000", "0100", "0010" a "0001". Po zapnutí napájení ho sice inicializujeme, takže by teoreticky měl cyklovat pouze mezi těmito hodnotami.

Může se však stát, že se rušením překlopí nějaký bit posuvného registru. V něm pak bude nadále cirkulovat chybná hodnota až do nového vypnutí za zapnutí, které se u dětské hračky provede snadno, ale na Marsu půjde o problém stejně závažný jako selhání během navádění lodě.

Podobné poruchy způsobí třeba blízký elektrický výboj, který na vodiče indukuje tím více energie, čím jsou delší. V obvodech ohrozí především rozvod napájení a země a samozřejmě delší vstupní přívody. Indukce vytvoří dočasné zhoupnutí či špičku, během níž se může mylně interpretovat nějaká '0' jako '1', či obráceně. I když se obvod trvale nepoškodí, změní se uložená informace. Uživatelé počítačů dobře znají podobné jevy vedoucí zpravidla na zamrznutí celého operačního systému.

V kosmu zase hrozí energetické špičky od průletů silně ionizovaných částic, třeba kosmických paprsků, elektronů nebo protonů, ale publikace popisují i mnoho případů, kdy se totéž přihodilo i na povrchu Země, například po oslabení ozonové vrstvy.[18]

Před trvalou poruchou se lze chránit redundancí, tj. použitím několika stejných dílů, z nichž se majoritou vybírá výsledek.[19] Naše navigační světlo však nezvládne ani dočasnou změnu, s níž by si měl robustně navržený obvod poradit bez potíží a sám se co nejdříve vrátit do správného stavu.

Jak to obvod opravíme? Nejlépe změnou návrhu. Zvolíme zapojení, které nemůže uvíznout v nechtěných stavech. Jednou z mnoha možností je využít 2-bitový registr rg, který nabývá jen čtyř hodnot, a tak u něho můžeme, ale nemusíme, vynechat asynchronní ACLRN inicializaci. Zapojení zůstane plně funkční, ať se registr po zapnutí ocitne v kterémkoli stavu, což je i znakem jeho robustnosti.

Můžeme inovaci navrhnout v mnoha variantách, z nichž zmíníme jenom dvě. V obou jsme přidali i počáteční hodnotu za definicí proměnné rg, ale výhradně kvůli lepší simulaci. V syntéze se nijak neprojeví.

---

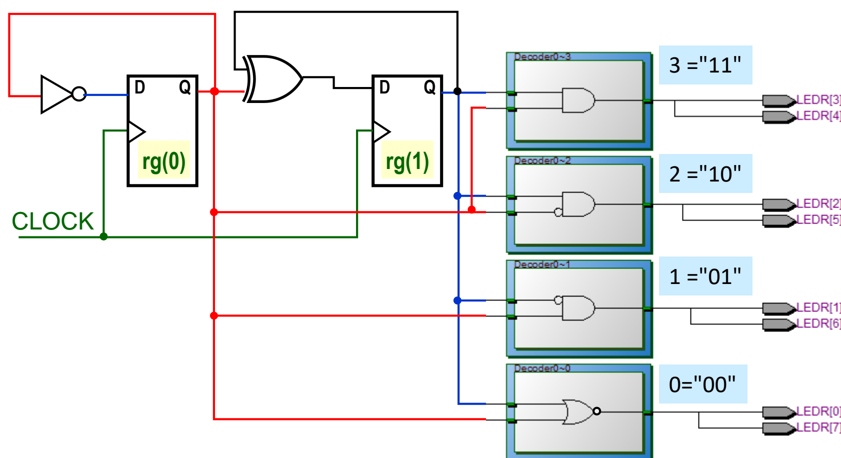
[18] V kosmické terminologii se vliv radiace na obvod označuje jako SEU (Single Event-Upset), jde-li o přechodnou, tzv. „měkkou" chybu. Trvalá změna provozu zařízení se pak nazývá SHE (Single Hard Error), jako třeba zaseknutý bit v paměťovém zařízení. SEE (Single Event-Effect) pak značí jakýkoli měřitelný účinek na funkci obvodu v důsledku SEU a SHE.

[19] V předchozím dílu učebnice jsme si navrhovali logickou funkci majority. Většinový výběr se mnohdy uplatňuje i na číselné hodnoty, kdy se příliš odchýlené vyloučí z výběru, což statistika zná jako „vylučování extrémních hodnot" (rejections of outliers). Ze zbytku se třeba vytvoří průměr.

První návrh využívá binární unsigned čítač, po němž následuje část, kterou nazýváme výstupní ω - funkcí. V ní popíšeme dekodér 1 ze 4, v němž využijeme toho, že v sekvenčním zdrojové doméně smíme přepisovat výstupy, a tak napřed celé LEDR vynulujeme, a poté ty požadované nastavíme do '1'.

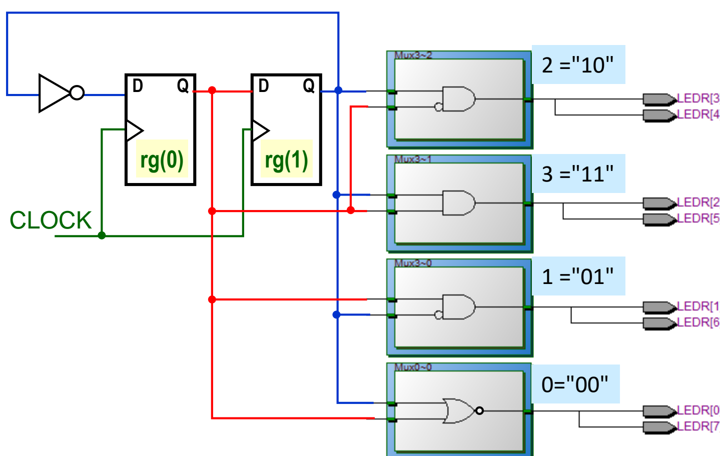
Druhý kód vychází z Johnsonova čítače, o němž z kapitoly 4.6.a víme, že jeho výstupem bude Grayův kód. Jeho výstupní funkci obsahuje case příkaz, který se napřed interpretuje multiplexory, s nimi se objeví v prvním kroku překladu (ten nám zobrazí RTL Viewer). Po následné minimalizaci, jejíž výsledek zobrazuje Technology Map Viewer, se rovněž převede na dekodér 1 ze 4.

```
architecture rtl_robust of Shift4Sign is
begin
  process(CLOCK)
  variable rg:unsigned(1 downto 0):="00";
  variable ix:integer range 0 to 3;
  begin
      if rising_edge(CLOCK) then
        rg:= rg+1;
      end if;
      -- decoder one hot
      ix:=to_integer(rg);
      LEDR<=(others=>'0');
      LEDR(ix)<='1'; LEDR(7-ix)<='1';
  end process;
end architecture;
```



Obrázek 34 - Navigační světlo s binárním čítačem

```
architecture rtl_robust1 of Shift4Sign is
begin
  process(CLOCK)
    variable rg:unsigned(1 downto 0):="00";
    begin
      if  rising_edge(CLOCK) then
          rg:= rg(0) & not rg(1);
      end if;
      LEDR<=(others=>'0');
      case rg is
        when "00" => LEDR(0)<='1'; LEDR(7)<='1';
        when "01" => LEDR(1)<='1'; LEDR(6)<='1';
        when "11" => LEDR(2)<='1'; LEDR(5)<='1';
        when "10" => LEDR(3)<='1'; LEDR(4)<='1';
      end case;
    end process;
end architecture;
```



Obrázek 35 - Navigační světlo s Johnsonovým čítačem

Které řešení vybrat? Obě vedou na skoro totožné zapojení, jen s přeházenými obvody výstupní funkce, [20] a tak odpověď závisí na tom, jaké klademe podmínky na výsledek.

- Chceme-li robustní zapojení, které rychle napíšeme a snadno rozšíříme i na více výstupních bitů, pak zvítězí navigační světlo s binárním čítačem.

[20] Přesněji obě zapojení jsou téměř stejná. Řešení s binárním čítače chce 6 LE, zatímco s Johnsonovým jen 5 LE. Rozdíl vznikl pouze tím, že binární čítač je složitější o XOR, takže nelze logickou funkci před rg(1) využít na výstupní hodnotu, což je ovšem zcela zanedbatelná diference, která se v praxi vůbec neuvažuje. Zmiňujeme se o ní jedině pro exaktní úplnost.

- Žádáme-li robustní zapojení s minimem rušivých pulzů na výstupech, pak vybereme navigační světlo s Johnsonovým čítačem, jehož výstup v Grayově kódu se vždy mění pouze v jednom bitu, takže bude vyšší naděje, že v navazující logice nevzniknou hazardy.
- Trváme-li z nějakého důvodu na zcela čistých výstupech bez zákmitů, pak výstupní funkce ω nesmí obsahovat logiku s více vstupy. Smíme v ní užít nejvýše invertory. Vrátíme se tak k posuvnému registru, který zaručí čisté výstupy, což je jeho další předností.
  Jeho robustnost zvýšíme přídavnými obvody. Jednou z mnoho možností bude do posuvného registru doplnit synchronní inicializaci na hranu hodin. (Asynchronní ACLRN nesmíme nikdy používat jako pracovní!) Dále vytvoříme řídicí 2-bitový čítač, který bude do posuvného registru periodicky nahrávat novou hodnotu, s níž se pak třikrát provede posun, a znovu se vykoná nové nahrání[21]. Výsledek se pak může klidně vystřelit i na Mars:-)

4.8 Čítače a děliče

Zatím jsme využívali čítače s vnitřními registry typu unsigned, které po dosažení nejvyšší hodnoty přetekly přes rozsah na nuly a pokračovaly od ní, a tak čítaly stále dokola. Pokud chceme zkracovat jejich cyklus, pak potřebuje též porovnávat jejich okamžitou hodnotu, což se někdy provádí lépe s čísly integer.

Čítače a děliče mají stejné vnitřní zapojení. Zpravidla nepoužívají logiku ve své výstupní ω-funkci, ale vyvádí ven přímo hodnoty vnitřního registru.

Jejich δ-funkce následující hodnoty obsahují přičítání, zpravidla +1 u čítačů nahoru, nebo odčítání, -1 u čítačů dolů. U obousměrných se mezi nimi přepíná. Jejich cyklu se zkracuje testem, že se již dosáhlo maximální žádané hodnoty a mají se vynulovat, čím začnou od začátku.

Označme frekvenci hodinového signálu přivedeného na vstup CLK jako f_{clk}. Vyvedeme-li výstup z nejvyššího aktivního klopného obvodu interního registru čítače, pak bude opakovaně měnit z ´0´ na ´1´ a z ´1´ na ´0´. Frekvenci jeho změn označme jako f_{out} . Pokud binární čítač čítá po 1 v rozsahu 0 až M-1, kde M je jakékoliv přirozené číslo větší než 1, pak čítač současně dělí i vstupní frekvenci číslem M:

$$f_{out} = \frac{f_{clk}}{M}$$

Oba obvody se tak liší nejvíce jenom výsledným použitím. V literatuře se proto často používá slovo dělič (frekvence) jako synonymum pro binární čítač po 1, neboť návrh je totožný. Každý čítač může bez úprav fungovat i jako dělič. Z čítače se na výstup vyvádí celá hodnota vnitřního registru. Pokud se zapojení zamýšlí výhradně jako dělič, pak se volí jen čítání nahoru a jeden výstupní bit.

Čísla integer často zjednoduší návrh, ale hodí se jim zadat **rozsah** (range). Jde o typ skalární typ, který nemá žádnou bitovou délku, na rozdíl od unsigned a signed typů. Nezaměňujte s jazykem C, v němž existují přesné bitové délky integer dané vnitřní interpretací v procesoru.

V obvodech může integer mít **jakoukoli bitovou délku** větší než 0. Pokud však překladač kvůli „naší chybě" špatně odhadne, v jakém rozsahu se integer proměnná používá, protože jsme mu neposkytli dostatek informací, pak vytvoří reprezentaci v nějaké předdefinované délce, zpravidla 32 bitů, čímž naroste nejen počet registrů, ale i velikost veškeré navazují logiky, protože i sčítačky a komparátory budou pak 32-bitové. Obvod nám vyjde zbytečně velký a pomalý.

---

[21] Pomocný dvoubitový čítač bude ve skutečnosti fungovat jako konečný automat (FSM - Finate State Machine). Ty budou tématem pozdější kapitoly.

Maximální frekvence hodinového vstupu CLK u čítačů/děličů klesá samozřejmě s bitovou délkou jejich vnitřního registru, protože budou delší i komparace a sčítání.

Porovnání na nejvyšší hodnotu se lépe provádí užitím jiných operací než "rovná se" či "nerovná se", a to především kvůli zvýšení jejich robustnosti, což jsme diskutovali v kapitole 4.7.a na straně 51.[22] Srovnání hodnoty s konstantou má zhruba stejnou složitost u všech typů porovnání.

Děliče frekvence bývají mnohem jednodušší než čítače a nemívají ani asynchronní či synchronní nulování, a tak začneme od nich

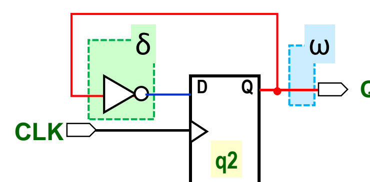
4.8.a @ Příklad VIII. Geneze děliče 10 se symetrickým výstupem

Žádáme-li symetrický výstup se střídou 1:1 (angl. duty cycle 50 %), pak ho u sudých dělitelů nejsnáze vytvoříme, když požadovaného dělitele rozložíme na dělení 2, jemuž bude předřazený jiný dělič.

Dělič 2 s každou aktivní hranou hodin čítá další element z řady: '0', '1', '0', '1', '0'... atd. Následující hodnota jeho funkce δ bude '1' při vnitřním registru v '0' , a '0' při '1', což vede na pouhý invertor.

Jeho kód popisuje schéma vpravo a můžeme napsat třeba následovně:

```
library ieee; use ieee.std_logic_1164.all;
entity DivBy2 is
    port ( CLK : in std_logic; Q : out std_logic);
end entity;
architecture rtl1 of DivBy2 is
begin
    process(CLK)
    variable q2 : std_logic:='0'; --
    begin
      if rising_edge(CLK) then
         q2:=not q2; -- the delta function - the next value
      end if;
      Q<=q2; -- the omega function is only a wire
    end process;
end architecture;
```

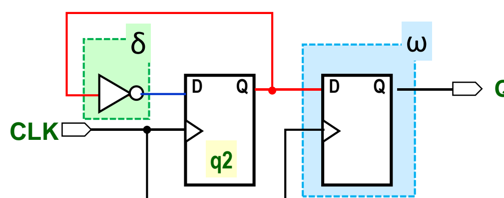


Obrázek 36 Dělič dvěma

Inicializace u definice proměnné q2 je nezbytná kvůli simulaci. Vynechali jsme ACLRN inicializaci, nicméně dělič bude obvodově fungovat, ať se po zapnutí napájení ocitne v jakémkoli stavu. Simulátor by bez ní ale ukazoval pořád 'U', undefined hodnotu std_logic. Neznal by počáteční stav a negace 'U' je 'U'.

Pokud přemístíme generaci výstupu `Q<=q2;` do synchronní sekce přidáme jeden registr navíc, neboť učiníme ω funkci závislou na hodinovém signálu. Každé <= souběžné přiřazení se v ní tak překládá.

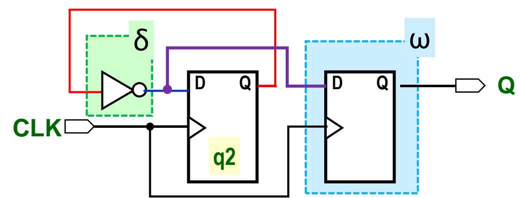
```
architecture rtl2reg of DivBy2 is
begin
   process(CLK)
   variable q2 : std_logic:='0'; --
   begin
     if rising_edge(CLK) then
        Q<=q2;   -- here, omega adds 1 additional register
        q2:=not q2; -- the delta function - the next value
     end if;
   end process;
end architecture;
```



[22] Koncové podmínky cyklu se nedoporučují ve tvarech rovnosti ani v klasickém programování. Pokud se proměnná cyklu omylem zvýší nějakou chybou v našem kódu, tak může přeskočit test konce rovností a pokračovat za povolené hodnoty.

Další kód ukazuje důsledek obráceného pořadí příkazů (napřed q2:=not q2; a po něm Q<=q2;), čím předepíšeme, aby se výsledek funkce δ (následujícího stavu registru q2) posílal do výstupního registru. Do proměnné q2 zapisujeme totiž blocking := příkazem, po němž nabude nové hodnoty.

```
architecture rtl3wrong of DivBy2 is
begin
  process(CLK)
  variable q2 : std_logic:='0'; --
  begin
    if rising_edge(CLK) then
      q2:=not q2; -- the delta function - the next value
      Q<=q2;   -- now, omega uses the next value
    end if;
  end process;
end architecture;
```

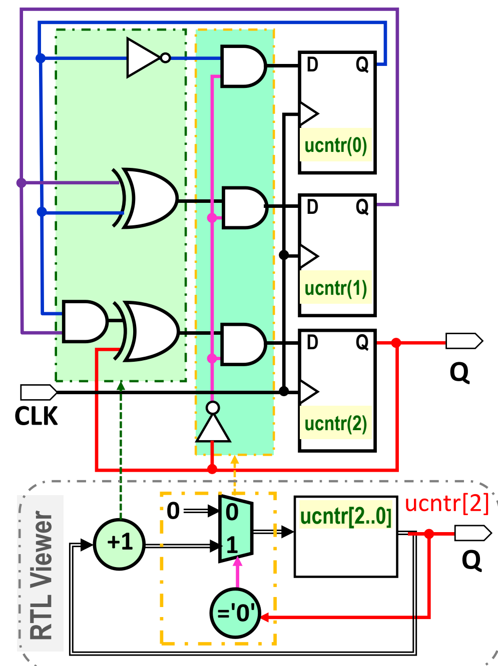


Obrázek 37 - Nevhodné pořadí operací

U složitějších obvodů je lepší **využít napřed hodnotu uloženou registru**, třeba k porovnání či jiné operaci, a až pak ji změnit. Jinak zřetězíme dvě logické operace za sebou. Někdy je to nutné, ale tady vůbec ne. U našeho děliče 2 se složitost samozřejmě nezvýší, vždyť obě funkce jsou primitivní ─ první je pouhou negací a druhá zase spojkou vedoucí na výstup. Tady jen děláme dělič o něco složitější, než je nutné.

Dělič 5 lze navrhnout mnoha způsoby, třeba i s unsigned registrem, jak ukazuje následující kód:

```
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity DivBy5 is  port ( CLK : in std_logic; Q : out std_logic);
end entity;
architecture rtl1unsigned of DivBy5 is
begin
   process(CLK)
   constant ZERO:unsigned(2 downto 0):=(others=>'0');
   variable ucntr : unsigned(ZERO'RANGE):=ZERO;
   begin
      if rising_edge(CLK) then
         if ucntr(ucntr'HIGH)='0'
            then ucntr:=ucntr+1; else ucntr:=ZERO;
         end if;
      end if;
      Q<=ucntr(ucntr'HIGH);
   end process;
end architecture;
```

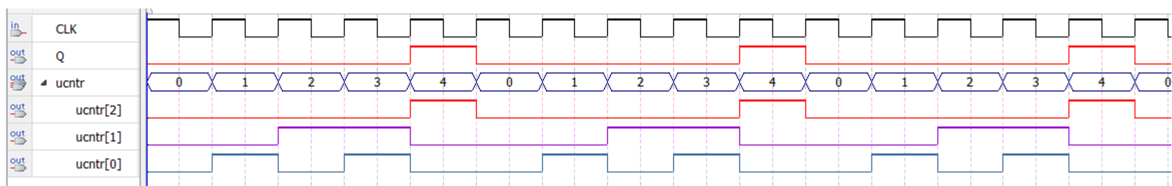


Obrázek 38 - Dělič pěti

Definovali jsme si konstantu *ZERO* a atributy jsme odvozovali veškeré další údaje, čím jsme číslům propůjčili jasný význam na rozdíl od jejich přímého napsání. Zápis `ZERO'RANGE` se nahradil `2 downto 0`, tedy rozsahem v definici *ZERO*, `ucntr'HIGH` nám zase vrací číselně nejvyšší index rozsahu, tedy 2.

VHDL unsigned kód není názorný, neboť příliš napodobuje skutečnou implementaci, kterou ukazuje pravá část obrázku nahoře. V jeho dolní části se naznačuje první krok překladu, schéma v RTL Viewer, v němž se používá úsporná AHDL[23] syntaxe s hranatými indexy. Nad ním uvádíme zapojení z hradel a klopných obvodů. Nejvyšší bit `ucntr` určuje výstupní hodnotu i okamžik vynulování.

[23] AHDL - Altera Hardware Description Language - interní jazyk navržený k úsporným popisům ve schématech Quartusu. Více https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/hdl/ahdl/ahdl_intro.htm



S vnitřním registrem typu integer vede kód na stejní zapojení, který uvedl Obrázek 38.

```
architecture rtl2integer of DivBy5 is
begin
  process(CLK)
  constant DivBy: integer:=5;
  variable cntr : integer range 0 to DivBy-1:=0;
  begin
    if rising_edge(CLK) then
      if cntr<DivBy-1 then cntr:=cntr+1; else cntr:=0; end if;
    end if;
    if cntr<DivBy-1 then Q<='0'; else Q<='1'; end if;
  end process;
end architecture;
```

Nutně musíme uvést rozsah range v definici cntr. Vynecháme-li ho, naroste nám počet použitých logických elementů ze 3 na 42, jelikož jsme neposkytneme překladači dostatek informací o implementaci naší integer proměnné, a tak se vytvoří její 32-bitová reprezentace.

Obě architektury pořád nejsou univerzální. Fungují leda na dělič 5, u něhož se do 0 přejde ihned, jakmile druhý bit dosáhne stavu '1'. Navíc dávají na výstupu Q pouze ojedinělé pulzy.

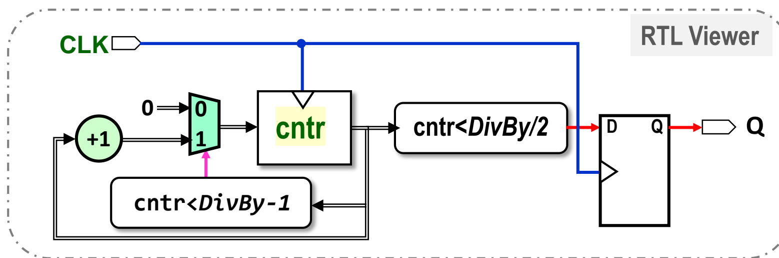
Dosáhnout střídy 1:1 u děliče lichým číslem není snadné [24], ale zlepšit jeho symetrii můžeme následujícím kódem, který již vykazuje rysy univerzálnosti. Není vázaný jen na dělení 5, ale lze ho upravit na jiný poměr pouhým přepsáním hodnoty konstanty *DivBy*.

```
architecture rtl3universal of DivBy5 is
begin
  process(CLK)
  constant DivBy: integer:=5;
  variable cntr : integer range 0 to DivBy-1:=0;
  begin
    if rising_edge(CLK) then
      if cntr<DivBy/2 then Q<='0'; else Q<='1'; end if;        --output
      if cntr<DivBy-1 then cntr:=cntr+1; else cntr:=0; end if;  --increment
    end if;
  end process;
end architecture;
```

Předchozí architektury tvořily výstup Q z nejvyššího bitu registru, který je vždy zcela bez hazardů. Zde se však Q odvozuje podmínkou cntr<*DivBy/2*, která se již realizuje kombinačním obvodem, v němž se v okamžiku změny hodnoty registru cntr (po náběžné hraně CLK) mohou objevit rušivé hazardní zákmity. Přesunuli jsme tedy tvorbu výstupu Q do synchronní sekce, takže výsledek porovnání se do pomocného klopného obvodu nahrává s aktivní hranou hodin, tedy dávno po skončení předešlých hazardů a před jejich novým výskytem. Můžeme pak Q použít i jako hodinový signál rozvedený dalším obvodům.

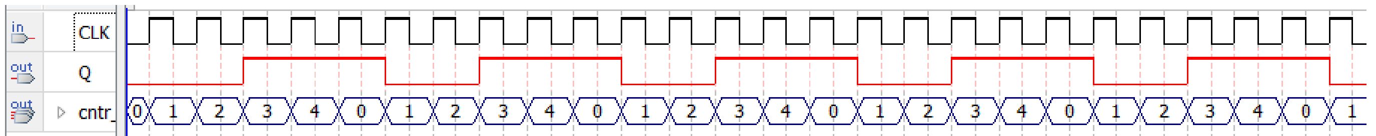
---

[24] Symetrické děliče lichým číslem vyžadují, aby se náběžná hrana výstupu měnila s náběžnou hranou hodin, ale sestupná se sestupnou, což je v rozporu s používanými klopnými obvody, které dovolí změnu jen na jeden typ hrany. Lze je sice vytvořit, ale jedině složitou kombinací několika obvodů. Požadavek se snáze řeší přes obvody PLL (Phase-Locked Loop), které bývají v FPGA. PLL umí vynásobit frekvenci racionálním číslem, ale jeho popis přesahuje rozsah této publikace. Nicméně jeho použití je prosté a rychlé. V době psaní této publikace se však nepodařilo najít vhodný návod. Asi ho bude nutné časem napsat:-)



Připomínáme skutečnost (viz též Obrázek 37 na str. 55), že je lepší podmínku generující výstup Q vložit před načítání proměnné `cntr` čítače. Prohození řádků obou operací, ve VHDL kódu nahoře označených komentáři output a increment, může snížit maximální frekvenci děliče u vyšších hodnot *DivBy*. Komparátor `cntr<DivBy/2` by po prohození porovnával až výsledek sčítačky +1 a ten bude s časovým zpožděním. A nová aktivní hrana hodin smí přijít až po ustálení výsledků obou operací, chceme-li dostat správný výstup. Například zvolíme-li DivBy=1001, pak by se prohozením řádků snížila maximální použitelná frekvence CLK zhruba o třetinu. (U dělení malou hodnotou 5 ji omezí mnohem víc limity klopných obvodů.)

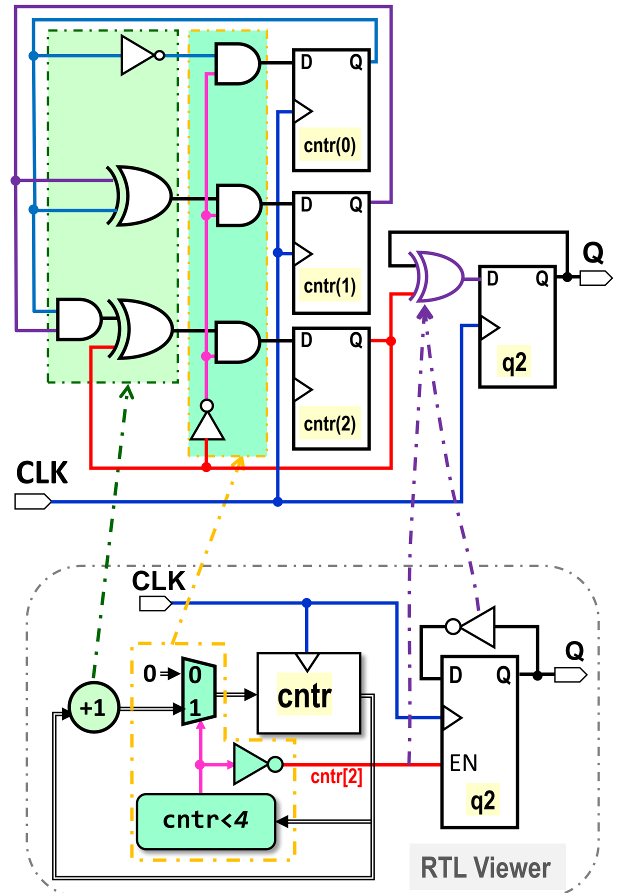
Nahrávání do registru samozřejmě zpožďuje výstup Q oproti vnitřní hodnotě čítače `cntr` o jeden takt hodin, jak ukazuje výsledek simulace.



Dělič 10 dostaneme složením děliče pěti a dvěma. Z děliče pěti nevyvádíme výstup, a tak okopírujeme jeho kód uvedený v architektuře rtl2integer, a do části nulování vložíme děliče dvěma. Ten bude dostávat pořád hodiny, jen se zde povolí jeho překlopení (vstupem EN-enable).

Kód může vypadat následovně:

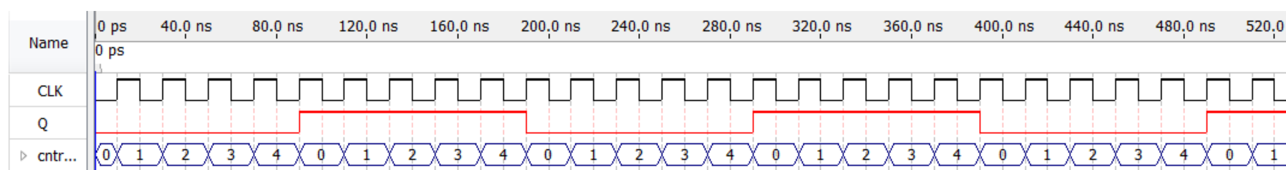
```
library ieee; use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DivBy10 is
     port ( CLK : in std_logic; Q : out std_logic);
end entity;

architecture rtl1 of DivBy10 is
begin

   process(CLK)
   constant DivBy: integer:=5;
   variable cntr : integer range 0 to DivBy-1:=0;
   variable q2 : std_logic:='0';
   begin
      if rising_edge(CLK) then
         if cntr <DivBy-1
            then  cntr:=cntr+1;
            else   cntr:=0;
                   q2:=not q2; -- DivBy2
         end if;
      end if;
      Q<=q2;  -- the omega function is only a wire
   end process;

end architecture;
```



Obrázek 39 Dělič 10

V první fázi překladu se v meta-schématu (RTL Viewer) může ještě objevit blok komparátoru cntr <4, jelikož náš kód se zatím pouze konvertoval na bloky realizovatelné souběžnými příkazy.

V další minimalizaci se podmínka nahradí její negací cntr>=4, neboť ta se v našem kódu rovná nejvyššímu bitu potřebné vnitřní 3-bitové reprezentace integer našeho čísla cntr. Simulace ukáže správný návrh:



I dělič 2 s enable se nakonec může realizovat jinak, třeba pomocí hradla xor bez využití vstupu enable u DFF.

Pokud překladač usoudí, že bude výhodnější ušetřit spojku na obvod q2, pak připojí enable v DFF na '1' a nahradí invertor hradlem xor, na jehož druhý vstup napojí signál, který původně vedl na enable DFF[25], což provádí stejnou operaci. Jde však o různé alternativní možnosti s totožnou výslednou funkcí. Upozorňujeme na ně jenom z hlediska úplnosti, aby nás nezaskočily, až je někde uvidíme.

Přesné využití logických elementů ovlivňují další faktory, například dostupnost interních vodičů v okolí právě užívaného logického elementu, což závisí na ostatních obvodech. Někdy se překladači více šikne vytvořit dělič 2 první variantou (pomocí enable u klopného obvodu DFF a invertoru ve zpětné vazbě), jindy třeba zase aplikuje xor alternativa, tedy řízený invertor ve zpětné vazbě.

[25] Připomínáme, že hradlo A xor B je non-ekvivalence (not A and B) or (A and not B), která se redukuje při B='1' na **not A**, zatímco při B= '0' bude **A**. Pomocí vstupu B tak řídíme, zda se má A negovat či ne. A spotřebuje se vždy celá kombinační funkce v logickém elementu FPGA, ať v ní realizujeme pouhou operaci not či xor.

4.8.b @ Příklad IX. Dělič vysoké frekvence 10 miliony se symetrickým výstupem

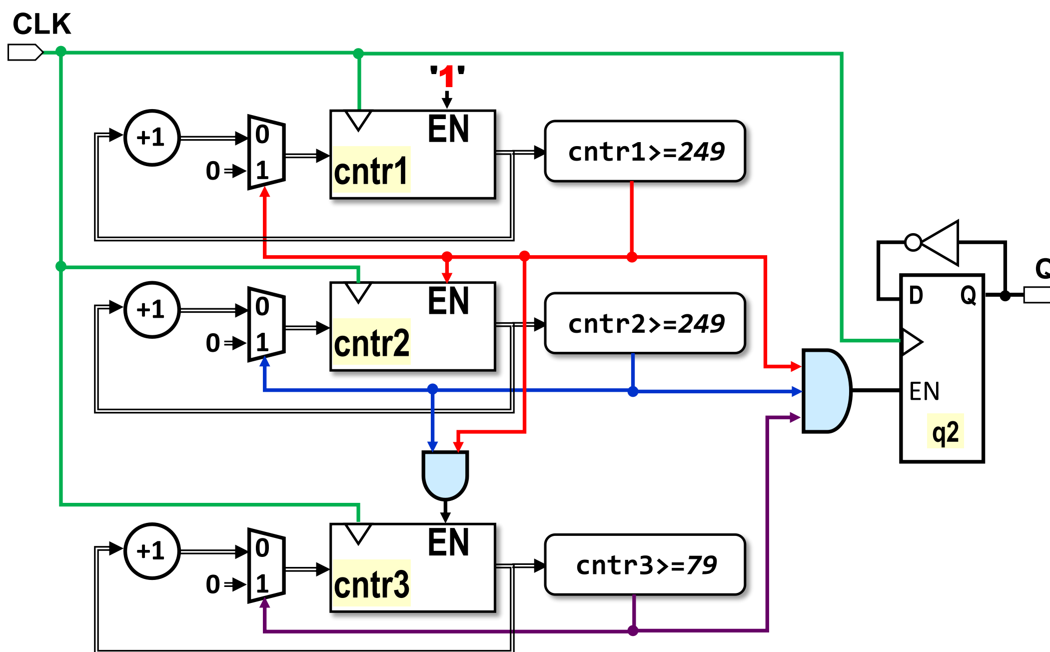
Dělič lze sice realizovat pouhým přepsání konstanty v předchozím kódu,

```
constant DivBy: integer:=5000000;
```

ale měl by nízkou maximální frekvenci.

Vnitřní proměnná cntr zde již vyžaduje 23-bitovou vnitřní reprezentaci, takže sčítačka +1 i komparátor narostou a bude jim trvat delší dobu, než se ustálí jejich výstupy. U děliče tím klesne frekvence, kterou dokáže ještě zpracovat. Zrychlíme ji, pokud zkrátíme vnitřní registry.

Dělení 10 000 000 rozložíme napřed na 5 000 000*2, abychom na výstupu měli dělič 2, který nám zaručí symetrii. Pokračujeme v rozkladu a zvolíme taková čísla, která se dají realizovat 8-bitovými registry, neboť víme, že naše logické elementy dovedou 4-vstupové kombinační funkce. Napíšeme dělení jako 250*250*80*2 = 10 000 000.



Obrázek 40 Blokové schéma děliče 10 miliony

Blokovou strukturu ukazuje obrázek nahoře. Využíváme v ní vstupy EN (enable) klopných obvodů DFF. Všechny vnitřní čítače tak dostávají trvale hodinový signál, pouze určujeme, zda se má na něj reagovat či ne.

První čítač cntr1 má enable trvale v '1', takže čítá vždy. Pokud dosáhne čísla 249, tak jeho komparátor přepne vstupní multiplexor, aby po následující hraně hodin se nahraje '0', čímž se začne od začátku. Výstup jeho komparátoru se pošle na enable vstup dalšího čítače. Ten pouze v tomto okamžiku načte +1, takže lze tedy říct, že cntr2 počítá, kolikrát se cntr1 vynuloval. Poslední čítač cntr3 se zase inkrementuje až tehdy, když se nulují oba horní čítače. Výstupní klopný obvod q2 se překlopí, když se nulují všechny tři čítače naráz, což nastane po 250*250*80=5 milionů, takže na Q bude CLK vydělený10 miliony.

Všimněte si, že všechny čítače i komparátory pracují paralelně a nečekají na své výsledky. Hradla and přidají jen zanedbatelné zdržení.

Kaskáda děličů má stejnou složitost jeho obří dělič vzniklý z DivBy10 pouhým přepsáním jeho konstanty constant DivBy: integer:=5000000; Ve skutečnosti potřebuje dokonce o jeden LE méně, což ale znamená bezvýznamnou úsporu. Mnohem důležitější je fakt, že zpracuje skoro o 50 procent vyšší frekvenci na svém vstupu CLK.

Lze si položit otázku, zda by se nedal zefektivnit ještě jemnějším rozkladem. Pokud něco takového budeme potřebovat, musíme experimentovat a vyzkoušet různá dělení. Nárůst nám ale zabrzdí rychlost použitých klopných obvodů. Na FPGA řady Cyclone II by naopak hrubší rozklad na 2000*2500*2 nepřinesl větší urychlení oproti přímému dělení 5 miliony - ověřeno:-)

Kód si napřed ukážeme ve výukové verzi, v níž přímým přepisem obrázku uvidíme jednotlivé operace.

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity DivBy10M is
   port ( CLK : in std_logic;
          Q : out std_logic);
end entity;
architecture rtlEdu of DivBy10M is
begin
   process(CLK)
   constant DivBy1: integer:=250;
   constant DivBy2: integer:=250;
   constant DivBy3: integer:=80;
   variable cntr1 : integer range 0 to DivBy1-1:=0;
   variable cntr2 : integer range 0 to DivBy2-1:=0;
   variable cntr3 : integer range 0 to DivBy3-1:=0;
   variable q2 : std_logic:='0';
   variable clear1, clear2, clear3 : boolean;
   begin
      if rising_edge(CLK) then
         clear1:=cntr1>=DivBy1-1; clear2:=cntr2>=DivBy2-1; clear3:=cntr3>=DivBy3-1;
         if TRUE then  -- cntr1 is always enabled
            if clear1 then cntr1:=0; else cntr1:=cntr1+1; end if;
         end if;
         if clear1 then  -- cntr2 is enabled
            if clear2 then cntr2:=0; else cntr2:=cntr2+1; end if;
         end if;
         if clear1 and clear2 then  -- cntr3 is enabled
            if clear3 then cntr3:=0; else cntr3:=cntr3+1; end if;
         end if;
         if clear1 and clear2 and clear3 then  -- div2 is enabled
            q2:=not q2;
         end if;
      end if;
      Q<=q2;
   end process;
end architecture;
```

Ve výukovém VHDL kódu se zavedly pomocné proměnné respektující strukturu obrázku, které se jinak nechávají na překladači. Úsporný kód, viz dále, použije vnořené příkazy if-then-else , které zaručí slučování podmínek, což na obrázku prováděla hradla AND. Zde se nehodí příkazy if-then elsif, protože tady nejde o prioritní kaskádu, v níž se vybírá pouze jedna operace. Budeme občas načítat i více čítačů najednou.

V úsporném kódu použijeme i obrácené podmínky, aby se další if příkazy vkládaly až do sekcí else, tedy na konce předchozích if příkazů, což povede na přehlednější kód.

Úvodní definice architektury se shodují, ale nevytváříme již pomocné proměnné clearX.

```
architecture rtlShort of DivBy10M is
begin
   process(CLK)
   constant DivBy1: integer:=250;
   constant DivBy2: integer:=250;
   constant DivBy3: integer:=80;
   variable cntr1 : integer range 0 to DivBy1-1:=0;
   variable cntr2 : integer range 0 to DivBy2-1:=0;
   variable cntr3 : integer range 0 to DivBy3-1:=0;
   variable q2 : std_logic:='0';
   begin
      if rising_edge(CLK) then
         if cntr1<DivBy1-1 then cntr1:=cntr1+1;
         else cntr1:=0;
               if cntr2<DivBy2-1 then cntr2:=cntr2+1;
               else cntr2:=0;
                     if cntr3<DivBy3-1 then cntr3:=cntr3+1;
                     else cntr3:=0;
                           q2:=not q2;
                     end if;
               end if;
         end if;
      end if;
      Q<=q2;
      end process;
  end architecture;
```

4.8.c ***Cvičná úloha 8 - Generátor pulzně šířkové modulace

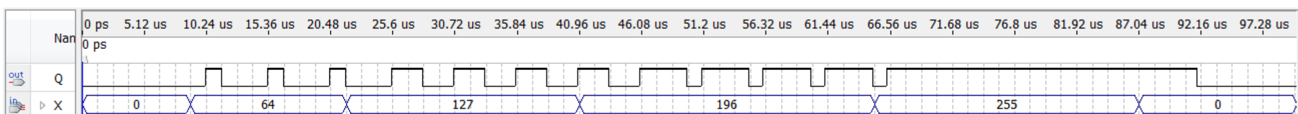
Zadání: Ze vstupní frekvence 50 MHz vytvořte generátor pulzně-šířkové modulace (PWM)[26], který bude dávat výstupní frekvenci kolem jednoho 1 kHz s střídou řízenou 8-bitovým vstupem X.

```
entity PWM8 is
port (    CLOCK_50 : in std_logic;
          X:in std_logic_vector(7 downto 0);
          Q : out std_logic);
end entity;
```

Při X=0 bude Q trvale '0', při X=127 se dosáhne zhruba střídy 1:1 a při X=255 bude na výstupu trvale '1'.

Návod: PWM vytvoříte šikovnou úpravou děliče uvedeného nahoře, u něhož stačí jen dva vnořené děliče, pokud zvolíte vhodný rozklad na dělitele. Kvůli němu se nevyžaduje přesných 1 kHz, protože poslední dělitel musí mít přesnou hodnotu. Jakou? Zamyslete se na požadavkem '0' při X=0 a '1' při X´255.

Dole vidíte záznam simulaci, která se vytvořila po zablokování předdělení, takže se CLOCK_50 pouštělo přímo na závěrečný stupeň.



[26] Úvod do PWM najdete třeba na https://en.wikipedia.org/wiki/Pulse-width_modulation

4.8.d Univerzální čítač

Univerzální čítač můžeme vytvořit takto:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity CounterUp is
  generic ( BITLEN : integer := 8; MAXVAL : integer:=250 );
  port (  CLK, ACLRN, RESET : in std_logic;
          Q : out std_logic_vector(BITLEN-1 downto 0));
   begin  assert BITLEN>0 and MAXVAL<2**BITLEN
          report "Required BITLEN>0 and MAXVAL<2**BITLEN" severity failure;
end entity;
architecture rtl of CounterUp is
begin
  process(CLK, ACLRN)
  variable cntr : integer range 0 to MAXVAL:=0;
  begin
    if ACLRN='0' then cntr:=0;
    elsif rising_edge(CLK) then
      if RESET='1' or cntr>=MAXVAL then cntr:=0; else cntr:=cntr+1;
      end if;
    end if;
    Q<=std_logic_vector(to_unsigned(cntr, Q'LENGTH));
  end process;
end architecture;
```

Všimněte si definice variable cntr : integer range 0 to *MAXVAL*:=0; kde jsme jasně Quartusu specifikovali rozsah použitého skaláru cntr. Vynechání range by v případě 8 bitového čítače zvýšilo počet použitých LE (logických elementů) zhruba čtyřikrát (z původních 11 na 43), protože by se proměnná cntr realizovala jako 32 bitový registr.

Funkce δ následující hodnoty if RESET='*1*' or cntr>=*MAXVAL* then cntr:=0; else cntr:=cntr+1; end if;
uvnitř synchronní sekce buď nuluje, je-li RESET v '1', nebo po dosažení maximální hodnoty.

Výstupní funkce ω Q<=std_logic_vector(to_unsigned(cntr, Q'*LENGTH*)); pak specifikuje propojení na výstup.

Otázka: Šlo by zadávat jen MAXVAL? Předchozí čítač vyžaduje BITLEN (bitovou délky výstupu) a hodnotu MAXVAL (maximálního čísla před vynulováním), ale z té lze BITLEN vypočítat.

Ano, lze, ale vznikne obvod s předem nejasnou délkou Q, což není praktické. Jeho architektura by zůstala stejná jako v předchozím kódu, ale v entitě by provedly zvýrazněné změny:

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
use ieee.math_real.all;
entity CounterUp1 is
  generic ( MAXVAL : integer:=256 );
  port  ( CLK, ACLRN, RESET : in std_logic;
        Q : out std_logic_vector(integer(ceil(log2(real(MAXVAL+1))))-1 downto 0));
  begin assert MAXVAL>1 report "Required MAXVAL>1" severity failure;
end entity;
```

Přidala se reálná knihovna, avšak se používá leda k výpočtu hodnoty BITLEN v době překladu, jinde ne, takže obvod jde syntetizovat. Stanovený počet bitů je pak dostupný v atributu Q'LENGTH.

Samotný výpočet `integer(ceil(log2(real(MAXVAL+1))))` znamená: maximální požadovaná hodnota na čítači zvýšená o 1, MAXVAL+1, se převede na reálné číslo, z něhož se následně vypočte logaritmus o základu 2, který se zaokrouhlí na strop, tedy nahoru. Ověříme správnost na několika hodnotách. Je-li

MAXVAL=255, výsledek bude 8, při MAXVAL=256 dostaneme 9. Z MAXVAL=1023 získáme 10, z MAXVAL=1024 pak 11. MAXVAL=3 dá 2, MAXVAL=4 zas 3. Vzorec tedy počítá správně.

Obvodu nám sice iniciativně vypočte bitovou délku vektoru Q, ale my si ji musíme stanovit také, protože na Q potřebujeme připojit vektor shodné délky, takže jsme si práci neušetřili:-) Ale aspoň jsme ukázali řešení, který se možná hodí někomu někde jinde, třeba jako inspirace.

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

entity DivByEven is
   generic ( EVEN_DIV : integer:=100 );
   port ( CLK : in std_logic; Q : out std_logic);
   begin
      assert EVEN_DIV mod 2=0 report "EVEN_DIV divisor is not EVEN number" severity error;
end entity;

architecture rtl of DivByEven is
begin
   process(CLK, ACLRN)
   constant DIVISOR:integer := EVEN_DIV/2;
   variable cntr : integer range 0 to DIVISOR-1:=0;
   variable   q2 : std_logic := '0';
   begin
      if rising_edge(CLK) then
         if DIVISOR>1 and cntr<DIVISOR-1 then cntr:=cntr+1;
         else cntr:=0; q2:=not q2;
         end if;
      end if;
      Q<=q2;
   end process;
end architecture;


- divider_generic of frequency--
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;

entity divider_generic is
   generic( DIVISOR: integer := 10 );
   port(CLK   : in std_logic;
         q   : out std_logic);
   begin          assert DIVISOR>1
            report "TOO SMALL DIVISOR: divider_generic requires DIVISOR of frequency greather than 1."
            severity severity_level(error);
end entity;
```

```vhdl
architecture behav of divider_generic is

-- We set IRANGE to (DIVISOR/2-1) for even (cz:sude) number and to (DIVISOR-1) for odd (cz:liche)
constant IRANGE:integer := (DIVISOR-1)*(DIVISOR mod 2)+(DIVISOR/2-1)*((DIVISOR+1)
mod 2);
begin

  process (CLK)
  variable   cnt : integer range 0 to IRANGE:=0;
  variable   q2 : std_logic := '0';
  begin
  if rising_edge(CLK) then
  -- if-elsif depends on constants, so compiler will implement only one part
    if DIVISOR=2  -- dividing by 2 is a special simple case
          then q2:=not q2;
       elsif DIVISOR mod 2 = 0 then
      -- To divide by EVEN number, we divide first by DIVISOR/2 and then by 2
      -- by this way, the complexity of circuit will be reduced approx. by 20 %
             if cnt>=IRANGE then cnt:=0; q2:=not q2;
             else cnt := cnt + 1;
             end if;
       else  -- dividing by ODD (cz:liche) number ------------------------------
             if cnt>=DIVISOR/2 then q2:='1';
             else q2:='0';
             end if;
             if cnt>=IRANGE then cnt:=0;
             else cnt:=cnt+1;
             end if;
       end if;
  end if;

  q<=q2;  -- copy internal variable of process to output

  end process;
end behav;
```

5 Konečné automaty alias FSM (Finite State Machines)

-- zatím nedopsáno --

6 Příloha A: Řešení cvičných úloh

6.1 Cvičná úloha 1: Lineární ukazatel

Jedno z možných řešení může vypadat následovně:

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

entity unsigned2barG is
    generic(HBIT_A:integer :=3;
            HBIT_BAR:integer :=9);
   port (A: in std_logic_vector(HBIT_A downto 0);
         BAR: out std_logic_vector(HBIT_BAR downto 0));
   begin assert HBIT_A>1 and HBIT_BAR>1
         report "Required HBIT_A>1 and HBIT_BAR>1"
         severity failure;
end entity;

architecture beh of unsigned2barG is
begin
    process(A)
   constant ONES : std_logic_vector(BAR'RANGE):=(others=>'1');
   variable result: std_logic_vector(BAR'RANGE);
   variable maxbit:integer range 0 to 2**HBIT_A;
   begin
     result:=(others=>'0'); --correct initializations inside begin end process
ilp: for i in 0 to A'HIGH loop
          if A(i)='1' then
          maxbit := 2**i; --OK, the rightside is a constant known in the compile time
          if maxbit>=result'HIGH then result:=ONES;
          else
             -- left shift '1' corresponding to value A(i)
            result:= result(result'HIGH-maxbit downto 0) & ONES(maxbit-1 downto 0);
          end if;
        end if;
     end loop;
     BAR<=result;
   end process;
end architecture;
```

6.2 Cvičná úloha 2: Porovnání čísel v Grayově kódu

Možné řešení může vypadat takto:

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

entity GrayCodeGreaterThan is
    generic(HBITX:integer :=3; HBITY:integer:=4);
   port (X: in std_logic_vector(HBITX downto 0);
         Y: in std_logic_vector(HBITY downto 0);
         GreaterThan: out std_logic -- X greater then Y
        );
   begin assert HBITX>1 and HBITY>1 report "Required HBITX>1 and HBITY>1"
         severity failure;
end entity;

-----
architecture beh of GrayCodeGreaterThan is

     function Gray2Int (g:std_logic_vector) return integer is
     variable result:std_logic_vector(G'RANGE);
     begin
       result(g'HIGH) := g(g'HIGH);
       for i in G'HIGH-1 downto 0 loop
         result(i) := result(i+1) xor g(i);
       end loop;
       return to_integer(unsigned(result));
     end;

begin
   GreaterThan<='1' when Gray2Int(X)>Gray2Int(Y) else '0';
end architecture;
```

6.3 Cvičná úloha 3: MaxMinSwap pomocí procedure

Vysledné řešení může vypadat takto:

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

entity MaxMinSwap  is
   generic(HBIT:integer:=3);
    port
    (  X1, X2 : in std_logic_vector(HBIT downto 0);
       XMAX, XMIN : out std_logic_vector(HBIT downto 0)
    );
    begin
      assert HBIT>=0 report "Required HBIT>=0" severity failure;
end entity;

architecture rtl of MaxMinSwap is

  procedure MMSwap(A, B: in std_logic_vector; signal MAX, MIN : out std_logic_vector) is
  begin
    if unsigned(A)>=unsigned(B) then MAX<=A; MIN<=B;
    else MAX<=B; MIN<=A;
    end if;
  end procedure;

begin
      MMSwap(X1,X2,XMAX,XMIN);
end architecture;
```

Vytvoříme-li si balíček, třeba v souboru my_logic.vhd

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
package my_logic is
    procedure MMSwap(A, B: in std_logic_vector; signal MAX, MIN: out std_logic_vector);
end package;

package body my_logic is
  procedure MMSwap(A, B: in std_logic_vector; signal MAX, MIN : out std_logic_vector) is
  begin
    if unsigned(A)>=unsigned(B) then MAX<=A; MIN<=B;
    else MAX<=B; MIN<=A;
    end if;
end procedure;
end my_logic;
```

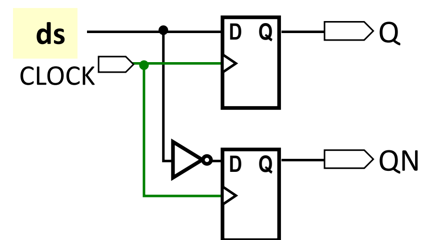
pak se původní architektura rtl výrazně zjednoduší na:

```
architecture rtl2 of MaxMinSwap is
use work.my_logic.all;
begin
      MMSwap(X1,X2,XMAX,XMIN);
end architecture;
```

6.4 Cvičná úloha 4: Registr s xor Data

Kvůli lepší orientaci zopakujeme celý kód zadaného příkladu i s entitou

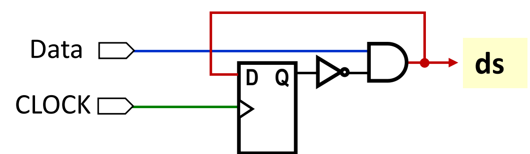
```
library ieee; use ieee.std_logic_1164.all;
entity DataFlipFlop is
   port (  Data, CLOCK : in std_logic; Q, QN : out std_logic);
end entity;

architecture rtl3_xor of DffQN is
begin
   process(CLOCK)
   variable ds:std_logic:='0'; -- power-up initialization only
   begin
      if rising_edge(CLOCK) then
         ds:=Data and not ds;
         Q<=ds; QN<=not ds;
      end if;
   end process;
end architecture;
```



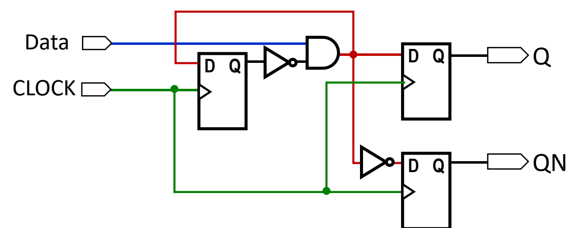
Obě závěrečná přiřazení

```
Q<=ds; QN<=not ds;
```

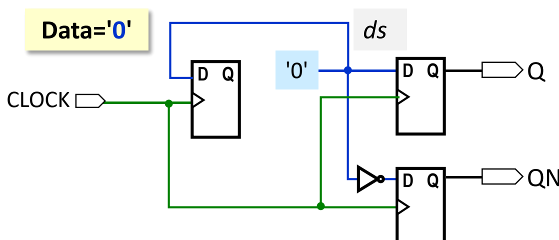
uvnitř if podmínky detekce náběžné hrany jsme si vysvětlovali. Popisují dvojici DFF registrů.

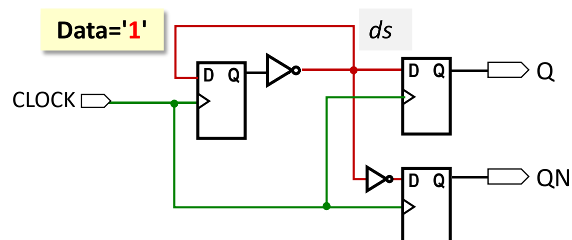


První příkaz ds:=Data and not ds; specifikuje vložení registru typu DFF, jehož vstup D dostává logický součin vstupu Data s negovanou hodnotou výstupu Q, v němž se pamatuje hodnota ds při předchozí náběžné hraně.

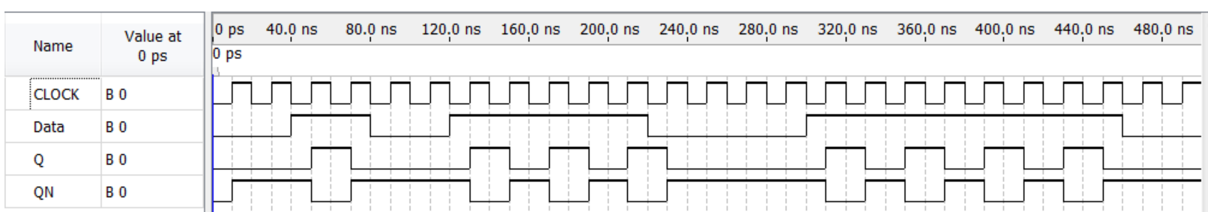


Spojíme-li vše dohromady, dostaneme výsledné schéma. Dosadíme-li do něj hodnoty Data jako '0' a '1' (použijeme tedy Shannonovu expanzi), obdržíme schémata, z nichž již snáze odvodíme odezvu obvodu na zadané signály:





Při Data='0' se po nejbližší náběžné hraně překlopí DFF (paměť ds) do '0' a spolu s tím i výstupy na Q='0' a QN='1'. Jakmile Data přejdou do '1', s každou náběžnou hranou se všechny DFF budou překlápět na opačné hodnoty. Domněnku si nakonec ověříme simulací.



6.5 Cvičná úloha 5: Grayův čítač

Jde snad o nejlehčí příklad zadaný k řešení. Stačí drobné modifikace kódu uvedeného na Obrázek 22 ze stránky 43.

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

entity GrayCntr is
   generic(MAXBIT:integer:=4);
      port (CLOCK, X_reset, ACLRN: in std_logic;
            Y: out std_logic_vector(MAXBIT downto 0));
      begin
         assert MAXBIT>=2 report "Required MAXBIT>=2" severity failure;
end entity;


architecture rtl of GrayCntr is
begin
   process(CLOCK, ACLRN)
      constant ZERO:unsigned(Y'RANGE):=(others=>'0');
      variable rg:unsigned(Y'RANGE):=ZERO;
      begin
      if ACLRN='0' then rg:=ZERO;
            elsif rising_edge(CLOCK) then
              if X_reset='1' then rg:=ZERO;
              else rg:=rg+1;
              end if;
         end if;
            Y(MAXBIT)<=std_logic(rg(MAXBIT));
      iloop:for i in MAXBIT-1 downto 0 loop
               Y(i)<=std_logic(rg(i+1) xor rg(i));
            end loop;
      end process;
end architecture;
```

Konstrukce ve smyčce iloop posílá do Y negaci bitu, je-li vyšší bit v 1, tedy vytváří tzv. Zrcadlový binární kód, probraný v první části učebnice, který ukazuje i obrázek dole na případu 3 bitů.

N
0
1
2
3
4
5
6
7

bin2	bin1	bin0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

gray2	gray1	gray0
0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

Obrázek 41 - 3bitový binární zrcadlový Grayův kód

6.6 Cvičná úloha 6: Posuvný obecné délky s nulování a nastavením

Úloha se dá vyřešit jednoduchou modifikací kódu uvedeného v řešení příkladu. V synchronní sekci jde o prioritní úlohu, kde nulování vstup RESET má přednost, po něm následuje SLOAD a nakonec RightDir.

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

entity ShiftBidir is
  generic(MAXBIT:integer:=7);
  port(CLOCK, ACLRN : in std_logic;                      -- clock and asynchronous clear
       Reset : in std_logic;                              -- if Reset='1' then clear shift register
       SLoad : in std_logic;          -- if SLoad='1' and Reset='0 then load value from Data input
       Data: in std_logic_vector(MAXBIT downto 0);       --data input for SLoad
       RightDir : in std_logic;       -- if SLoad='0' and Reset='0 then on RightDir='1' right shift, else left shift
       LeftIn, RightIn : in std_logic;                   -- serial-in for left and right shifts
       Q: out std_logic_vector(MAXBIT downto 0));        -- output
  begin
    assert MAXBIT>1 report "Required MAXBIT>1" severity failure;
end entity;

architecture rtl of ShiftBidir is
begin
  process(CLOCK)
  constant ZERO: std_logic_vector(Q'RANGE) := (others=>'0');
  variable rg:std_logic_vector(Q'RANGE):=ZERO; -- initialization for a simulation only
  begin
    if ACLRN='0' then rg:=ZERO;
    elsif rising_edge(CLOCK) then
      if Reset='1' then rg:=ZERO;
      elsif Sload='1' then rg:=Data;
      elsif RightDir='1' then
        rg:= RightIn & rg(MAXBIT downto 1);
      else
        rg:= rg(MAXBIT-1 downto 0) & LeftIn;
      end if;
    end if;
    Q<=rg;
  end process;
end architecture;
```

6.7 Cvičná úloha 7: Variabilní dělič

Jedno z možných řešení ukazuje kód dole.

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

entity JohnsonDivSwitch is
  port ( CLOCK : in std_logic;   -- clock
         IXDIV: in std_logic_vector(1 downto 0);
         Q: out std_logic);      -- output
end entity;

architecture rtl of JohnsonDivSwitch is
begin
  process(CLOCK)
  variable rg:std_logic_vector(5 downto 0);
  variable  x:std_logic;
  begin

    if rising_edge(CLOCK) then
      case IXDIV is
        when "00"=> x:=rg(3);
        when "01"=> x:=rg(2);
        when "10"=> x:=rg(1);
        when others=> x:=rg(0);
      end case;
      rg:= not x & rg(rg'HIGH downto 1);
    end if;

    Q<=rg(3);
  end process;
end architecture;
```

Pomocnou proměnnou x jsme použili ke zjednodušení zápisu. Jelikož vždy definujeme její novou hodnotu, není potřeba si ji zapamatovat předchozí, takže se nám nepřidá další registr.

Opět zdůrazníme nutné použití others u poslední volby case. ModelSim by nám jinak hlásil chybu, protože ve skutečnosti IXDIV vstup nabývá 81 hodnot, jelikož jeho členy popisuje 9-hodnotová std_logic.

Položená otázka: *Může se při změně dělicího poměru objevit někdy pulz, ať v '0' či v '1', který bude kratší, než by odpovídalo průběhu při hodnotě IXDIV, která je nižší z obou měněných?*

Předpokládejme, že na rg(5 downto 0) máme hodnotu "111000" a dělíme 12, čili IXDIV="11". Zpětná vazba je z not rg(0). Přepneme-li IXDIV na "00", pak další s další hranou hodin bude rg "011100", jelikož vazbu vedeme teď z not rg(3)='0'. Změníme-li vzápětí IXDIV="11", pak rg po hraně hodin bude "101110", čili máme kratší pulz.

Lze tohle korigovat? Ano, ale ne jednoduše. Bylo by třeba IXDIV předpracovat dalším procesem na pomocný signál, který by testoval, zda změna je přípustná.

7 Seznam číslovaných obrázků a tabulek

Mnohé obrázky se nečíslovaly, pokud jen doplňovaly jinou kresbu.

Obrázek 1 - Modelování prioritního inhibitoru kaskádou dvouvstupových multiplexorů 8
Obrázek 2 - Loop a Latch ve zprávách o překladu 9
Obrázek 3 - Naznačení překladu přepisovaných signálů 10
Obrázek 4 - Modelování inhibitoru multiplexorem 11
Obrázek 5 - Překlad unárního XOR 22
Obrázek 6 - Výsledek opomenutí inicializace result:='0'; v kódu procesu 22
Obrázek 7 - Process se sensitivity list a jeho simulace 23
Obrázek 8 - Převod z Grayova RBC na binární číslo 25
Obrázek 9 - Zpoždění na RC článku 29
Obrázek 10 - Zprávy o vytvoření nežádoucího Latch 33
Obrázek 11 - Funkce rising_edge() a falling_edge() 34
Obrázek 12 - Důsledek nevhodné asynchronní inicializace DFF 37
Obrázek 13 - Finální synchronní klopný obvod s dvojím nulováním 37
Obrázek 14 - Ukázka reakcí synchronního klopného obvodu s dvojím nulováním 37
Obrázek 15 - Synchronní klopný obvod s dvojím nulováním reagující na sestupnou hranu 38
Obrázek 16 - Dva DFF s Q a QN 38
Obrázek 17 - DFF s Q a QN varianta nadbytečná - 39
Obrázek 18 - Optimální DFF s Q a QN 39
Obrázek 19 - Synchronní obvod se strukturou FSM (konečného automatu) 41
Obrázek 20 - Logické schéma Grayova čítače s překódováním výstupu 42
Obrázek 21 - Časová simulace Grayova čítače 42
Obrázek 22 - VHDL kód s funkcemi δ a ω 43
Obrázek 23 - Skluz (slack) v distribuci hodin 44
Obrázek 24 - 4-bitový posuvný registr 45
Obrázek 25 - 4-bitový posuvný registr 45
Obrázek 26 - Operátory & v posuvném registru 45
Obrázek 27 - 4-bitový převodník paralelní z paralelního přenosu na sériový 47
Obrázek 28 - Princip sériové komunikace při 4-bitovém přenosu 48
Obrázek 29 - Kruhový čítač 48
Obrázek 30 - Johnsonův dělič frekvence - 4, 6 a 8 49
Obrázek 31 - Výstupy Johnsonova 5-bitového čítače 49
Obrázek 32 - Navigační světlo 50
Obrázek 33 - Navigační světlo za předpokladu užití DFF jen s asynchronním nulováním 50
Obrázek 34 - Navigační světlo s binárním čítačem 52

Obrázek 35 - Navigační světlo s Johnsonovým čítačem ... 52
Obrázek 36 Dělič dvěma ... 54
Obrázek 37 - Nevhodné pořadí operací ... 55
Obrázek 38 - Dělič pěti ... 55
Obrázek 39 Dělič 10 ... 57
Obrázek 40 Blokové schéma děliče 10 miliony ... 59
Obrázek 41 - 3bitový binární zrcadlový Grayův kód ... 70

7.1 Seznam tabulek

Tabulka 1 - Pravdivostní tabulka 3-bitového prioritního inhibitoru ... 7
Tabulka 2 - Deklarace v process, function a procedure ... 21
Tabulka 3 - Jednotky času ve VHDL ... 29
Tabulka 4 - Funkce Grayova čítače s překódováním výstupu ... 42

8 Závěr

Nejde o finální verzi, ale rozpracovaný dokument!

Jde o pracovní text nového výukového materiálu, který začal vznika během roku 2021 paralelně s dalšími očebnicemi. Jedná se 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 2021: ČVUT Fakulta elektrotechnická,

Katedra řídicí techniky

Technická 2, Praha 6

8.1 Historie verzí dokumentu

Verze beta 0.1 - 11. červen 2021

Verze beta 0.2 - 28. červenec 2021

- provedené korekce překlepů a přidané části, které se později přesunyly do učebnici Logické obvody, neboť začal příliš narůstat rozsah.

Verze beta 0.3 - 15. srpen 2021

- dopsaný začátek kapitoly 4

Verze beta 0.5 - breřen 2023

- text začleněný do pětice učebnic