Podmínky v jazyce VHDL, konverze datových typů, ukázky realizace

With/select, when/else, if, case, detekce vzestupné a sestupné hrany signálu, simulace ve VHDL, konverze typů

Ing. Pavel Lafata, Ph.D.
lafatpav@fel.cvut.cz

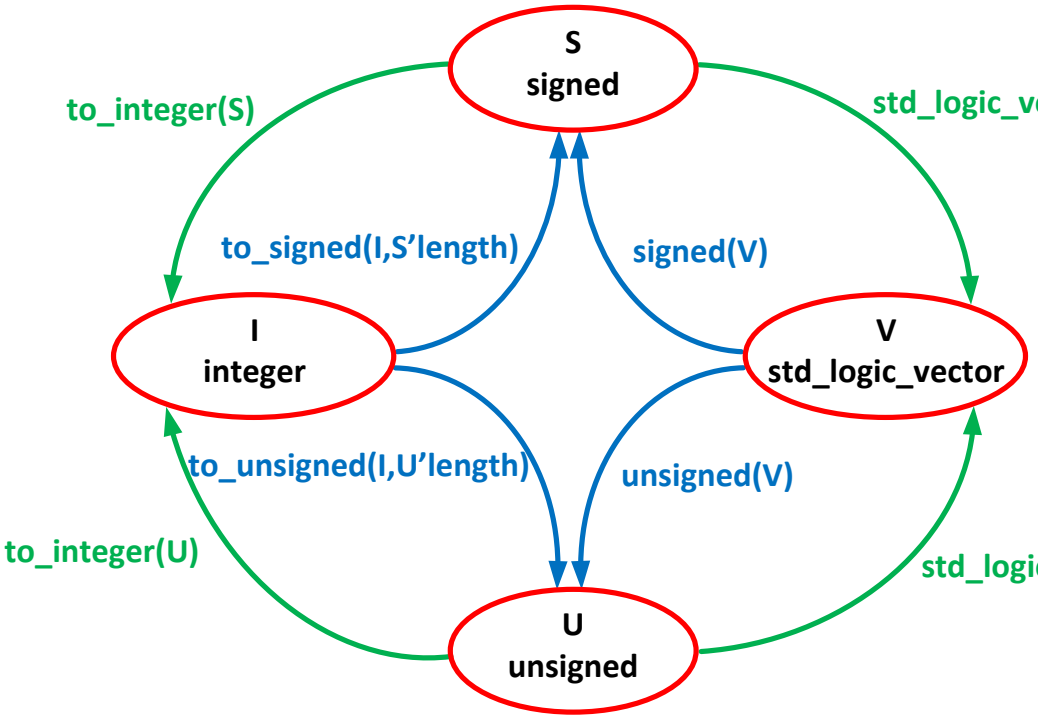
- **syntetizovatelný kód v jazyce VHDL**
  - jazyk VHDL slouží pro syntézu (a implementaci) logických obvodů a také pro jejich simulace a modelování
  - proces směřující k implementaci obvodu do FPGA pole:
  1. **parsování** – kontrola syntaxe VHDL kódu
  2. **syntéza** – několik kroků vedoucí k vygenerování netlistu
     - kompilace – převod forem disjunktní, konjunktní, smíšená apod.
     - optimalizace, selekce – analýza námi vytvořeného kódu a hledání typizovaných struktur (např. klopných obvodů, násobiček, multiplexorů…)
     - optimalizace – z hlediska cílového HW, časová optimalizace, sharing (sdílení HW), atd.
  3. **implementace** – opět několik kroků
     - mapování – přiřazení obvodových prvků vytvořeným při syntéze reálným prvkům dostupným v daném programovatelném poli
     - rozmístění (placement) a propojování (routing) bloků obvodu
     - vyhodnocení časových podmínek
     - výstupem je soubor pro „naprogramování“ pole
  - v různých fázích můžeme ovlivňovat implementaci – standardní parametry (optimalizace s důrazem na zpoždění, na počet obsazených prvků pole, na plochu obvodu…), specifické parametry (např. výběr kódování stavů při návrhu automatu…)
  - v jazyce VHDL můžeme rovněž provádět **simulace** – **v různých fázích**, viz dále

VHDL – Základní kroky a doporučení pro syntézu obvodů

- **syntetizovatelný kód v jazyce VHDL**
  - musíme vytvořit tzv. **syntetizovatelný kód VHDL**
  - typické problémy při návrhu:

1. **ne všechny VHDL příkazy a konstrukce lze syntetizovat**
   - některé VHDL příkazy nemohou být syntetizovatelné, určené jen pro simulace, např. wait for čas, detekce dvou různých hran v jednom procesu, některé smyčky, atd.
   - pozor při behaviorálním popisu – i malá změna kódu, velké změny implementace
   - musíme vzít v potaz cílový HW pro implementaci – různá omezení a vlastnosti
2. **práce s hodinovým signálem, synchronní obvody, hradlování hodin**
   - ve VHDL navrhujeme drtivou většinu obvodů jako synchronní (asynchronní jen pro specifické aplikace) – často řešíme problém s časováním
   - nutno zohlednit reálná zpoždění, skluzy a jitter v FPGA poli
   - vždy se snažíme vyhnout tzv. hradlování hodin (gated clock) – v cestě hodinového signálu se nachází kombinační logika (hradla), doporučené metody jak obejít
3. **vyhnout se latchům (hladinovým obvodům)**
   - latche můžeme omylem vytvořit při neúplném uzavření všech podmínek, neomezením rozsahů typů, nezakončením smyček… – zvyšují zpoždění, počet hradel, spotřebu…

- **Datové typy v jazyce VHDL – konverze**
  - jazyk VHDL je silně typově orientovaný – je nutné dbát použití daných datových typů
  - v obvodech, kde se využívají aritmetické operace (čítače, sčítačky…), se často hodí chápat logický vektor jako binárně zapsané číslo
  - standardizovaná knihovna **IEEE.numeric_std**, ve které existují datové typy **signed** a **unsigned** a pomocí které můžeme **převádět vektory na signed a unsigned a integer a zpět**!
  - **signed** = binárně zapsané číslo se znaménkem, **unsigned** = bez znaménka
  - existuje „neoficiální“ knihovna, **std_arith** (a také std_logic_unsigned a std_logic_signed), které dovolují provést aritm. operace s logickými datovými typy, ale **tato knihovna není standardizována a neměla by být používána!**



- **konverzní funkce:**
- **std_logic_vector <-> signed/unsigned**
- **signed/unsigned -> integer**
  - funkce to_integer
- **integer -> signed/unsigned**
  - funkce to_signed, to_unsigned
  - kromě čísla (integeru) je nutné uvést, na kolik bitů má být číslo převedeno

VHDL – Podmínkové konstrukce

- **paralelní podmínkové konstrukce – with-select, when-else**
  - příklad multiplexoru 4:1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux is
port (a : in std_logic;
      b : in std_logic;
      c : in std_logic;
      d : in std_logic;
      sel : in std_logic_vector(1 downto 0);
      x : out std_logic);
end mux;
architecture Concurrent of mux is
begin
with sel select
x <= a when "00",
     b when "01",
     c when "10",
     d when "11",
    'X' when others;
end Concurrent;
```

- deklarace základní IEEE knihovny, **ALL** upřesňuje, že použijeme vše z knihovny
- **entity** – jménem mux, deklarace portů: 1bitové std_logic vstupy (a, b, c, d), sel je 2bitový std_logic_vector vstup (1 downto 0) a x je 1bitový std_logic výstup
- **architecture** – jménem Concurrent (naše pojmenování) of „jméno entity“ architektura začíná **begin** a končí „**end jméno architektury**“ (nebo jen „**end**“ nebo „**end architecture**“)
- paralelní prostředí – žádný proces
- pomocí **with**-**select** testujeme vstup sel a na základě jeho stavu přepínáme na výstup x symbolem **<=** jeden ze vstupů a, b, c, d
- sel je vektor – musíme použít " "
- **'X'** – neurčitý stav – poslední podmínkou **others** ošetříme všechny zbývající možnosti (neurčitý stav např.) vstupu sel (není nutné, ale vhodné)
- **možnosti se nesmí překrývat – pouze jedna platná!**

- **paralelní podmínkové konstrukce – with-select, when-else**
  - ten samý multiplexor 4:1

```
x <= a when sel="00" else
     b when sel="01" else
     c when sel="10" else
     d when sel="11" else
    'X';
```

- stejný multiplexor realizovaný pomocí **when-else** paralelní podmínky (zbytek VHDL kódu je stejný)
- **when-else je obecnější podmínka** – v každém kroku můžeme testovat jinou podmínku na rozdíl od with-select
- při with-select v prvním řádku nejprve specifikujeme, který port (signál) testujeme a pak vyjmenováváme jednotlivé možnosti (hodnoty) a určujeme výstup – **první platná podmínka je provedena, zbytek přeskočen!**
- **when-else vede na implementaci prioritního kodéru, with-select je implementován jako multiplexor!**

```
sel1 : in std_logic_vector(1 downto 0);
sel2 : in std_logic_vector(1 downto 0);

x <= a when (sel1="00" and sel2="10") else
     b when (sel1(1)='1' or not sel2="00") else
     c when sel2="11" else
     d when (sel1 xor sel2)="11" else
    'X';
```

- multiplexor se dvěma řídícími vstupy (sel1, sel2)
- pomocí when-else můžeme testovat různé hodnoty vstupů a jejich kombinace
- **and** – logický AND, **or** – logický OR, **not** – logická negace, **xor** – logický XOR
- **sel1(1)** – logická hodnota na pozici (1) vstupního vektoru portu sel1

VHDL – Podmínkové konstrukce

- **sekvenční podmínky – if, case**
  - stále ten samý multiplexor

```
architecture Sequential of mux is
begin
process(a,b,c,d,sel)
begin
if sel="00" then
        x<=a;
elsif sel="01" then
        x<=b;
elsif sel="10" then
        x<=c;
elsif sel="11" then
        x<=d;
else
        x<='X';
end if;
end process;
end Sequential;
```

- **if a case jsou sekvenční příkazy** – musíme je vždy zapisovat pouze v těle procesu
- **proces má svůj citlivostní seznam** – určujeme vstupy které spustí proces, v tomto případě všechny vstupy multiplexoru: a, b, c, d, sel
- každý proces začíná a končí **begin** - **end**
- syntaxe podmínky if je: **if** – **then**
- pomocí **elsif** specifikujeme další možnosti, kterých může testovaný vstup nabývat
- pomocí **else** ošetříme všechny ostatní možnosti (pokud ani jedna z výše uvedených není splněna)
- if podmínky jsou **testovány sekvenčně** – postupně v procesu odshora dolů jsou podmínky testovány jedna po druhé -> pokud se narazí na platnou podmínku (true), je **vykonán VHDL kód za slovem then a celý blok if je ukončen**! -> **první podmínka true je vykonána, zbytek je přeskočen**! **– prioritní kodér = pořadí je priorta**
- každý blok podmínek if musí být zakončen klíčovým slovem **end if**
- **if podmínky můžeme různě kombinovat a vnořovat** – pozor ale při vnořování podmínek, abychom dokázali ošetřit všechny možnosti!

- **sekvenční podmínky – if, case**

```
if sel(1)='0' then
        if sel(0)='0' then
                x<=a;
        else
                x<=b;
        end if;
elsif sel(1)='1' then
        if sel(0)='0' then
                x<=c;
        else
                x<=d;
        end if;
else
        x<='X';
end if;
```

- ukázka vnořování if podmínek – můžeme otestovat každou pozici vektoru sel zvlášť – výsledkem bude zcela stejný multiplexor (stejný netlist a implementace, odlišný RTL)
- každý blok **if musíme zakončit vlastním end if!**
- pomocí if můžeme testovat **složené podmínky** (více vstupů a jejich kombinací) pomocí závorek () a logických funkcí, AND, OR, XOR…
- pozor v jazyce VHDL na rozdíl mezi **elsif** – **else if**!
- **elsif slouží pro pokračování předchozí podmínky if**, testování další možnosti, která se v tomto bloku podmínky if může objevit, jedná se tedy o pořád ten samý blok if a zakončíme jej tak celý **jedním end if**
- **else if** je v podstatě:
  else
    if ….
  jedná se tedy o **zcela samostatnou podmínku if vnořenou** do předchozí podmínky do jejího stavu else
  proto **musíme zakončit tuto vnořenou podmínku vlastním end if i vyšší podmínku pomocí jejího end if**!

VHDL – Podmínkové konstrukce

- **sekvenční podmínky – if, case**
  - ten samý multiplexor – realizace pomocí case

```
process(a,b,c,d,sel)
begin
case sel is
        when "00" => x<=a;
        when "01" => x<=b;
        when "10" => x<=c;
        when "11" => x<=d;
        when others => x<='X';
end case;
end process;
```

- podobná syntaxe jako with-select
- case – sekvenční příkaz -> musíme použít proces
- poslední řádek vždy ošetření všech ostatních možností pomocí **when others**
- **každý case musí být zakončen end case!**
- můžeme v jednom řádku i dvě možnosti – **symbol |** (nebo)
- u integeru můžeme testovat i rozsah – **x to y** (1 to 3)
- u case se **nesmí jednotlivé možnosti překrývat** – vždy musí platit přesně jen jedna možnost (na rozdíl od if)
- implementace: **if – prioritní kodér, case – multiplexor**

- jaký je rozdíl? kdy použít paralelní přiřazování a kdy sekvenční podmínky?
- v tomto příkladu jednoduchého multiplexoru je to jedno – všechny uvedené způsoby vedou na stejnou realizaci a implementaci
- ale obecně –
  - paralelní přiřazování používáme obvykle při realizaci kombinačních obvodů, podmíněné přiřazování výstupu, jednoduché podmínky (např. multiplexor)
  - sekvenční podmínky jsou nejčastěji používány pro realizaci sekvenčních synchronních obvodů, detekce hran hodinového signálu, vnořené podmínky (čítače, automaty…)

- **synchronní obvody – detekce hran hodinového vstupu (signálu)**
  - ve VHDL většinou realizujeme **synchronní obvody** (asynchronní jen v případě elementárních kombinačních obvodů – sčítačky, multiplexory, komparátory, apod.)
  - pro jejich synchronizaci používáme **hodinový vstup** (**clock**, **clk**) a jeho hrany
  - hodinový signál při SW návrhu používáme jako běžný logický signál – při HW implementaci s ním syntezátor pracuje trochu odlišně (viz distribuce hodin v FPGA)
  - pro synchronizaci celého obvodu, jeho procesů, komponent, využíváme detekci vzestupných či sestupných hran hodinového signálu – 3 metody:

```
process(clock)
begin
if clock='1' and clock'event then
…. -- prikazy
end if;
end process;
```

```
process(clock)
begin
if rising_edge(clock) then
…. -- prikazy
end if;
end process;
```

```
process
begin
wait until rising_edge(clock);
…. -- prikazy

end process;
```

- všechny 3 uvedené metody jsou ekvivalentní a mají stejný efekt – proces je vykonáván jen při detekci vzestupné hrany hodinového vstupu clock – první 2 používají **podmínku if**, poslední **příkaz wait**
- pokud použijeme **if podmínku**, vstup clock **musí být v citlivostním seznamu procesu**, naopak při použití **wait, proces nesmí mít žádný citlivostní seznam!**

VHDL – Detekce hran hodinového vstupu

- **synchronní obvody – detekce hran hodinového signálu**

```
process(clock)
begin
if clock='1' and clock'event then
…. -- prikazy
end if;
end process;
```

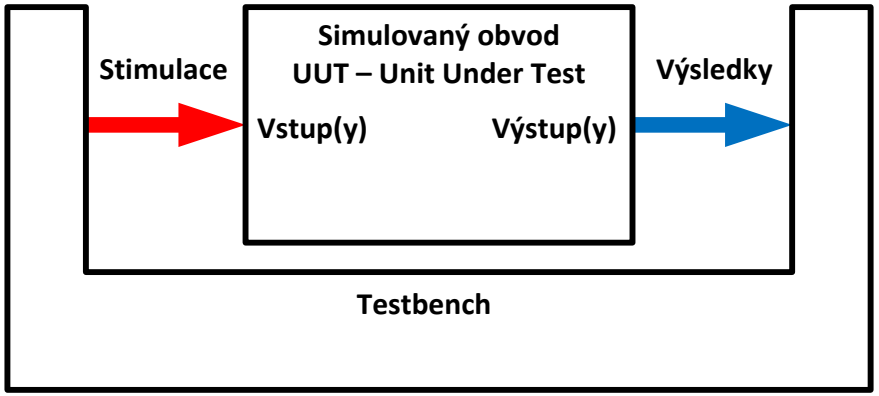
```
process(clock)
begin
if rising_edge(clock) then
…. -- prikazy
end if;
end process;
```

```
process
begin
wait until rising_edge(clock);
…. -- prikazy

end process;
```

- **metody:**
  1. použití **atributu 'event** – složená if podmínka detekuje stav clock v logické 1 a současně jeho změnu (event) = vzestupná hrana
  2. VHDL jazyk obsahuje přímo funkci pro detekci vzestupné hrany signálu/portu/proměnné – **rising_edge**(*objekt*) detekuje vzestupnou hranu objektu
  3. příkaz **wait until společně s funkcí rising_edge** – proces čeká, dokud není splněna podmínka (detekce vzestupné hrany) – v jazyce VHDL několik typů wait (wait on, until, for – některé jsou syntetizovatelné, jiné nikoliv!)
- jak upravit jednotlivé metody výše pro detekci sestupné hrany vstupu clock?

1. **if** clock='0' **and** clock'event **then**
2. **if** falling_edge(clock) **then**
3. **wait until** falling_edge(clock);

- **použití jazyka VHDL pro simulace obvodů**
  - stejný jazyk VHDL můžeme kromě syntézy obvodů použít i pro jejich simulaci! – navíc máme k dispozici příkazy určené jen pro simulace – např. **wait for *čas***
  - pro simulaci obvodu vytvoříme speciální VHDL modul – tzv. **VHDL testbench**
  - simulovaný obvod je označen jako – **Unit Under Test** – **UUT**
  - v testbench modulu zapíšeme buzení obvodu (tzv. stimulace) – **zapíšeme, jaké hodnoty se mají objevit na vstupech obvodu v určených časech**
  - simulátor pak zobrazí **výstupní hodnoty (reakci) UUT při daném buzení** a také můžeme zobrazit **vnitřní stavy UUT (hodnoty signálů, někdy i proměnných)**
  - simulace můžeme provádět v různých fázích syntézy a implementace obvodu – nejčastěji používané jsou tyto 2:
    1. **Behaviorální (RTL) simulace** – simulace v průběhu syntézy, SW simulace ideálního chování navrženého obvodu bez přihlédnutí k vlastnostem HW (žádná zpoždění…)
    2. **Post-route simulace** – simulace v poslední fázi před vlastní implementací, realistická simulace reálného obvodu implementovaného do zvoleného HW (realistické chování, zpoždění…)
  - celá řada placených simulátorů i zdarma: iSIM, ModelSim,…



- **použití jazyka VHDL pro simulace obvodů**
  - Jak vypadá typický VHDL testbench?
  - testbench je v podstatě entita, UUT (testovaná entita) je její komponenta
  - testbench tedy obsahuje signály, které pomocí mapování portů (viz dále) namapujeme na porty UUT
  - v těle architektury testbenche jsou pak **stimulační procesy** – buzení jednotlivých signálů-portů UUT, v případě synchronních obvodů je potřeba **vytvořit hodinový signál**
  - můžeme využít některé příkazy jazyka VHDL, které nejsou určeny pro syntézu, ale pouze pro simulace obvodů – např. **wait for „čas“** apod.

```
process
constant Clock_period: time := 10 ns;
begin
Clock <= '0';
wait for Clock_period/2;
Clock <= '1';
wait for Clock_period/2;
end process;
```

- **proces generující periodický hodinový signál pro simulaci**
  - **konstanta** Clock_period – hodnota periody hod. signálu – **datový typ time** použitelný pouze pro simulace – zde f_{Clock} = 100 MHz
  - **wait for** – čekání určenou dobu, použitelné jen v simulacích
  - proces generuje v signálu Clock periodicky logickou 1 a logickou 0 v polovině zadaného časového intervalu -> hodinový signál Clock s frekvencí 100 MHz

- **příklady v jazyce VHDL – klopný obvod D a jeho simulace**
  - základní klopný obvod D řízený vzestupnou hranu hodinového vstupu – behaviorální popis

```
entity Dff is
port (D : in std_logic;
      clock : in std_logic;
      Q : out std_logic;
      nonQ : out std_logic);
end Dff;

architecture Behavioral of Dff is
begin

process(clock)
begin
if clock='1' and clock'event then
            Q <= D;
            nonQ <= not D;
end if;
end process;

end Behavioral;
```

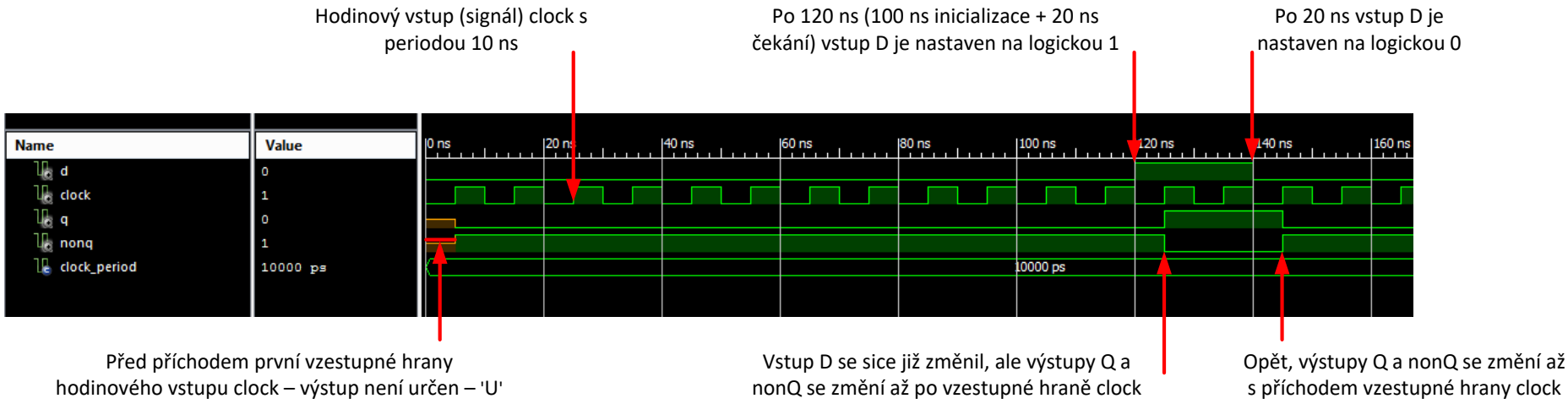
- **klopný obvod D** – s každou vzestupnou hranou hodinového vstupu clock překopíruje hodnotu ze vstupu D na výstup Q a negaci D na výstup nonQ
- **vstupy obvodu**: D a clock, **výstupy obvodu**: Q a nonQ
- pro behaviorální popis nám stačí jen jeden proces se vstupem clock v citlivostním seznamu a dále jedna podmínka pro detekci vzestupné hrany hodinového vstupu (např. metoda 1)
- s každou příchozí vzestupnou hranou, vstup D je vložen na výstup Q: **Q <= D**; a jeho negace na výstup nonQ: **nonQ <= not D**;

- **příklady v jazyce VHDL – klopný obvod D a jeho simulace**
  - proveďme nyní Behaviorální (RTL) simulaci pomocí kódu vlevo

```
stim_proc: process
begin
wait for 100 ns;
D <= '0';
wait for 20 ns;
D <= '1';
wait for 20 ns;
D <= '0';
wait;
end process;
```

- vytvoříme VHDL testbench a zapíšeme buzení, např.:
- v simulátoru je automaticky nastavena perioda hodinového vstupu 10 ns (můžeme klidně změnit)
- buzení obsahuje jen jeden jednoduchý proces:

1. nejprve **čekáme 100 ns** – na začátku obvykle čekáme na inicializaci obvodu než začneme s buzením
2. poté nastavíme na vstup **D logickou 0** a tuto hodnotu podržíme na vstupu **20 ns**
3. po 20 ns na vstup **D nastavíme logickou 1** opět **na 20 ns**
4. poté **vstup D nastavíme zpět na logickou 0 napořád**
5. příkaz wait bez určení doby čekání (**wait;**) znamená **nekonečně dlouhé čekání**!



- **klopný obvod D se synchronním resetem a jeho simulace**
- pokud reset (R) = logická 1, při detekci vzestupné hrany clock -> reset klop. obvodu

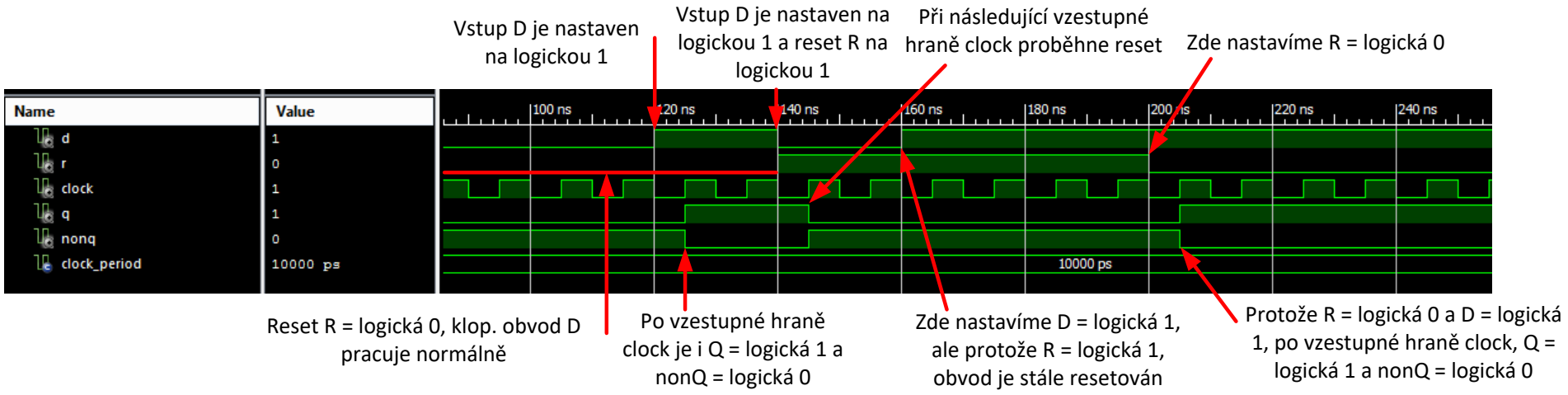
```
entity Dff is
port (D, clock, R : in std_logic;
      Q, nonQ : out std_logic);
end Dff;

architecture Behavioral of Dff is
begin
process(clock)
begin
if clock='1' and clock'event then
        if R='1' then
                Q <= '0';
                nonQ <= '1';
        else
                Q <= D;
                nonQ <= not D;
        end if;
end if;
end process;
end Behavioral;
```

- deklarace portů entity – můžeme pomocí čárky **sdružit více portů se stejným směrem a datovým typem**, viz. D, clock, R a Q, nonQ
- totéž lze i při deklaraci signálů, proměnných apod.
- přidáme do předchozího kódu **nový port R** (reset)
- **synchronní reset** – **reset R je spouštěn (kontrolován) vzestupnými hranami vstupu clock**
- proto **citlivostní seznam procesu obsahuje jen vstup clock – ten je použit jako jediný řídící vstup!**
- použijeme **vnořené if podmínky** – pokud je reset aktivován, výstup Q = logická 0 (nonQ = logická 1), jinak klopný obvod D funguje normálně (na výstup Q se kopíruje vstup D, na nonQ negace D)

- **behaviorální (RTL) simulace navrženého klop. obv. D se synchronním resetem**

```
wait for 100 ns;
D <= '0';
wait for 20 ns;
D <= '1';
wait for 20 ns;
R <= '1';
D <= '0';
wait for 20 ns;
D <= '1';
wait for 40 ns;
R <= '0';
wait;
```

- **synchronní reset = kontrolován hodinovým vstupem**
- při aktivaci resetu – obvod se resetuje až při příchodu nejbližší následující vzestupné hraně hodinového vstupu clock



- **klopný obvod D s asynchronním resetem a jeho simulace**
- klop. obvod D **s asynchronním resetem = reset není vázán na stav hodin clock**

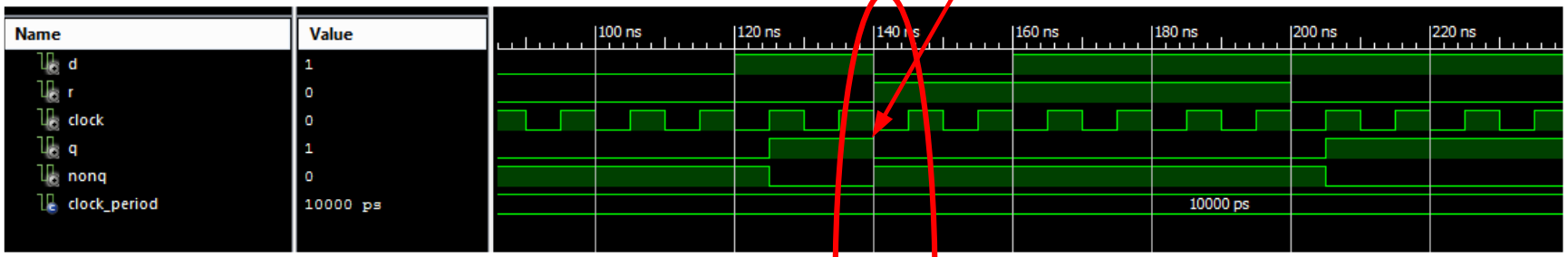
```
entity Dff is
port (D, clock, R : in std_logic;
      Q, nonQ : out std_logic);
end Dff;

architecture Behavioral of Dff is
begin
process(clock, R)
begin
if R='1' then
            Q <= '0';
            nonQ <= '1';
elsif clock='1' and clock'event then
            Q <= D;
            nonQ <= not D;
end if;
end process;
end Behavioral;
```

- asynchronní reset = není kontrolován hodinovým vstupem clock – **je proveden ihned, okamžité resetování klopného obvodu**
- reset R je tak řídícím vstupem, **R musí být uveden v citlivostním seznamu procesu**
- stačí jen úprava if podmínek – nejprve testujeme stav vstupu reset, pokud není aktivní, testujeme následně přítomnost vzestupné hrany clock
- na rozdíl od předchozího případu jen 1 blok if podmínek = **jen 1 end if;**

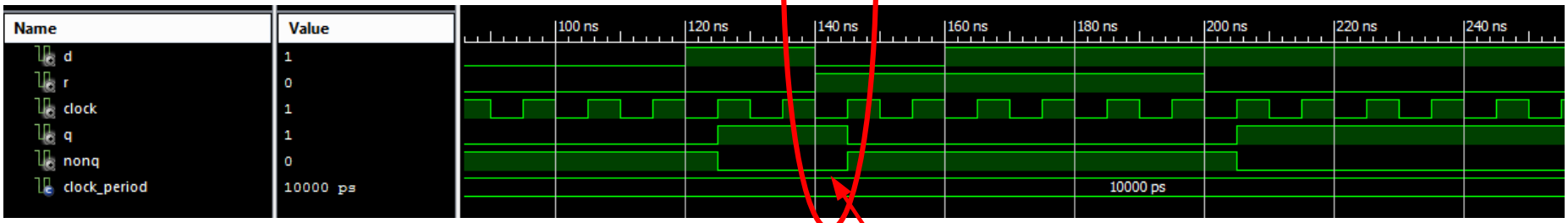
- **Behaviorální (RTL) simulace klop. obvodu D s asynchronním resetem**
- použijeme stejné buzení (testbench) pro synchronní i asynchronní verzi resetu

Když aktivujeme asynchronní reset (R = logická 1), resetuje se klopný obvod D okamžitě bez čekání na vzestupnou hranu clock

Asynchronní reset



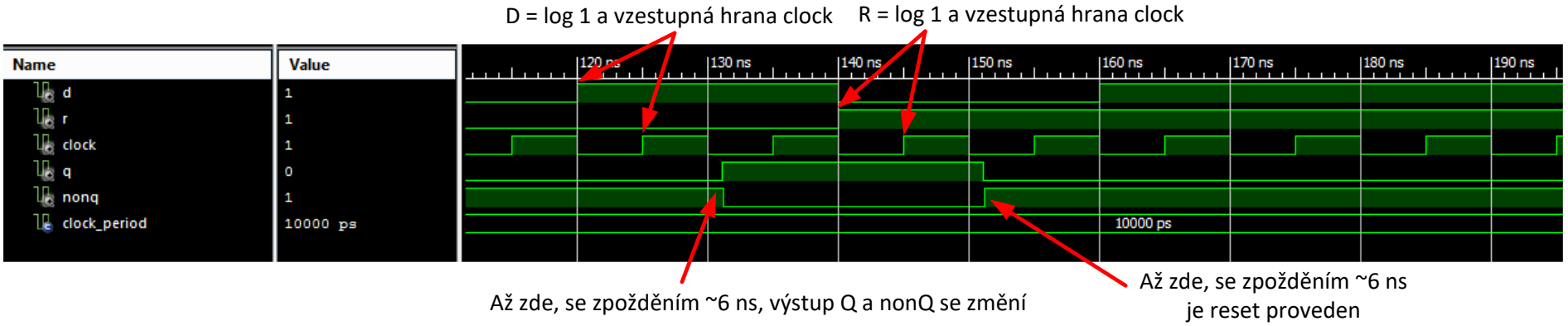
Synchronní reset

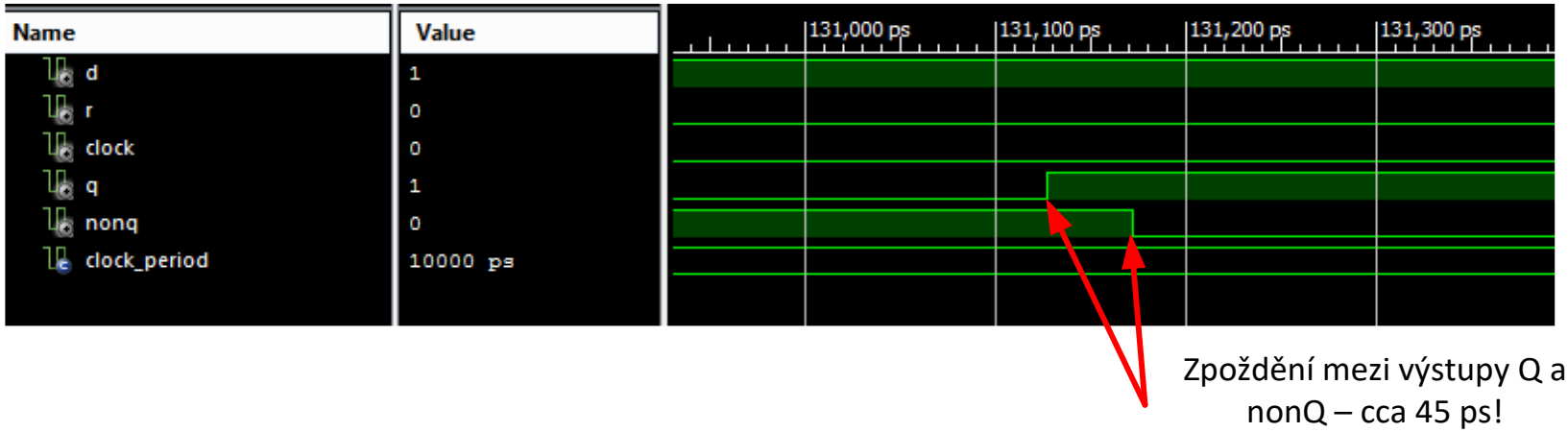


Když aktivujeme synchronní reset, je nutné na jeho provedení počkat na vzestupnou hranu clock - zpoždění

- **asynchronní reset** – výhoda – reset je proveden okamžitě (bez zpoždění), nevýhoda – nesynchronní výstup (metastabilita) – v jazyce VHDL obvykle přísně dbáme na dodržení synchronnosti – **záleží na aplikaci** (požadovaném obvodu)
- **synchronní reset** – nevýhoda – zpoždění čekáním na vzestupnou hranu clock, výhoda – plně synchronní výstup, žádný problém s nesynchronností

- **post-route simulace, je synchronní reset skutečně synchronní?**
- pokud použijeme HW kit, přiřadíme piny vstupům/výstupům, abychom získali realistické zpoždění včetně zpoždění cest





- **post-route simulace** – získáme reálné doby zpoždění a časové charakteristiky obvodu – **dobu zpoždění**, **zpoždění cest**, **zpoždění výstupů**…
- někdy se stane, že behaviorální simulace je OK (navržený obvod je OK), ale post-route simulace odhalí jeho **chybnou funkci (návrh) vlivem reálných dob zpoždění v FPGA!**

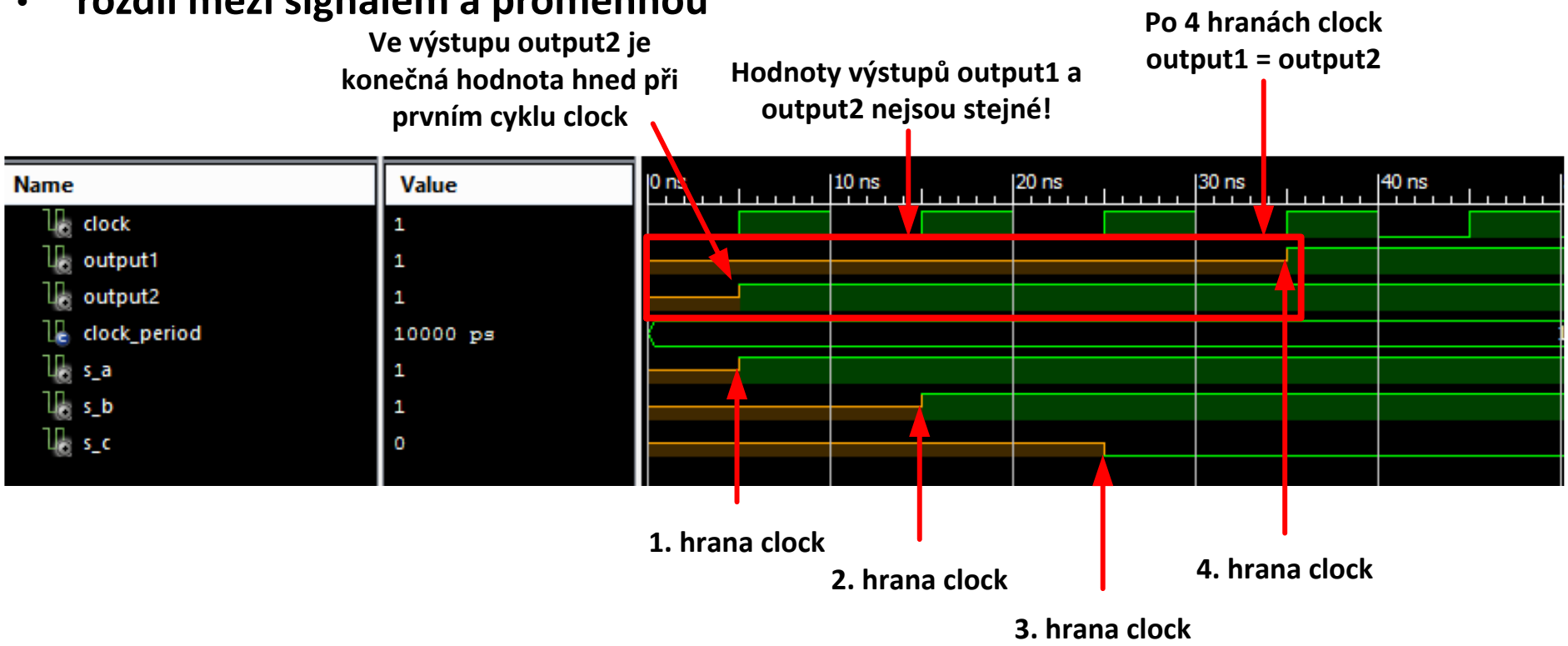
VHDL – Rozdíl mezi signálem a proměnnou ve VHDL

```
entity sigvar is
port (clock : in std_logic;
      output1, output2 : out std_logic);
end sigvar;

architecture Behavioral of sigvar is
signal s_a, s_b, s_c : std_logic;
begin

process(clock)
variable v_a, v_b, v_c : std_logic;
begin
if clock='1' and clock'event then
            s_a <='1';
            s_b <= s_a and '1';
            s_c <= s_a xor s_b;
            output1 <= not s_c;
            v_a :='1';
            v_b := v_a and '1';
            v_c := v_a xor v_b;
            output2 <= not v_c;
end if;
end process;
end Behavioral;
```

- **rozdíl mezi signálem a proměnnou**
- základní rozdíl jsme již zmínili v předchozí přednášce – zápis hodnoty do proměnné je okamžitý, zápis do signálu má zpoždění tzv. delta time
- ukažme si to názorně na příkladu – entita se 2 výstupy
- deklarujeme 3 signály – s_a, s_b a s_c (signál_a, signál_b, signál_c)
- a deklarujeme také 3 proměnné – v_a, v_b, v_c
- **signály jsou deklarovány přímo v bloku architektury, proměnné deklarujeme v procesu** (signály existují v celé architektuře, proměnné existují jen v tomto procesu)
- proces **spouštíme na vzestupnou hranu clock**
- v těle procesu provedeme přesně ty samé operace se signály i s proměnnými
- výstup signálů zapíšeme do output1 a output2 obsahuje výstup proměnných
- do signálu zapisujeme hodnotu symbolem **<=** ale pro zápis do proměnných používáme znak **:=**
- jsou hodnoty obou output1 a output2 stejné?

VHDL – Rozdíl mezi signálem a proměnnou ve VHDL

- **rozdíl mezi signálem a proměnnou**

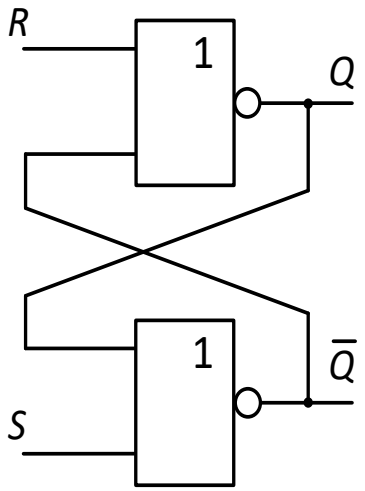


- hodnoty proměnných jsou aktualizovány okamžitě, takže po **první hraně clock**: v_a = '1', v_b = '1', v_c = '0', output2 = '1'
- hodnoty všech signálů jsou zapsány (updatovány) vždy až na konci procesu po **delta time**:
  po 1. hraně clock: s_a = '1', s_b = 'U' and '1' = 'U', s_c = 'U' xor 'U' = 'U', output1 = 'U'
  po 2. hraně clock : s_a = '1', s_b = '1', s_c = '1' xor 'U' = 'U', output1 = 'U'
  po 3. hraně clock : s_a = '1', s_b = '1', s_c = '1' xor '1' = '0', output1 = 'U'
  po 4. hraně clock : s_a = '1', s_b = '1', s_c = '0', output1 = '1'
- pozor při užívání a rozhodnutí proměnná vs. signál – **výsledný obvod se může často chovat zcela odlišně pokud zvolíme pro jeho realizaci signály nebo proměnné!**

- **Strukturální popis ve VHDL a mapování portů (port-map)**
- úkol – vytvořte asynchronní klop. obvod RS pomocí strukturálního popisu
- strukturální popis – nejnižší úroveň abstrakce, popisujeme v podstatě „zapojení“ obvodu

- asynch. obvod RS se skládá ze 2 hradel NOR a 2 zpětných vazeb mezi nimi, postup bude tedy:

1. vytvoříme entitu hradla NOR se vstupy *a*, *b* a výstupem *c* – entita gate_nor
2. vytvoříme entitu RS klop. obvodu se vstupy *R*, *S*, a výstupy *Q*, *nonQ* – entita RS
3. architekturu entity RS vytvoříme pomocí 2 komponent gate_nor zapojených dle schématu – zpětné vazby - „dráty“ realizujeme pomocí signálů a pomocí tzv. mapování portů (port-map) definujeme propojení komponent a signálů

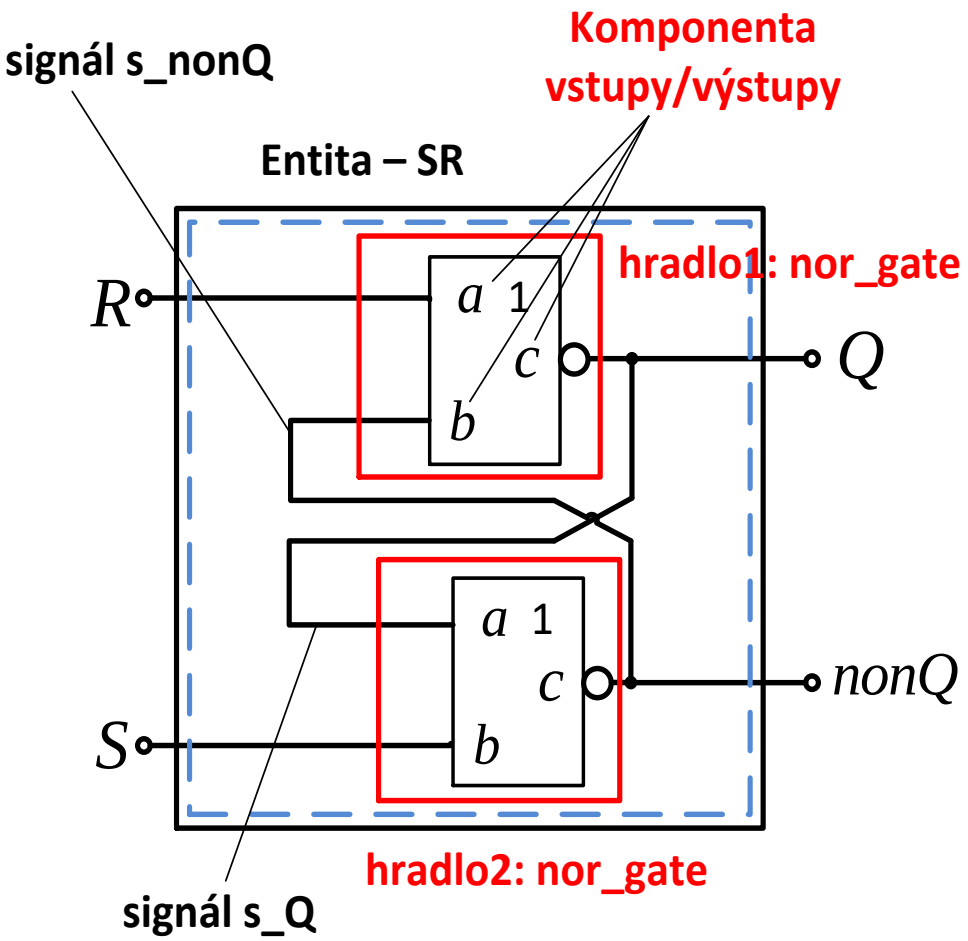


- **mapování portů (port-map)**
- mapování portů ve VHDL je procedura, kdy definujeme propojení a zapojení jednotlivých komponent, bloků a signálů uvnitř dané entity – VHDL umožňuje modulární hierarchický popis
- díky tomu v tomto příkladu zapojíme obě hradla NOR a obě zpětné vazby do struktury RS klopného obvodu
- 2 druhy mapování portů ve VHDL: **poziční mapování**, **jmenné mapování**

- **Strukturální popis ve VHDL a mapování portů (port-map)**
- entita hradla NOR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity nor_gate is
port (a, b : in std_logic;
      c : out std_logic );
end nor_gate;

architecture RTL of nor_gate is

begin

c <= a nor b;

end RTL;
```

- *a*, *b* jsou vstupy, *c* je výstup
- jednoduchá RTL realizace použitím klíčového slova **nor**



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RS is
port (R, S : in  std_logic;
      Q, nonQ : out std_logic);
end RS;

architecture Structural of RS is

component nor_gate is
port (a,b : in std_logic;
      c : out std_logic);
end component;

signal s_Q, s_nonQ : std_logic;

begin
hradlo1: nor_gate
port map (R, s_nonQ, s_Q);

hradlo2: nor_gate
port map (s_Q, S, s_nonQ);

Q <= s_Q;
nonQ <= s_nonQ;
end Structural;
```

- entita klop. obvodu RS, s názvem RS, má 4 porty – R, S jsou vstupy, Q, nonQ výstupy
- uvnitř architektury deklarujeme použití **komponenty nor_gate** – deklarace spočívá ve zkopírování portů deklarovaných pro tuto entitu nor_gate
- potřebujeme 2 „dráty“ pro 2 zpětné vazby – deklarujeme 2 signály s_Q a s_nonQ
- nyní musíme zapsat, že v entitě RS se nacházejí 2 komponenty nor_gate (2 hradla NOR) a jak jsou propojeny
- **hradlo1:nor_gate** – první část je pojmenování komponenty v rámci entity RS (naše pojmenování) : druhá část je název modulu komponenty – nor_gate
- druhý řádek je **poziční mapování** – klíčové slovo **port map**, do závorky **zapíšeme ve stejném pořadí deklarace portů komponenty jejich připojení na porty v entitě**: *a* – připojeno na *R*, *b* – připojeno na *s_nonQ*, *c* – připojeno na *s_Q*
- totéž provedeme s druhým hradlem NOR – pojmenujeme jej hradlo2, je opět komponentou nor_gate a provedeme opět mapování jeho portů
- **poziční mapování – musíme dodržet pořadí portů deklarované komponenty, přiřadíme jim porty entity!**

```
hradlo1: nor_gate
port map (a => R,
          b => s_nonQ,
          c => s_Q);

hradlo2: nor_gate
port map (a => s_Q,
          b => S,
          c => s_nonQ);
```

vs.

```
hradlo1: nor_gate
port map (b => s_nonQ,
          a => R,
          c => s_Q);

hradlo2: nor_gate
port map (c => s_nonQ,
          a => s_Q,
          b => S);
```

- **VHDL strukturální popis – mapování portů**
- mapování portů (port map):

1. **poziční** – viz předchozí slide
2. **jmenný** – s použitím jmen portů

- **jmenné mapování portů**
- jeho syntaxe:

  port map (komponenta port1 => entita port1/signál1,
  komponenta port2 => entita port2/signál2,
  …
  komponenta port => entita port/signál);

- **výhoda** – porty můžeme mapovat v libovolném pořadí, nemusíme dodržovat pořadí deklarace, viz vlevo
- **poziční mapování vs. jmenné mapování**
- **poziční** – jednodušší, kratší zápis, ale musíme dodržet pořadí portů tak, jak jsou deklarovány (když chceme přidat/upravit port, musíme dodržet pořadí! – při velkém množství portů nepřehledné)
- **jmenné** – delší zápis, ale nemusíme dodržovat pořadí portů (snadnější úpravy a přidávání nových portů)

- **VHDL-93** (update VHDL jazyka vydaný jako IEEE 1076-1993) přinesl řadu úprav a zjednodušení – mimo jiné deklaraci a užití komponenty

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RS is
port (R, S : in  std_logic;
      Q, nonQ : out std_logic);
end RS;

architecture Structural of RS is

component nor_gate is
port (a,b : in std_logic;
      c : out std_logic);
end component;

signal s_Q, s_nonQ : std_logic;

begin
hradlo1: nor_gate
port map (R, s_nonQ, s_Q);

…
end Structural;
```

vs.

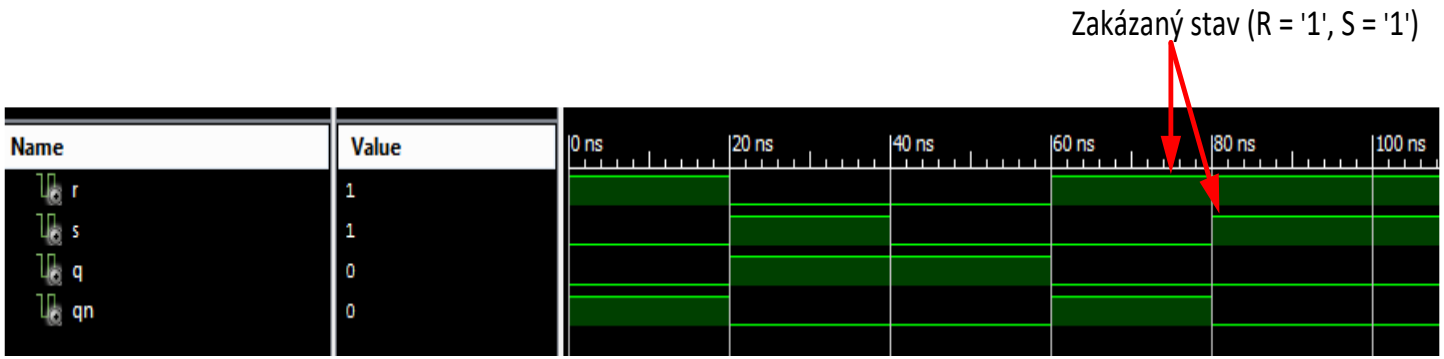
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RS is
port (R, S : in  std_logic;
      Q, nonQ : out std_logic);
end RS;

architecture Structural of RS is

signal s_Q, s_nonQ : std_logic;

begin
hradlo1: entity work.nor_gate
port map (R, s_nonQ, s_Q);

…
end Structural;
```

- v architektuře není třeba komponentu deklarovat
- při jejím použití syntaxe: hradlo1: **entity** work.nor_gate
- klíčové slovo **entity** a cesta k dané komponentě – **work** představuje dynamický prostor daného projektu

- **VHDL strukturální realizace RS klop. obvodu pomocí mapování portů**
  - simulace navrženého obvodu

```
stim_proc: process
begin
R <='1';
S <='0';
wait for 20 ns;
R <='0';
S <='1';
wait for 20 ns;
R <='0';
S <='0';
wait for 20 ns;
R <='1';
S <='0';
wait for 20 ns;
R <='1';
S <='1';
wait for 20 ns;
wait;
end process;
```

- simulace vstupních kombinací, včetně zakázaného stavu
- co se stane s výstupem v zakázaném stavu?



- **RS klopný obvod – behaviorální popis, co se zakázaným stavem?**
- behaviorálním popisem můžeme vytvořit RS klopný obvod: **priorita Set**, **priorita Reset**
- priorita znamená, že daný vsup převáží a výstup RS klop. obvodu bude určen jeho stavem
- vstupy R, S a výstupy Q, nonQ – ve VHDL různý způsob realizace

priorita Set

```
process(S,R)
begin
if S='1' then
        Q <= '1';
        nonQ <= '0';
elsif R='1' then
        Q <= '0';
        nonQ <= '1';
else
        null;
end if;
end process;
```

priorita Reset

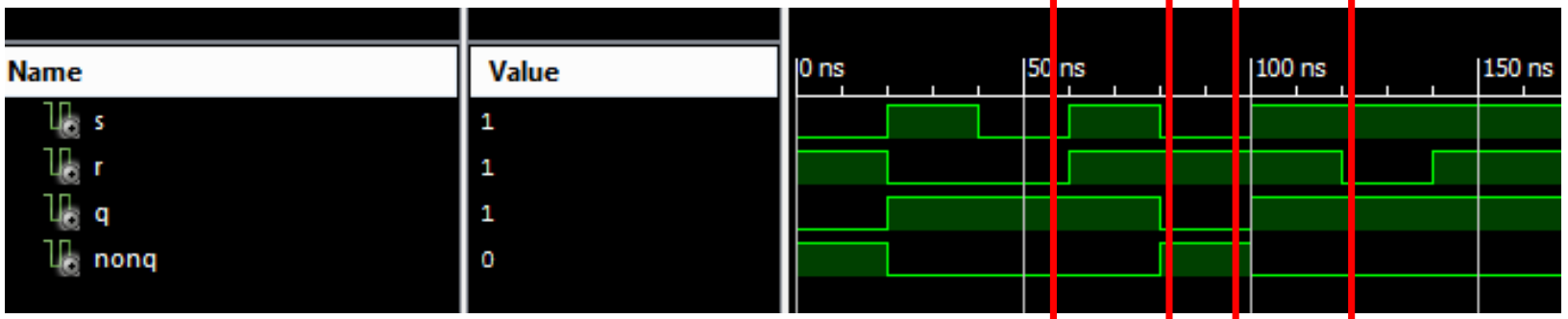
```
process(S,R)
begin
if R='1' then
        Q <= '0';
        nonQ <= '1';
elsif S='1' then
        Q <= '1';
        nonQ <= '0';
else
        null;
end if;
end process;
```

- prioritu vytvoříme jednoduše – pouze upravíme pořadí podmínek if
- první část if podmínky je dominantní (má prioritu) – pokud platí, je zbylá část přeskočena
- klíčové slovo **null;** – jeho použití znamená, že se nemá nic vykonat -> při této možnosti podmínky se nic neprovede

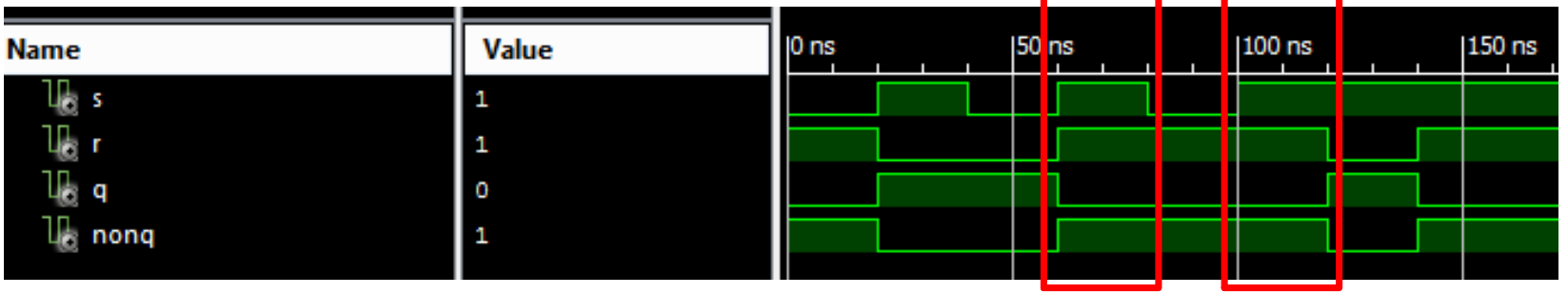
- **RS klopný obvod – behaviorální popis, co se zakázaným stavem?**
- behaviorálním popisem můžeme vytvořit RS klopný obvod: **priorita Set**, **priorita Reset**

Zakázaný stav
(R = '1', S = '1')

priorita Set

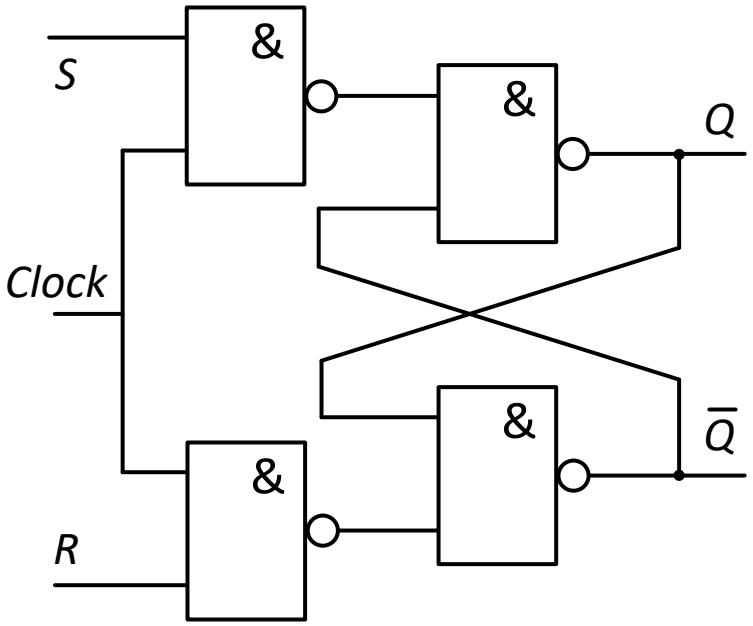


priorita Reset

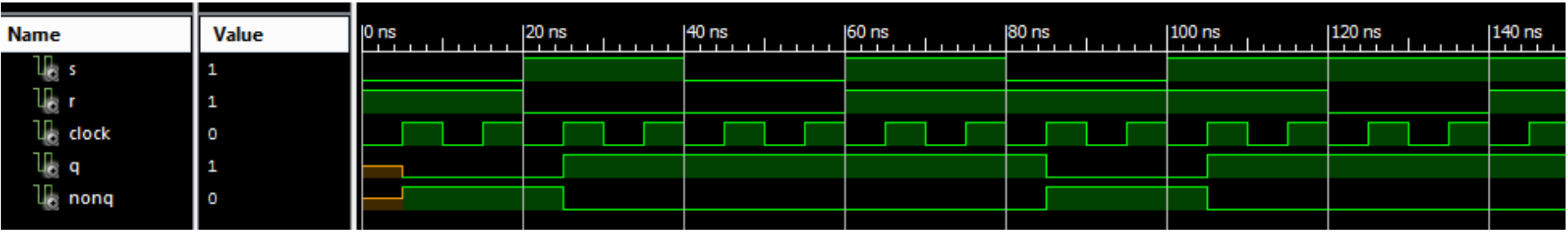


- jak upravit asynchronní RS klopný obvod na synchronní řízený hranou? – přidáme vstup Clock a detekci jeho hrany

- synchronní klopný obvod RS s prioritou na Set či Reset – přidáme vnořenou if podmínku pro detekci vzestupné hrany clock




```
process(clock)
begin
if clock='1' and clock'event then
        if S='1' then
                Q <= '1';
                nonQ <= '0';
        elsif R='1' then
                Q <= '0';
                nonQ <= '1';
        else
                null;
        end if;
end if;
end process;
```



- **Parametrizace VHDL kódu – generic**
  - **generic** – slouží pro parametrizaci návrhu entity, komponenty v jazyce VHDL
  - jedná se o parametr, kterým můžeme ovládat (specifikovat) různé části VHDL kódu
  - generic deklarujeme v první části entity (jako porty) a definujeme jejich datový typ a výchozí hodnotu, na rozdíl od portů však ne směr (nejsou směrové)
  - datový typ generic – nejčastěji integer
  - nejedná se o port, ačkoliv rovněž provádíme jeho **mapování** při použití komponenty obsahující část generic – pomocí **generic map**
  - např. můžeme měnit rozsah integeru, délku vektoru, počet cyklů smyčky, počet pozic rotace/posuvu registru, aritmetické operace, porovnání hodnoty v podmínce….
  - parametrizace nám umožňuje jednoduše z jednoho místa měnit hodnoty v různých částech VHDL kódu – parametrizovat navržený obvod
  - hodnoty tohoto parametru (generic) jsou dostupné v celé architektuře (globální), nebo můžeme parametrizovat jen určitou komponentu (port map)
- **generic syntaxe:**

  entity *název entity* **is**
  generic (*seznam generic*);
  port (*seznam portů*);
  end *název entity*;
  - ***seznam generic*** – *název* : *datový typ* := *počáteční hodnota*;

- **Parametrizace VHDL kódu – generic**
  - příklad – obecný w-bitový registr, 2 nejvyšší bity vysunuty ven

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity registr is
generic (w : integer := 7);
port (Clock : in std_logic;
        registr_in : in std_logic_vector(w downto 0);
        registr_out : out std_logic_vector(w downto w-1));
end registr;

architecture Behavioral of registr is
begin
process(Clock)
variable reg : std_logic_vector (w downto 0);
begin
if Clock='1' and Clock'event then
                reg := registr_in;
                registr_out<=reg(w downto w-1);
end if;
end process;
end Behavioral;
```

- **klíčové slovo generic** – v části entity před deklarací portů
- parametr w – délka registru – vložíme do něj např. 7 (8bitový registr)
- **generic** – bez směru
- nyní můžeme parametr ***w*** použít jednoduše v deklaracích a práci s ostatními porty, signály a proměnnými
- parametr ***w*** můžeme použít i v rámci různých operací

- **Parametrizace VHDL kódu – generic**
  - příklad – použijme předchozí návrh parametrizovaného registru jako komponentu

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Reg is
port (Clock : in std_logic;
      Reg_in : in std_logic_vector(15 downto 0);
      Reg_out : out std_logic_vector(15 downto 14));
end Reg;

architecture Behavioral of Reg is
component registr
generic (w : integer);
port (Clock : in std_logic;
      registr_in : in std_logic_vector(w downto 0);
      registr_out : out std_logic_vector(w downto w-1));
end component;
begin
Reg:registr
generic map(15)
port map(Clock,Reg_in,Reg_out);
end Behavioral;
```

- entita **Reg** – 16bitový registr, pozice 16 a 15 vysunuty ven
- komponenta **registr** – předchozí parametrizovaný registr
- při mapování komponenty provedeme nejprve mapování parametrů generic **– generic map**
- opět 2 možnosti generic map – **poziční a jmenné**:
  generic map(15)
  generic map(w=>15)
- po generic map následuje standardní port map – **mezi oběma mapováními se nepíše středník (;)!**