Programovatelná hradlová pole FPGA, jazyk VHDL

FPGA, jazyk VHDL, základy

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

FPGA, VHDL – Programovatelná pole a jazyky, vývoj

- HDL – **Hardware Description Language** – nejedná se o programovací jazyk! **v jazyce VHDL neprogramujeme!**, navrhujeme, simulujeme, provádíme syntézu digitálních obvodů tím, že je popisujeme – description language = **jazyk pro popis**!
- společně s programovatelnými poli – **vývoj „programovacích" a HDL jazyků**
  - původní PLA a PAL programovány „manuálně" editováním konfiguračního souboru s přepalováním cest a propojů v poli
  - **PALASM** = PAL Assembler, jazyk vytvořený na počátku 80. let
    - pro všechny výstupy obvodu byly vyjádřeny Booleovy rovnice v textové formě pomocí PALASM a následně překonvertovány do konfiguračního souboru cest
  - **CUPL** = Compiler for Universal Programmable Logic, v roce 1983
    - první komerční (původně proprietární) jazyk, pro PC a MS-DOS systém
  - **ABEL** = Advanced Boolean Expression Language, v roce 1984
    - popis Booleovými rovnicemi, pravdivost. tabulkami, přechody stavového aut., simulace
  - **VHDL** = VHSIC Hardware Description Language, první IEEE standard vydán 1987
    - IEEE standard, současná verze 2008 – navržený digitální obvod nezávislý na implementaci, měl by fungovat v HW různých výrobců (ne vždy pravda)
    - lze použít pro syntézu i simulace digitálních obvodů – 3 úrovně abstrakce, modularita
    - jednodušší pro začátečníky, používá se spíše v Evropě, Japonsku, Koreji
  - **Verilog**, první IEEE standard vydán 1984
    - rovněž IEEE standard, základní idea stejná, jiná syntaxe, další odlišnosti
    - připomíná spíše jazyk C, složitější pro začátečníky, rozšířen hlavně v J a S Americe, Indii, JV Asii…

FPGA, VHDL – základy jazyka VHDL

- **VHDL** = VHSIC Hardware Description Language
  - VHSIC = Very High Speed Integrated Circuits – rychlé integrované obvody, FPGA, CPLD, ….
    - syntaxe jazyka VHDL může připomínat známé programovací jazyky (C, Java,…), ale **hardware pro implementaci (FPGA, CPLD) je zcela odlišný od mikroprocesorů!**
    - v jazyce VHDL popisujeme strukturu a chování logického obvodu = **modelujeme fyzickou strukturu logických obvodů**

- **Model logického obvodu -> RTL model (Register Transfer Level) -> Hradla**
  - nejprve v jazyce VHDL vytvoříme model (popis) logického obvodu – více možností: model chování, model činnosti (pomocí Booleových rovnic), model zapojení
  - druhý krok syntéza – syntezátor převede náš model na tzv. RTL model (model činnosti obvodu) a z něj vygeneruje „zapojení“ obvodu na úrovni hradel
  - ve fázi syntézy – můžeme definovat dílčí parametry, optimalizace (z hlediska zpoždění, počtu hradel,…), dodatečné podmínky
  - v jazyce VHDL můžeme rovněž obvody simulovat – v jednotlivých fázích: simulace chování, simulace činnosti, simulace zapojení…
  - poslední krok – implementace obvodu do cílového HW

- v jazyce VHDL **3 úrovně abstrakce** = **3 způsoby modelování logických obvodů**
  - **Behaviorální popis** – model chování obvodu
  - **Dataflow popis (RTL)** – model činnosti obvodu
  - **Strukturální popis** – model zapojení obvodu

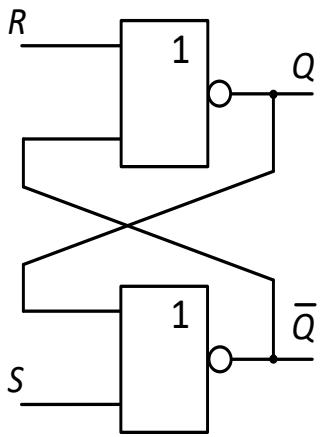
1. **Behaviorální popis** – modelování chování obvodu
   - nejvyšší úroveň abstrakce
   - „programátorský“ způsob – obvykle nejjednodušší a rychlý způsob popisu, flexibilní
   - modelovaný obvod je „black box“ – nevíme z čeho se skládá, neznáme jeho zapojení
   - známe ale chování obvodu – známe výstupy obvodu při jednotlivých vstupních kombinacích –> umíme vyjádřit chování (reakci) obvodu při daném buzení vstupů
   - implementace obvodu do výsledného HW je ale vždy založena na vygenerovaném zapojení hradel (netlist) –> to za nás musí v tomto případě udělat syntezátor
   - nevýhoda – neznáme zapojení obvodu, nemůžeme zkontrolovat zda syntezátor správně pochopil náš model – jednoduše při drobné změně popisu chování můžeme vytvořit zcela jiný obvod, nebo zbytečně složitý obvod apod.
   - nad procesem syntézy máme jen velmi malou kontrolu, vše dělá syntezátor automaticky
   - je potřeba dodržet doporučený model popisu obvod (např. čítače, sčítačky, násobičky, stavového automatu…) – máme jistotu, že jej syntezátor správně implementuje
   - **příklad – asynchronní klopný obvod RS**
     - použitím pravdivostní tabulky můžeme popsat chování:
     - S a R jsou vstupy, Q a nonQ jsou výstupy obvodu
     - if S = 1 a R = 0 pak Q = 1 a nonQ = 0 (jedničková transformace)
       if S = 0 a R = 1 pak Q = 0 a nonQ = 1 (nulová transformace)
       if S = 0 a R = 0 pak Q = Q a nonQ = nonQ (paměťová transformace)
       if S = 1 a R = 1 pak Q = X and nonQ = X (zakázaný stav) – ošetření si ukážeme později

2. **Dataflow (RTL) popis** – modelování činnosti obvodu

- střední úroveň abstrakce
- RTL = **Register Transfer Level** = modelování obvodu na úrovni registrů
- registry propojené kombinační logikou
- opět neznáme přesné zapojení obvodu – ale známe činnost obvodu -> víme, jaké logické operace jsou prováděny se vstupními proměnnými, dokážeme zapsat činnost obvodu pomocí Boolových rovnic
- syntezátor ve VHDL stále za nás automaticky vygeneruje zapojení obvodu (netlist) – ale v tomto případě víme víc o struktuře obvodu (známe použité logické operace)
- nižší šance, že syntezátor vytvoří chybný obvod než v případě behaviorálního popisu, máme tak určitou (omezenou) kontrolu nad procesem implementace
- snáze můžeme implementovat úpravy a modifikace obvodu
- **stejný příklad – asynchronní klopný obvod RS**
  - při použití dataflow (RTL) popisu – klopný obvod vyjádříme pomocí jeho Booleových rovnic:
  - Q = R NOR nonQ
    nonQ = S NOR Q

3. **Strukturální popis** – modelování zapojení obvodu

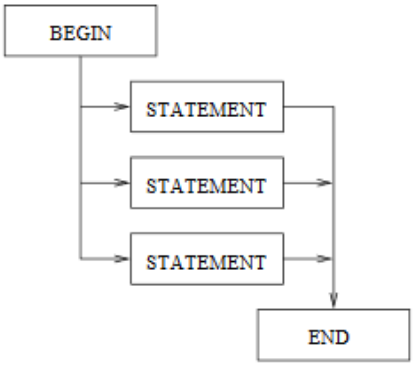
- nejnižší úroveň abstrakce, přesně známe vnitřní stavbu obvodu – známe použitá logická hradla a jejich zapojení, dokážeme sami vygenerovat jejich seznam (netlist)
- maximální kontrola nad procesem syntézy a implementace – sami můžeme optimalizovat zapojení obvodu z hlediska zpoždění, spotřeby, počtu hradel…
- sami vytvoříme model zapojení obvodu – syntezátor jej pouze převede na netlist
- obvykle nejnáročnější způsob popisu (na čas a délku kódu), při drobné změně funkce obvodu obvykle musíme provést rozsáhlé změny kódu, méně flexibilní způsob
- **opět stejný příklad – asynchronní klopný obvod RS**
  - pomocí strukturálního popisu – „nakreslíme“ schéma zapojení obvodu v kódu jazyka VHDL (můžeme použít i schematický editor, ale i přímo kód VHDL)
  - nejprve vytvoříme jedno elementární hradlo NOR
  - dále použijeme 2 tato hradla NOR a navzájem je propojíme do struktury RS klopného obvodu (definujeme tzv. signály VHDL – „dráty“ v obvodu)
  - poslední krok, definujeme propojení vstupů a výstupů hradel NOR k vstupům a výstupům RS klopného obvodu

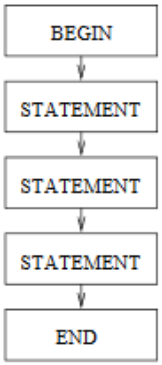


- **3 způsoby popisu v jazyce VHDL**
  - každý má své výhody a nevýhody, jaký zvolit? – záleží na situaci, navrhovaném obvodu – v jazyce VHDL je ideální kombinovat všechny 3 dohromady, i v rámci jednoho modulu
  - např. jednoduché základní bloky můžeme vytvořit behaviorálním popisem a pak je propojit strukturálním -> obvody složené z velkého množství hradel, strukturální popis složitý -> RTL popis, potřebujeme vyjádřit činnost obvodu pomocí Booleových rovnic

- **VHDL concurrent (paralelní) vs. sequential (sekvenční) prostředí**
  - **concurrent** = paralelní – všechny příkazy a části kódu jsou vykonávány naráz (paralelně)
    - nezáleží na pořadí v jakém jsou zapsány
    - concurrent = představuje kombinační logické obvody -> pevné „zadrátování“ hradel a bloků obvodu, změna vstupů – okamžitá změna výstupů
  - **sequential** = krok za krokem – VHDL příkazy a kód je vykonáván krok za krokem v pořadí zápisu, sequential = sekvenční logika

- **jazyk VHDL** (a pole FPGA) jsou **defaultně paralelní prostředí** = **kombinační logika**
- pro zápis **sekvenčního kódu** musíme použít tzv. **proces** (viz dále)
- v jednom VHDL modulu můžeme klidně **kombinovat paralelní i sekvenční části** – jen pozor na časové chování jednotlivých částí!

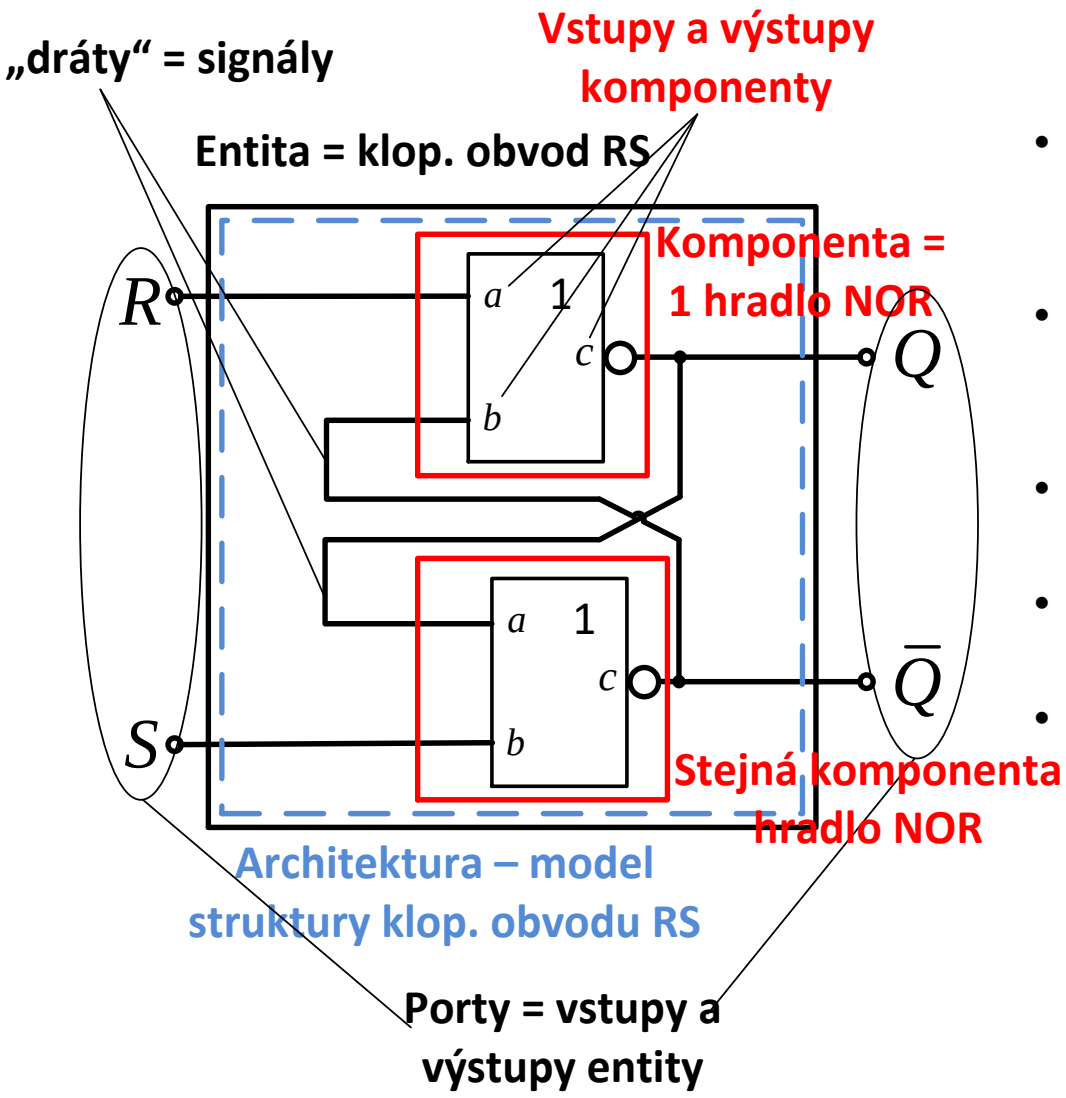




- **jazyk VHDL – jak vypadá typický VHDL modul a co obsahuje?**
  - ukázka čistého VHDL modulu

```
-- volitelné textové záhlaví, může např. obsahovat
-- jméno tvůrce, datum, verzi, atd.

library IEEE;
-- deklarace použitých knihoven (viz dále)

entity test is
-- deklarace entity a jejích portů (viz dále)
end test;

architecture Behavioral of test is
-- sem zapisujeme tzv. architekturu entity
-- také zde deklarujeme signály, konstanty…

component test2
-- pokud využíváme komponenty, deklarujeme zde
end component;

begin
-- vlastní VHDL kód architektury
end Behavioral;
```

1. **hlavička** – může obsahovat libovolný text a komentáře,
   komentáře ve VHDL píšeme „**--**“
2. **deklarace knihoven** – všechny příkazy, funkce, operace jsou obsaženy v knihovnách VHDL
3. **deklarace entity** – entita ve VHDL představuje výsledný navrhovaný obvod, definujeme její název (test) a její porty = vstupy a výstupy
4. **deklarace architektury** – architektura představuje vlastní popis chování nebo činnosti nebo zapojení (dle způsobu) obvodu, definujeme její název
   jedna entita – klidně více architektur
5. **komponenta** – pokud je entita (obvod) složen z menších dílčích bloků, označujeme tyto bloky jako komponenty (viz dále)



- **komponenta** – entita v rámci vyšší entity – **modularita ve VHDL**
  rozložíme obvod na samostatné elementární menší části, „sub-bloky“, a jejich propojením získáme „vyšší, složitější” výsledný obvod
- **signály** – ve VHDL nahrazují „dráty“, propojování komponent, portů, entit, nemají žádný směr (obousměrné)
- **port** – interface, rozhraní, kterým daná komponenta a entita komunikuje – jsou vždy směrové (viz dále)
- **VHDL není case sensitive** (velká – malá písmena)
- většina příkazů (výrazů) je zakončena **středníkem ;** - není nutné odřádkování
- **může mít jedna entita více architektur? – ve VHDL ano!**
  často lze ten samý obvod popsat několika odlišnými způsoby – jeden může být vhodnější pro simulaci obvodu, jiný pro implementaci…
  z entity pak vybereme architekturu, která se má použít

- **VHDL základy – datové typy, datové objekty, porty, procesy**
- **datové typy**
  - při deklaraci každého portu, signálu, proměnné, konstanty **musíme vždy určit datový typ objektu** – jaký typ dat a v jakém formátu je může daný objekt přenášet
  - syntaxe – ***název datového objektu*** **:** ***název datového typu***;
  - číselné datové typy: **integer**, **real**, natural, positive
  - logické typy: **std_logic** (standard logic), std_ulogic, bit, boolean, **std_logic_vector**
  - jiné: string, character, time, array, bit_vector, signed, unsigned
  - ve VHDL můžeme definovat vlastní datový typ – výčtem
  - a vytvořit subtyp – omezení základního typu, např. rozsah integeru, délku vektoru, atd.
  - **std_logic**, **std_ulogic – resolved (rozhodnuté)**
    - základní logický datový typ, tzv. MVL-9 (Multi Value Logic 9)
    - **s „u”** = unresolved, **bez „u”** = resolved, častěji používáme resolved, výčtový typ:
      'U' uninitialized – ještě neinicializovaná hodnota v simulaci
      'X' unknown – neznámá hodnota, nelze při simulaci určit
      '0' logická 0 – logická 0
      '1' logická 1 – logická 1
      'Z' stav vysoké impedance – třístavová logika, stav vysoké impedance (odpojení)
      'W' weak unknown – „slabá“ neznámá, hodnota má blíže k neznámé, ale není 'X'
      'L' weak low – „slabá“ logická 0, hodnota zřejmě blízko log. 0, ale není to '0'
      'H' weak high – „slabá“ logická 1, hodnota zřejmě blízko log. 1, ale není to '1'
      '-' don't care – neurčitý stav (může být cokoliv)

- **std_logic vs. std_ulogic**
  - **ulogic** = unresolved = **nerozhodnuté** logické hodnoty
  - **logic** = resolved = **rozhodnuté** logické hodnoty
  - **typ logic je subtypem ulogic typu** – tzv. **rozhodovací (resolution) funkce**
  - proč? situace – máme datovou sběrnici (vodič), ke které jsou připojeny výstupy dvou a více obvodů, výstup prvního je třeba logická 1 a druhého logická 0 -> co bude tedy výsledná hodnota na sběrnici? – **pokud použijeme ulogic, výsledkem je chyba -> nelze**
  - **rozhodovací funkce** – tabulka definující výsledek kombinace dvou libovolných logických hodnot – datový typ **ulogic převede na logic**
  - kdykoliv v jazyce VHDL zapojíme dva výstupy dohromady, je použita rozhodovací funkce

```
-------------------------------------------------------------------
-- resolution function
-------------------------------------------------------------------
CONSTANT resolution_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -      |   |
--      ---------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
    );
```

  - dříve návrháři obvodů v jazyce VHDL **používali spíše ulogic datový typ** (a museli sami definovat výsledek spojených výstupů), dnes se již v 99,9% případů **používá typ logic**

- **VHDL základy – datové typy, datové objekty, porty, procesy**
- **std_logic_vector = logický vektor = sběrnice logických hodnot**
  - častá situace v logických obvodech – potřebujeme sdružit několik výstupů do sběrnice (svazku výstupů, „drátů“), např. výstup 4bitového čítače = 0001
  - VHDL – **logický vektor** = **sběrnice jednotlivých std_logic** (signálů, portů, proměnných…)
  - sdružíme několik samostatných std_logic do jednoho std_logic_vector, proč?
  - díky vektoru můžeme pak pracovat s celou sběrnicí naráz, ale samozřejmě lze stále pracovat s každým vodičem (portem, signálem…) vektoru zvlášť
  - jedna logická hodnota ve VHDL – **zapisujeme pomocí apostrofů**, např. **'1' = logická 1**
  - vektor (sběrnice) – **zapisujeme do uvozovek**, např. **"01" = vektor logická 0, logická 1**
  - při deklaraci musíme vždy uvést **délku vektoru** = **počet bitů (drátů) sběrnice**
  - ve VHDL zvykem číslovat od nuly a sestupně (**downto**), ale samozřejmě můžeme číslovat i vzestupně (**to**) a nemusíme od nuly, příklady:
    std_logic_vector (3 downto 0) – 4 „dráty“, očíslované 3, 2, 1, 0
    std_logic_vector (0 to 5) – 6 „drátů“, očíslovaných 0, 1, 2, 3, 4, 5
    std_logic_vector (5 downto 2) – 4 „dráty“, očíslované 5, 4, 3, 2
  - při deklaraci **přiřadíme každému „drátu“ jeho pozici ve vektoru** -> tu si pak musíme pamatovat, pokud s ní chceme pracovat, např:
    signal test : std_logic_vector (3 downto 0) – signál s názvem test je vektor se 4 „dráty“ očíslovanými 3, 2, 1, 0
    test(2)<= '1'; – do „drátu“ na pozici č. 2 (druhý zleva) vložíme logickou 1

FPGA, VHDL – základy jazyka VHDL

- **číselné typy real, integer, natural, positive**
  - rozsahy číselných typů se mohou lišit dle HW a implementace
  - **real** – reálná čísla v rozsahu -1.10^{38} až $+1.10^{38}$
  - **integer** – 32bitová celá čísla v rozsahu $-(2^{31}-1)$ až $+(2^{31}-1)$
  - **natural** – nezáporná celá čísla 0 až $+(2^{31}-1)$, **positive** – kladná celá čísla 1 až $+(2^{31}-1)$
  - při deklaraci portu, signálu, proměnné číselného typu **můžeme omezit jeho rozsah**:
    signal test integer; - syntezátor automaticky alokuje plný rozsah 32 bitů
    signal test integer range 0 to 15; - omezíme rozsah od 0 do 15, syntezátor alokuje pouze 4 bity (stačí pro daný rozsah) – výrazně ušetříme HW prostředky!
  - pozor – '1' je logická 1, zatímco 1 je číslo (integer) 1 – vzájemně nekompatibilní!
- **boolean, bit, character, string, signed, unsigned, time**
  - **boolean** – 2 hodnoty: true, false – nekompatibilní s std_logic -> **'1' ≠ true ve VHDL**!
  - **bit** – 2 hodnoty: '0', '1', subtyp od std_logic, můžeme vytvořit vektor -> **bit_vector**
  - **character** – může být jeden povolený znak ve VHDL, zapisujeme pomocí **' '**
  - **string** – vektor znaků (character), musíme určit délku stringu (počet znaků), **" "**
  - **signed**, **unsigned** – binární číslo s a bez znaménka, spolu s IEEE knihovnou numeric_std převádíme z/do logických hodnot a používáme aritmetické operace
  - **time** – celočíselný formát, definovány též jednotky času (např. sec, ms, min, hr…)
- **subtype** – pomocí omezení můžeme definovat vlastní podtyp základního typu:
  subtype small_integer **is integer range** 0 **to** 9;
  subtype my_vector **is** std_logic_vector(3 **downto** 0);

- **type** – ve VHDL můžeme vytvořit vlastní datový typ – vhodné pro stavové automaty
- jedná se vždy o **výčtový typ** – musíme vyjmenovat všechny možné hodnoty typu
- vlastní datový typ deklarujeme v části architektura, nebo v balíčku (package), knihovně (library) apod.

- syntaxe při definování vlastního datového typu:
  type (*jméno typu*) **is** (*výčet všech možných hodnot typu oddělených čárkou ","*);
  - následně můžeme definovaný typ použít při deklaraci portů, signálů, proměnných atd.:
    název datového objektu* : *název datového typu;
  - příklady:
    type den **is** (pondeli, utery, streda, ctvrtek, patek, sobota, nedele);
    signal dnes : den;
  - pozor – i když dva různé typy obsahují přesně ty samé hodnoty, nejsou tyto typy stejné a nelze je navzájem zaměňovat, používat pro stejné proměnné, signály, porty atd.:
    type den **is** (pondeli, utery, streda, ctvrtek, patek, sobota, nedele);
    type dny **is** (pondeli, utery, streda, ctvrtek, patek, sobota, nedele);
    signal dnes : den;
    signal zitra : dny;
    dnes <= zitra; -- nelze, chyba, každý signál je jiného typu!!
  - definování vlastního typu pomocí omezení rozsahu (to, downto), např.:
    type mala_cisla **is integer range** 0 **to** 100;
    signal hodnota : mala_cisla;
  - u vlastního datového typu můžeme rovněž jako parametr definovat **jednotky** (units)

- **signály (signal), proměnné (variable), konstanty (constant) = datové objekty**
- datové objekty souží k ukládání, přenášení, načítaní, definování … hodnot
- **signál** – „drát“ v jazyce VHDL, propojuje komponenty, porty, procesy, přenáší, propojuje
  - **představuje fyzickou část obvodu – „drát“** – fyzická reprezentace vodiče
  - **nejsou orientované** (žádný vstupní/výstupní směr) – výjimkou jsou procedury (později)
  - **globální** – deklarovány v architektuře -> můžeme k nim přistupovat v celé architektuře
  - **signál deklarujeme jménem a datovým typem** – libovolný datový typ
  - při deklaraci můžeme do signálu vložit počáteční hodnotu, nebo ji určit až při použití
    signal test : std_logic :='1'; – deklarace signálu jménem „test“ standardního logického datového typu s počáteční hodnotou logická 1
    signal test : std_logic_vector(1 downto 0); – vektor o velikosti 2 bez počáteční hodnoty
  - při deklaraci přiřazujeme hodnotu do signálu pomocí znaku **:=**
  - přiřazení hodnoty do signálu při jeho použití později provádíme pomocí znaku **<=**
    test <= "10"; – přiřadili jsme hodnoty logická 1 a logická 0 do signálu test
  - do každého bitu (drátu) vektoru můžeme samozřejmě přiřadit hodnotu samostatně:
    test(1) <= '1';
    test(0) <= '0'; – výsledek bude stejný jako výše
  - vytvoříme signál typu logický vektor a chceme do něho přiřadit hodnotu "0001":
    signal test : std_logic_vector(3 downto 0) := "0001";
    signal test : std_logic_vector(3 downto 0) :=(0=>'1', 1=>'0',2=>'0',3=>'0'); – pozice => hodnota
    signal test : std_logic_vector(3 downto 0) := (0=>'1',others=>'0'); – klíčové slovo others

- **signály, proměnné, konstanty – datové objekty ve VHDL**
  - **proměnná = variable** – dočasná buňka v paměti pro uložení libovolné hodnoty
    - **nemá žádnou fyzickou interpretaci v obvodu** – pouze hodnota někde v paměti
    - **lokální** – deklarována jen v určitém procesu, lze k ní přistupovat jen v tomto procesu, nikde jinde mimo něj neexistuje!
    - **sdílená (shared) variable** – deklarována v architektuře (jako signál) a je přístupná v celé architektuře (ve všech procesech)
    - **proměnná má své jméno a určený datový typ** – opět libovolný
    - opět můžeme přiřadit při deklaraci počáteční hodnotu, nebo nemusíme
      variable test : std_logic := '1'; – deklarace proměnné „test“ standardního logického typu s počáteční hodnotou logická 1
      variable test : std_logic_vector(1 downto 0); – proměnná „test“ logický vektor bez počáteční hodnoty
    - pro přiřazení hodnoty do proměnné při deklaraci i později slouží vždy znak **:=**
      test := "10"; – přiřadili jsme do proměnné „test“ hodnoty logická 1 a logická 0
    - různé metody přiřazování a práce s proměnnými jsou zcela stejné, jako se signály

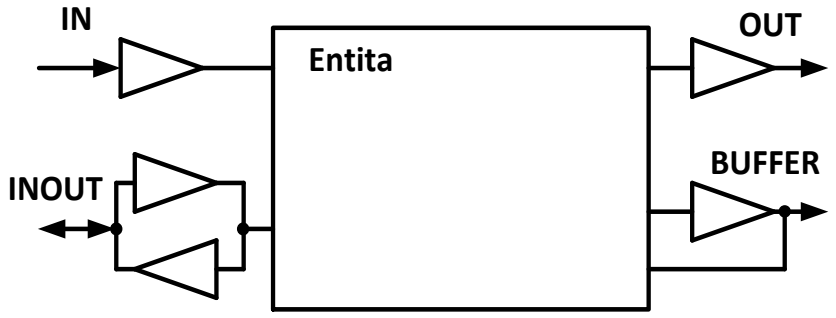
- **pozor – zásadní rozdíl mezi proměnnou a signálem ve VHDL!**
  - **proměnná** – přiřazení hodnoty do proměnné je provedeno **okamžitě**
  - **signál** – přiřazení hodnoty do signálu v procesu je provedeno **až na konci procesu po tzv. delta době (delta time)!**
  - ukážeme si názorně na příkladu později

- **signály, proměnné, konstanty – datové objekty ve VHDL**
  - **konstanta = constant** – je proměnná, do které přiřadíme jen počáteční hodnotu při její deklaraci a dále tuto hodnotu nemůžeme nikde měnit (jen read-only proměnná)
  - konstantu můžeme deklarovat na více místech:
    na začátku architektury (jako signály a sdílené proměnné) – pak lze konstantu využívat v celé architektuře
    v konkrétním procesu (jako proměnné) – pak ji lze využít jen v tomto procesu
    v balíčku (package) – tzv. deferred (odložená) konstanta, hodnotu konstanty můžeme dodatečně definovat později až při jejím vyvolání z balíčku

  - jak pracovat se signály, proměnnými, konstantami a kdy co použít?
  - **signál** – propojování komponent, portů, bloků, přenos hodnot mezi procesy, lze zobrazit hodnotu v simulátoru, delta zpoždění při aktualizaci hodnoty signálu (na konci procesu)
  - **proměnná** – nesdílená (standardní) proměnná existuje jen v rámci daného procesu, sdílená proměnná je dostupná v celé architektuře, použití zejména v čítačích, děličkách, pro ukládání mezivýsledků při různých operacích, některé simulátory umožňují zobrazit její hodnotu v simulaci (Xilinx iSIM ne), její hodnota je aktualizována okamžitě
  - **konstanta** – pevně předdefinovaná proměnná (read-only), použití např. při definování určité hodnoty na počátku (počet bitů vektoru, rozsah čítače) apod.

- **porty – ports**
  - port – **komunikační rozhraní entity**, vstup, výstup entity (komponenty)
  - stejně jako v případě deklarace signálů, proměnných, konstant musíme určit jeho **název a datový typ** a počet bitů (drátů) v případě typu vektor
  - pro **zápis** na výstupní port symbol **<=**, pro **čtení** vstupního portu symbol **=>**
  - u portu musíme navíc určit jeho typ – **směr komunikace –> 4 typy (5 typů)**:

1. **port in** – vstupní port, smíme pouze číst jeho hodnotu danou vnějším zdrojem (read only)
2. **port out** – výstupní port, smíme pouze zapisovat hodnoty vystupující z entity (write only)
3. **port buffer** – výstupní port se zpětnou vazbou, smíme zapisovat hodnotu ven, ale následně i tuto hodnotu použít zpět (write only s možností zpětného použití hodnoty)
4. **port inout** – plně obousměrný port, můžeme zapisovat i číst hodnoty (read, write)



- jaký je rozdíl mezi portem typu **out** a portem typu **buffer**?
  - **na port out smíme pouze zapsat hodnotu**, tuto hodnotu nemůžeme jinde použít
  - příklad – negovaný výstup nonQ klopného obvodu – pro negovaný výstup nonQ potřebujeme udělat negaci výstupu Q: nonQ <= NOT Q; – ale to nesmíme pokud je Q port typu out! – abychom mohli Q znegovat, musíme znát jeho hodnotu! řešení?

- **porty**
  - řešení negace výstupu klopného obvodu? – 2 možnosti:
    1. deklarovat port Q jako typ buffer – díky zpětné vazbě známe jeho hodnotu a můžeme ji znegovat
    2. nebo vytvoříme pomocný signál (třeba Q_pomocny), uložíme do něj hodnotu, která se má objevit na výstupu klopného obvodu, a pak jej zapíšeme přímo na výstup Q a jeho negaci na výstup nonQ: Q <= Q_pomocny; Q <= NOT Q_pomocny;
  - **inout port**
    - inout port umí plně obousměrnou komunikaci – proč jej nepoužít pro všechny porty?
    - **2 problémy inout portů** – HW a SW
    - HW implementace inout portů v reálných polích FPGA je **komplikovaná a náročná**, obousměrný budič a buffer jsou nezbytné, často existují různá HW omezení pro konkrétní FPGA pole od výrobců, vyšší zpoždění apod.
    - SW problém – práce s inout portem vyžaduje **implementovat třístavovou logiku** řízenou stavovým automatem – např. nejprve zapíšeme hodnotu na výstup portu, pak jej přepneme do stavu vysoké impedance Z (odpojíme výstup) a čekáme na odpověď protistrany, pokud přijde, přečteme její hodnotu a pak opět…
    - díky tomu se snažíme obvykle vyhnout použití inout portů pokud je to možné – samozřejmě jsou aplikace, kdy to nejde a inout je nezbytný – např. obousměrné sběrnice typu IC, I2C, 1-wire communication, atd.

- **proces (process) v jazyce VHDL**
  - prostředí VHDL je defaultně paralelní (concurrent) – co se sekvenčními příkazy?
  - **proces** – slouží k zapouzdření sekvenčních částí kódu (např. if, case, loop, atd.)
  - proces používáme zejména pro návrh sekvenčních obvodů
  - procesy zapisujeme uvnitř architektury
  - architektura může obsahovat libovolný počet procesů – **všechny procesy běží navzájem paralelně** (současně), ale příkazy **uvnitř každého z procesů jsou vykonávány sekvenčně**
  - proces lze pojmenovat (návěstí), má začátek a konec (**begin, end**) a tzv. **citlivostní seznam**
  - **citlivostní seznam** = seznam vstupů procesu, na změnu alespoň jednoho je proces spuštěn
  - příklad procesu

```
test: process(vstup)
begin
vystup<=vstup;
end process;
```

- entita má 2 porty – **vstup** je port in, **vystup** je port out
- název procesu „test“ lze použít při odkazování (návěstí)
- **citlivostní seznam** obsahuje port vstup – kdykoliv je detekována změna hodnoty portu vstup, je proces vykonán
  v citlivostním seznamu zapisujeme seznam všech vstupů, signálů, sdílených proměnných, při změně kteréhokoli je proces spuštěn
- vlastní (sekvenční) příkazy vykonávané procesem jsou mezi **begin – end**
- ukázkový proces – pouze kopíruje hodnotu vstup na vystup, kdykoliv se vstup změní – buffer

FPGA, VHDL – základy jazyka VHDL

- **operátory v jazyce VHDL** – 4 druhy
  - logické
  - relační
  - aritmetické
  - ostatní

- **logické operátory**
  - elementární logické funkce a operace Booleovy algebry – na rozdíl od ní však:
  - **NOT** – má ve VHDL nejvyšší prioritu
  - **AND, OR, NAND, NOR, XOR, XNOR** – mají všechny stejnou prioritu
  - ve VHDL má logický součin i součet stejnou prioritu!
  - pokud ve VHDL zapisujeme výraz obsahující různé log. operace – musíme je oddělit pomocí závorek ()
  - příklad: a **OR** (b **AND** c) = $a \vee (b \cdot c)$

- **relační operátory**
  - **=** – rovno, **/=** – nerovno – použitelné pro všechny datové typy
  - **>** – větší než, **<** – menší než, **>=** – větší nebo rovno, **<=** – menší nebo rovno – tyto operátory jsou definovány jen pro číselné typy
  - výsledek relačních operátorů je vždy true–false
  - vhodné při podmínkových konstrukcích
  - pokud chceme v podmínce použít více relačních operátorů, oddělíme je pomocí závorek () a použijeme potřebnou operaci (AND, OR, XOR…)

FPGA, VHDL – základy jazyka VHDL

- **operátory v jazyce VHDL** – 4 druhy
  - logické
  - relační
  - aritmetické
  - ostatní

- **aritmetické operátory**
  - pro základní aritmetické operace, definovány jen pro číselné datové typy
  - ****** – umocňování, **abs** – absolutní hodnota – operátory s nejvyšší prioritou
  - ***** – násobení, **/** – dělení, **mod** – modulus, **rem** – remainder – druhá priorita
  - **+** – sčítání, **-** – odečítání – nejnižší priorita
  - priorita – opět pomocí závorek ()
  - logické obvody, logická pole – nejsou primárně určena pro aritmetické operace, různé HW limity a omezení u různých výrobců a řad FPGA, zejména operacím dělení, umocňování apod. se snažíme vyhnout (nahradit odečítáním, sčítáním apod.)

- **ostatní operátory**
  - **&** – operátor zřetězení (concatenation), můžeme zřetězit nevektorové objekty ve vektory, nebo vektory mezi sebou, např. std_logic do std_logic_vector, char do string atd. – nejde o logický součin AND!
  - operátory posuvu: **sll** – shift left logical, **srl** – shift right logical, **sla** – shift left arithmetic, **sra** – shift right arithmetic, **rol** – rotate left, **ror** – rotate right – pro bitové vektory

- **atributy v jazyce VHDL**
  - datový objekt má svou hodnotu, ale kromě toho můžeme zjistit další specifické vlastnosti a podrobnosti o něm
  - ve VHDL definována řada atributů – některé univerzální, většina jen pro konkrétní typ
  - syntaxe atributu: ***název objektu'atribut***
  - vybrané nejčastěji používané atributy:
    - **'event** – detekce změny (eventu) signálu, portu… používané zejména v sekvenčních logických obvodech pro detekci vzestupné a sestupné hrany (viz dále)
    - **'left, 'right** – hodnota na levém a pravém okraji vektoru (MSB, LSB)
    - **'low, 'high** – nejmenší a největší hodnota z množiny (např. pole)
    - **'length** – počet prvků objektu (např. délka vektoru)
    - **'val(x)** – hodnota objektu na pozici x (např. ve vektoru)
    - **'pos(x)** – číselné vyjádření pozice x v objektu (např. ve vektoru)
    - **'active** – hodnota true-false jestli se během tohoto cyklu změnila hodnota objektu
    - a velká spousta dalších…