1 Stručný popis jazyku VHDL

Jazyk VHDL *(**V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage)* je spolu s jazykem Verilog HDL jedním z nejpoužívanějším jazykům pro popis hardwarových struktur hradlových polí. Je definován standardem IEEE 1076 z roku 1987 (revidován v roce 1993 a v roce 1997 rozšíření na VHDL-AMS (Analogue & Mixed Signals)). Umožňuje návrh jak logických tak i sekvenčních struktur a jeho hlavní výhodou je jeho univerzálnost. Konečná implementace navržené struktury je závislá až na kompilaci VHDL kódu. Tzn., že pomocí tohoto jazyka lze provádět návrhy pro hradlová pole většiny výrobců (Xiling, Altera, Lattice apod.) a následně použít vhodný kompilátor, jehož volná verze je obvykle dostupná na internetových stránkách výrobců.

Návrh struktury je možné (a vhodné) rozdělit na samostatné bloky, které jsou následně spojeny ve finálním modulu pomocí objektu typu signál, který většinou odpovídá reálnému elektrickému signálu. Tento postup je v podstatě adekvátní propojování skutečných obvodů v reálu.

Jazyk VHDL má dvě povinné komponenty, které jsou nezbytné v každém modulu.

1.1 Komponenta Entity

Tato komponenta definuje vstupní a výstupní signály daného modulu, typ signálů, případně šířky sběrnic, lze si ji představit jako rozhraní modulu a okolí. Je-li modul pouze součástí rozsáhlejšího návrhu, slouží jeho výstupní signály (Porty – viz. níže) k propojení s nadřazeným modulem (ekvivalence propojování integrovaných obvodů při klasické konstrukci). Pokud je tento modul nejvyšším v hierarchii, tzn. zapouzdřuje celý návrh, jsou jeho porty fyzicky připojeny na vývody hradlového pole.

Syntaxe:

```
ENTITY název_entity IS
PORT(
  jméno_signálu1   :mód datový_typ;
  jméno_signálu2   :mód datový_typ
  );
END název_entity;
```

1.1.1 Porty (brány)

Porty popisují vnější signály Entity, jsou tedy jakýmsi rozhraním, které umožňuje propojení modulu s okolím. Tyto signály jsou charakterizovány:

Jménem – což je libovolná skupina písmen, čísel, případně podtržítka

Módem – mód určuje směr toku dat a může nabývat pěti typů:

IN – data z portu lze pouze číst

OUT – data vycházejí z portu (výstupní port nelze použít jako signál uvnitř entity)

BUFFER – obousměrný tok dat, ale v každém okamžiku může být aktivní pouze jeden signál (výstup se zpětnou vazbou, lze ho užít uvnitř entity)

INOUT – obousměrný tok dat, ale může být více aktivních signálů najednou

LINKAGE – neznámý směr datového toku

Datovým typem – lze spojovat pouze porty stejného typu.

Standard IEEE 1164 definuje multihodnotovou logiku **std_ulogic**:

„U“ – neinicializováno

„X“ – zesílená neznámá

„0“ - zesílená 0

„1“ – zesílená 1

„Z“ – vysoká impedance

„W“ – oslabená neznámá

„L“ – oslabená 0

„H“ – oslabená 1

„-“ – neurčená hodnota

Hodnoty U,X,Z,W,L,H je nutné psát v kódu velkými písmeny.

Syntaxe:

Port (jméno_signálu : datový typ);

Příklad:

```
Port (
  a,b       : in std_logic;  -- vstupy
  vstup     : in std_logic_vector(7 downto 0);  -- 8bitová vstupní sběrnice
  vystup    : out std_logic(7 downto 0);
  );
```

1.1.2 Položka Generic

Tato položka v deklaraci entity je obdobou portů (případně konstanty), ale nepředstavuje žádný elektrický signál. Používá se pouze jako parametr (nemění se v čase) a slouží pro lepší čitelnost a zlepšuje univerzálnost modulu při jeho dalším použití. Umožňuje například parametricky definovat šířku sběrnice apod.

Syntaxe:

Generic (generic_jméno : datový_typ [:= hodnota]);

Příklad:

Generic (Delay : integer := 5);

1.2 Komponenta Architecture

Tato povinná komponenta popisuje chování Entity (vnitřní strukturu (funkci) obvodu). Má dvě části: deklarační, která slouží pro deklaraci signálů, proměnných apod. a část příkazovou s popisem vlastní logické funkce obvodu.

Syntaxe:

ARCHITECTURE název_architektury **OF** název_entity **IS**

```
  signál        název_signálu       : datový_typ;
  variable      název_proměnné      : datový_typ;
  constant      název_konstanty     : datový_typ := hodnota;
BEGIN
-- popis funkce modulu
END;
```

1.2.1 Datové objekty a typy

Datové objekty :

Signals (signály) – obvykle jsou fyzicky přítomné ve formě elektrických signálů, lze je připojovat na vstupní a výstupní signály entity, pokud jsou stejných datových typů.

Pro připojení signálů se používá symbol „<=“.

Variables (proměnné) – používají se jako pomocné objekty (nepředstavují skutečné signály, nelze je připojovat na porty jako signály a jsou lokální v procesech).

Pro změnu hodnot proměnných slouží symbol „:=“

Constants (konstanty) – mají neměnnou hodnotu.

Files (soubory) – obsahují data určitého typu a používají se např. jako vstupy a výstupy při simulaci.

Datovým objektům jsou přiřazována identifikátory (jména) obdobně jako u portů. Je-li takovýto objekt deklarován v architektuře entity, je potom viditelný ve všech příkazech této architektury. Pokud je ovšem deklarován v těle procesu je použitelný pouze zde a v ostatních částech kódu je přístupný.

Datové typy :
Datové typy se dělí do dvou skupin, a to na skalární *(Scalar)* a složené *(Composite)*.
Mezi skalární patří:
Integer – celočíselná hodnota, standardně $\pm 2^{31} - 1$
Real – reálná čísla z rozsahu ± 1.0E38
Enumerated – výčtový typ
Složené datové typy:
Array – skupina elementů stejného datového typu sdružená do jednoho objektu.
Record – skupina elementů různých datových typů.

1.3 Operátory a jejich priorita

Operátory jsou v jazyce VHDL v zásadě čtyřech druhů: logické, relační, aritmetické a operátory posuvu a rotace. Pomocí vztahů definovaných těmito operátory jsou v kódu popisovány vztahy a logické funkce mezi proměnnými a signály, čímž vlastně popisujeme budoucí hardwarovou strukturu.

logické operátory: NOT, AND, OR, NAND, NOR, XOR, XNOR
aritmetické operátory: +, -, *, /, mod, abs, ** (exponent)
relační operátory: =, /=, >, <, >=, <=
operátory posuvu a rotace:
SLL (Shift Left Logical)

SRL (Shift Right Logical)
SLA (Shift Left Aritmetical)
SRA (Shift Right Aritmetical)
ROL (Rotate Left Logical)
ROR (Rotate Right Logical)

Priorita operátorů je uvedena v Tab. 1. Dalším speciálním prvkem je operátor *Slučující (concatenation) „&“,* s jehož pomocí lze slučovat datové typy stejného typu, ale nestejné velikosti. Jeho funkce je patrná z příkladu.

```
sb1, sb2        :in std_logic_vector(1 downto 0);
sb_out          :out std_logic_vector(2 downto 0);
sb_out <= sb1 & sb2;
```

V příkladu jsou dvě dvoubitové vstupní sběrnice připojeny na výstupní čtyřbitovou tak, že sb1 je na horních dvou bitech sb_out a sb2 na dolních.

Priorita	Operátory
1 (nejvyšší)	**, ABS, NOT
2	*, /, MOD
3	+, -, &
4	SLL, SRL, SLA, SRA ROL, ROR
5	=, /=, <, >, <=, >=
6	AND, OR, NAND, NOR, XOR, XNOR

Tab. 1 - Priorita operátorů

1.4 Přiřazování signálů

1.4.1 Jednoduché přiřazení signálu

Tento způsob je nejjednodušší a přiřazuje signálu na levé straně výrazu výsledek funkce pravé strany.

Příklad: Q <= not (A or B) and C;

1.4.2 Výběrové přiřazení signálu (selected)

Tento způsob přiřazování signálu nemá prioritní charakter, jedná se o obdobu příkazu CASE – WHEN.

Syntaxe:

WITH výběrový_signál **SELECT**

```
 jméno_signálu   <= hodnota_1 WHEN hodnota_1_výběrového signálu,
                 <= hodnota_2 WHEN hodnota_2_výběrového signálu,
                 <= hodnota_3 WHEN hodnota_3_výběrového signálu;
```

Příklad:

```
WITH sel SELECT
 out1            <= i0 WHEN “0000” TO “0100”,   -- výběr „od“ „do“
                 <= i1 WHEN “0101” | “0111”,    -- „nebo“
                 <= i2 WHEN “0101”,
                 <= i3 WHEN OTHERS;             -- ostatní hodnoty
```

1.4.3 Podmíněné přiřazování signálu (conditional)

Třetím způsobem přiřazování signálů v jazyce VHDL je podmíněné přiřazování, které má prioritní charakter a může při kompilaci vést na složitější obvod.

Syntaxe:

```
jméno_signálu   <=   hodnota_1 WHEN podmínka_1 ELSE
                     hodnota_2 WHEN podmínka_2 ELSE
                     hodnota_3 WHEN podmínka_3 ELSE
                     hodnota_4;
```

Příklad:

```
hd1             <=   i0    WHEN w='0' ELSE
                     i1    WHEN x='1' ELSE
                     i2    WHEN z='1' ELSE '0';
```

1.5 Popis sekvenční logiky

Sekvenční logikou jsou realizovány operace u kterých stav na výstupu není závislý pouze na současném stavu vstupů jako je tomu u kombinační logiky u které vystačíme s popisem logickými rovnicemi. Stav na výstupu je zde ovlivňován i předešlým stavem vstupů (případně výstupů). Z toho plyne, že sekvenční logika by měla být, až na vyjímky, synchronizována hodinovým (případně jiným signálem).

1.5.1 Process

Proces je základním stavebním prvkem při popisu sekvenční struktury. Procesů může být v kódu libovolný počet a v podstatě jde o samostatné části kódu, které jsou aktivní pouze v okamžicích kdy dojde ke změně „citlivých proměnných“, které jsou v procesu takto definovány. Při změně ostatních („necitlivých proměnných“) se proces neaktivuje.

Syntaxe:

```
jméno_procesu:   PROCESS (signály) -- seznam citlivých signálů (clk apod.)
  -- deklarace proměnných a signálů
BEGIN
  -- tělo proces
END;
```

Proměnné deklarované v těle procesu jsou dostupné a viditelné pouze v něm, nikoli v celém modulu a jejich hodnoty zůstávají neměnné mezi jednotlivými aktivováními procesu.

1.5.2 Příkaz IF – THEN – ELSE

Jedná o sekvenční příkaz, který lze užívat pouze v rámci procesu. Tzn., že ho nelze užít v sekci kombinační logiky. Má charakter prioritního přiřazení a může tudíž vést na složitější obvodové řešení po kompilaci.

Syntaxe:

1) **IF** podmínka **THEN** příkaz **END IF;**
2) **IF** podmínka **THEN** příkaz_1 **ELSE** příkaz_2 **END IF;**
3) **IF** podmínka_1 **THEN** příkaz_1
   ELSIF podmínka_2 **THEN** příkaz_2
   ELSIF podmínka_3 **THEN** příkaz_3 **END IF;**

Při použití třetí možnosti definice takovéto podmínky je u složitějších konstrukcí vhodnější použít příkaz **CASE – WHEN**. Kompilátor dokáže takto napsaný kód lépe a úsporněji přeložit.

1.5.3 Příkaz CASE

Podmínky pro použití tohoto příkazu jsou totožné jako u příkazu IF – THEN, rozdíl je ve výsledné hardwarové implementaci po kompilaci.

Syntaxe:

```
CASE výraz IS
    WHEN hodnota_výrazu_1   => příkaz_1;
    WHEN hodnota_výrazu_2   => příkaz_2;
    WHEN hodnota_výrazu_3   => příkaz_3;
    WHEN OTHERS             => příkaz_4;
END CASE;
```

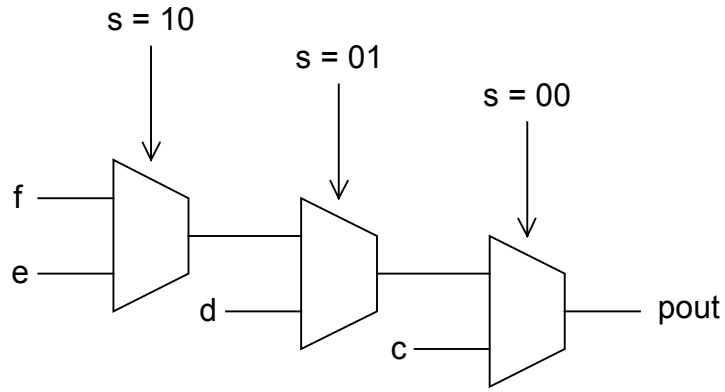
1.5.4 Rozdíl v implementaci příkazů IF-THEN a CASE-WHEN

V této kapitole jsou uvedeny dva kódy popisující totožnou funkci, jeden s použitím příkazu IF-THEN a druhý s využitím CASE-WHEN a schématické provedení implementace těchto kódů v obvodu (viz. Obr. 1 a Obr. 2).

Příklad 1 – IF-THEN

```
entity my_if is
port (
    c, d, e, f      : in std_logic;
    s               : in std_logic_vector(1 downto 0);
    pout            : out std_logic
    );
end my_if;

architecture my_arc of my_if is
begin
myif_pro: process (s, c, d, e, f)
begin
if s = "00" then pout <= c;
            elsif s = "01" then pout <= d;
                        elsif s = "10" then pout <= e;
else pout <= f;
            end if;
end process myif_pro;
end my_arc;
```

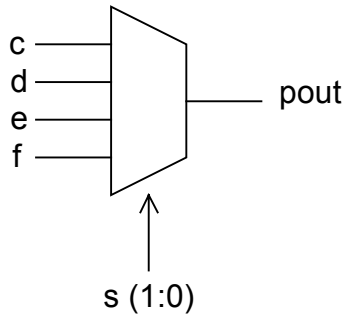


Obr. 1 - Implementace příkazu IF-THEN

Příklad 2 – CASE – WHEN

```
entity my_case is
port (
    c, d, e, f      : in std_logic;
    s               : in std_logic_vector(1 downto 0);
    pout            : out std_logic
);
end my_case;

architecture my_arc of my_case is
begin
mycase_pro: process (s, c, d, e, f)
begin
case s is
    when “00” => pout <= c;
    when “01” => pout <= d;
    when “10” => pout <= e;
    when others => pout <= f;
  end case;
end process mycase_pro;
end my_arc;
```



Obr. 2 - Implementace příkazu CASE-WHEN

1.5.5 Příkaz WAIT

Dalším příkazem skupiny sekvenčních příkazů je příkaz WAIT. Použitím tohoto příkazu zajistíme pozastavení běhu dokud nenastane definovaná podmínka a používá se pro spouštění procesu či procedury.

Syntaxe:

WAIT (**ON** seznam_signálů) (**UNTIL** výraz**)** (**FOR** čas);

Příklady:

```
WAIT ON  s1,s2;              -- čeká na změnu jednoho ze signálů
WAIT FOR 50ns;               -- zpoždění 50ns
WAIT UNTIL enable = '1';     -- zastav dokud je signál enable v log. jedničce
```

Podmínky lze v příkazu kombinovat:

WAIT ON a,b **UNTIL** clk='1';

1.5.6 Předdefinované atributy

Atributy poskytují informace o svých nositelích, kterými mohou být typy, signály, vektory, pole apod.

```
'EVENT          -- pravdivé, pokud událost právě nastala
'LAST_VALUE     -- hodnota prvku před změnou
'LOW  'HIGH     -- nejnižší (nejvyšší) index pole
'LEFT 'RIGHT    -- levá (pravá) hranice indexů
'LENGTH         -- počet prvků pole
```

1.5.7 Příkazy pro vytváření smyček – WHILE, FOR a EXIT

Opět se jedná o instrukce, které lze používat pouze uvnitř procesů a na základě typu a nastavených podmínek umožňují opakované provádění sekvence příkazů.

Syntaxe:
návěští:**WHILE** podmínka **LOOP**
sekvence příkazů
END LOOP návěští;

návěští:**FOR** parametr **IN** rozsah **LOOP**
sekvence příkazů
END LOOP návěští;

Příkaz EXIT umožňuje předčasné opuštění smyčky na základě nějaké události (například příchodu signálu).

1.5.8 Další možnosti VHDL

Jazyk VHDL umožňuje ještě několik zajímavých funkcí, které zde nebudu podrobně popisovat (syntaxe, příklady) a to z důvodu, že nebyli v této práci použity.

První z nich je možnost používat ***procedury*** obdobným způsobem jako u vyšších programovacích jazyků. Procedura je v podstatě podprogram, který modifikuje vstupní parametry. Syntetizována je při každém volání a jako vstupní parametry lze použít signály, konstanty i proměnné. Přirozeně nevrací žádnou hodnotu.

Dalším zajímavým prvkem je možnost používání ***funkcí***. Co se týče syntézy, je prováděna obdobně jako u procedur, tzn. při každém volání funkce. Na rozdíl o nich však vrací hodnotu (datový typ). Ve funkcích nemohou být, na rozdíl od procedur, deklarovány signály, ale pouze proměnné a konstanty, jejichž platnost je omezena právě jen funkcí. Uplatnění nalezne zejména při matematických výpočtech, kdy používáme především proměnné a nepoužíváme porty.

Příkaz ***generate*** slouží k popisu pravidelných hardwarových struktur, kdy používáme určitý blok kódu mnohonásobně. Například smyčkou FOR-IN můžeme vytvořit potřebné množství požadovaných prvků.

1.6 Stavové automaty a jejich popis ve VHDL

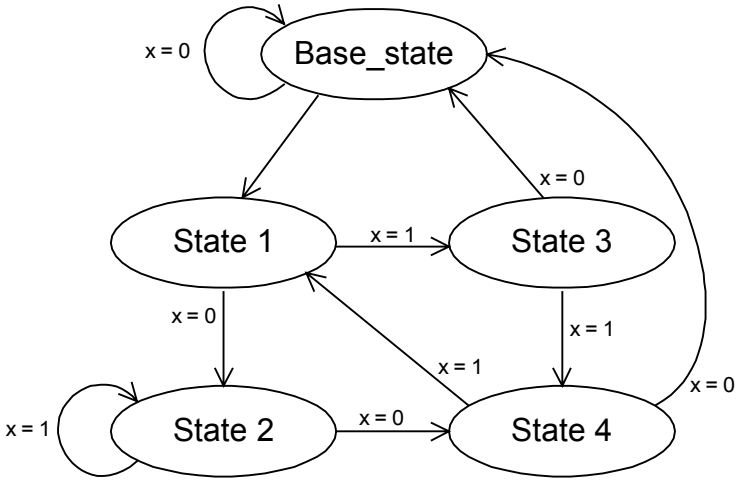
1.6.1 Stavový automat, model Moore a Mealy

Stavový konečný automat je synchronní model (zařízení), které v přesně definovaných okamžicích (obvykle s náběžnou hranou hodinového signálu) přechází mezi jednotlivými definovanými stavy. Tyto přechody nejsou náhodné, ale opisují přechodový diagram stavového automatu. To znamená, že další stav, do kterého automat přejde při příchodu další náběžné hrany, závisí jednak na současném stavu automatu a jednak na stavových signálech. Stavové signály umožňují větvení diagramu do v podstatě libovolné struktury.

Diagram stavového automatu má jeden základní stav (v příkladu na Obr. 3 je označen jako **Base_state**), to toho stavu přechází automat po resetu. Tento stav je nezbytností jinak by automat „nevěděl kde má začít“. Pro každý stav jsou definovány hodnoty výstupních vektorů (signálů), bez nichž by činnost automatu postrádala smysl, protože by navenek neprojevoval žádnou činnost (v příkladu nejsou naznačeny).

V uvedeném příkladu má diagram pět stavů a je jednu jednorozměrnou stavovou veličinu **x**. Její hodnota rozhoduje o tom, který stav bude následovat v čase **t + 1**.

Stavové automaty jsou dvou druhů: Moorův a Mealyho. Hlavní a jediný rozdíl spočívá v okamžicích změn stavů výstupních vektorů. Moorův automat mění stav již při změně stavové veličiny, kdežto u Mealyho modelu dochází k promítnutí stavu automatu na výstupní veličinu až po synchronizaci hodinovým signálem.



Obr. 3 - Příklad stavového digramu

1.6.2 Popis stavových automatů v jazyce VHDL

Stavové automaty je nejvýhodnější popisovat třemi nezávislými stavy. První proces zachycuje náběžnou hranu hodinového signálu hodnotu stavu. Druhý proces slouží pro určení následujícího stavu na základě stavu současného a hodnotě stavové veličiny. Poslední proces definuje výstupní hodnoty automatu podle stavu kde se automat právě nachází. Níže uvedený příklad popisuje stavový automat z Obr. 3.

```
Entity Moor IS
PORT(
  clk, rst, x  :in std_logic;
  z            :out std_logic_vector (1 downto 0);
  );

ARCHITECTURE Ar_Moor OF Moor IS
TYPE       states IS (base_state,state1,state2,state3,state4); --výčet stavů automatu
SIGNAL state      : states;
SIGNAL next_state         : states;
```

```
BEGIN
clkd:        PROCESS (clk, rst)          -- první proces pro zachycení stavu
BEGIN
IF (rst = '0') THEN        state <= base_state;
  ELSIF (clk'EVENT AND clk = '1') THEN
      State <= next_state;
  END IF;
END Process clkd;

state_trans: PROCESS (state, x)  -- druhý proces pro určení následujícího stavu
BEGIN
CASE state IS
  WHEN   base_state     => IF (x = '1') THEN next_state <= State1;
                           ELSE next_state <= Base_state;
  WHEN   state1         => IF (x = '1') THEN next_state <= State3;
                           ELSE next_state <= State2;
  WHEN   state2         => IF (x = '1') THEN next_state <= State2;
                           ELSE next_state <= State4;
  WHEN   state3         => IF (x = '1') THEN next_state <= State4;
                           ELSE next_state <= Base_state;
  WHEN   state4         => IF (x = '1') THEN next_state <= State1;
                           ELSE next_state <= Base_state;
END PROCESS state_trans;

output: PROCESS (state)          -- třetí proces sloužící k definici výstupu
BEGIN
CASE state IS
  WHEN base_state       => z <= "00";
  WHEN state1     => z <= "01";
  WHEN state2     => z <= "11";
  WHEN state3     => z <= "01";
  WHEN state4     => z <= "10";
END PROCESS output;
```