Použití jazyka VHDL pro návrh číslicových obvodů

Předchozí díl volného pokračování seriálu seznámil čtenáře s kroky návrhu obvodu na programovatelných hradlových polích spolu se softwarovými nástroji. V tomto příspěvku bude představena syntetizovatelná množina jazyka VHDL. Vzhledem k bohatosti jazykových konstrukcí budou prezentovány pouze jednoduché logické bloky, z nichž lze sestavit složitější obvody a z problematiky VHDL je prezentováno jen to nejpodstatnější. Spíše se v textu snažíme upozornit čtenáře na některé začátečnické chyby při návrhu. Vybrané jazykové konstrukce a jejich použití jsou demonstrovány pomocí jednoduchého návrhu na programovatelném hradlovém poli. Předpokládáme, že čtenář je již seznámen s některým programovacím jazykem, v ideálním případě s jazykem Pascal, jenž je VHDL v určitých ohledech podobný.

POUŽITÉ ZKRATKY A ZKRATKOVÁ SLOVA

AD	*Analogově-Digitální*
ALU	*Arithmetic-Logic Unit* (aritmeticko-logická jednotka)
CS	*Citlivostní Seznam*
FSM	*Finite State Machine*
KI	*Kompletní Implementace*
MUX	*MUltipleXer*
OE	*Output Enable*
RTL	*Register Transfer Level*
v/v	vstupně/výstupní
VHDL	VHSIC *Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
VS	*Verifikační Simulace*
WE	*Write Enable*

Úvod

Jazyk VHDL je specializovaný jazyk určený pro popis číslicových integrovaných obvodů. Jazyk byl původně vyvinut v US Department of Defense v rámci vládního programu USA VHSIC s cílem vytvořit prostředek pro dokumentaci a popis chování zákaznických integrovaných obvodů [1] a jednotnou platformu pro simulaci hardwaru nezávislou na technologii [1 a 2]. Filozofie VHDL vychází z jazyka ADA a specifikace jazyka byla přijata jako standard IEEE 1076-1987. Ten byl již několikrát přepracován a rozšířen, poslední revize pochází z roku 2007. Jazyk je navržen tak, aby podporoval všechny úrovně abstrakce běžně používané pro návrh: umožňuje popsat obvod na hradlové, RTL i algoritmické úrovni. Popis lze snadno simulovat a ladit pomocí standardních nástrojů. Použití VHDL jako vstupu pro syntézu bylo logickým důsledkem popsaného vývoje. Syntetizovat ovšem lze jen omezenou množinu jazykových konstrukcí, a pouze tou se budeme zabývat.

Specifikace obvodu

Základní konstrukce potřebné pro návrh číslicového obvodu a jejich užití si ukážeme na příkladu logického obvodu provádějícího kompenzaci stejnosměrného posuvu (ofsetu) hodnoty změřené AD převodníkem. Obvod pracuje v těchto krocích:

- načtení hodnoty ofsetu ze vstupní sběrnice *offset*; všechny zpracovávané hodnoty jsou osmibitové, ve dvojkovém doplňku;
- načtení změřené hodnoty ze vstupní sběrnice *AD*, která je připojená k výstupu AD převodníku;
- kompenzace ofsetu provedená podle následujícího vztahu: *kompenzovaná hodnota=výstup ADC – offset,* kde odečtení je realizováno přičtením *not(offset)+1*;
- odeslání kompenzované hodnoty na primární výstup bloku.

Blokové schéma navrhovaného obvodu je uvedeno na *obr. 1*. Vidíme, že obvod je dekomponován do datové cesty (blok ALU spolu s multiplexerem vstupních signálů) a řadičem (blok CTRL). Poznamenejme zde, že struktura a implementace navrhovaného obvodu je podřízena didaktickým účelům, a byl-li by takový systém realizován jako skutečný funkční blok pro praktické použití, byl by navržen poněkud optimálnějším způsobem a i jeho specifikace by musela být detailnější.

K indikaci platných dat z ADC na primárním vstupu obvodu a synchronizaci práce našeho bloku s okolním světem je užit *handshake* protokol realizovaný pomocí signálů *dready* a *done* (*obr. 2*). Náběžná hrana signálu *dready* indikuje přítomnost platných dat z AD převodníku na primárním vstupu, náběžná hrana signálu *done* naopak ukončení výpočtu kompenzace a přítomnost platných dat na výstupu ukázkového obvodu. Data z AD převodníku nechť zůstanou platná na vstupu obvodu po celou dobu, kdy je *dready = 1*. Externí signál *dready* je asynchronní vůči hodinám našeho bloku, a bude jej tedy nutné synchronizovat do lokální hodinové domény.

Ukázkový příklad byl syntetizován pomocí ISE WebPack a simulován v simulátoru ModelSim.

Aritmeticko-logická jednotka

Navržená ALU je do jisté míry univerzální a podporuje širší spektrum operací, než vyžaduje specifikace obvodu. V *tab. 1* je uveden seznam všech operací, které je schopna vykonávat, a dále na *obr. 3* je rozkresleno její schéma na úrovni RTL. ALU obsahuje dva datové registry – akumulátor A, do kterého se ukládají všechny výsledky operací a zároveň slouží jako první operand, a pomocný registr B, který obsahuje druhý operand. Všimněte si v *tab. 1*, že jednotlivé

Tab. 1 Seznam operací implementovaných v ALU[1]

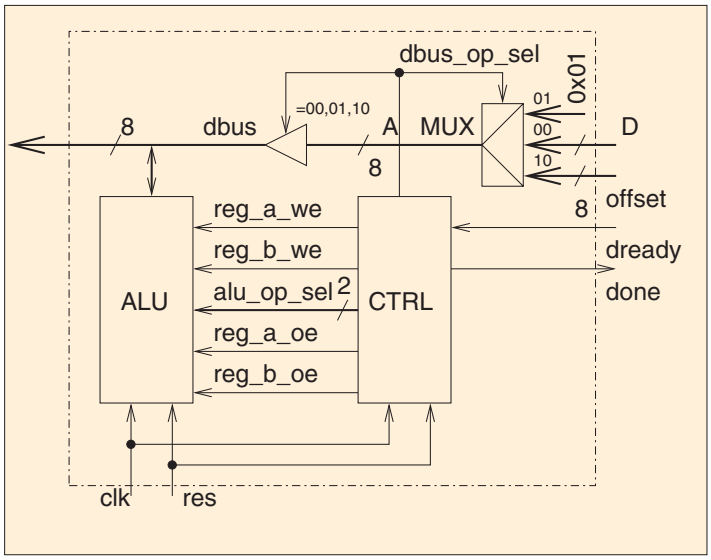
Operace	*reg_a_we*	*reg_b_we*	*alu_op_sel*	*reg_a_oe*	*reb_b_oe*
zápis do reg. A z *dbus*	**01**	0	00	0	0
zápis do reg. B z *dbus*	00	**1**	00	0	0
A = A + B	**10**	0	**01**	0	0
A = not A	**10**	0	**00**	0	0
A = A or B	**10**	0	**11**	0	0
A = A and B	**10**	0	**10**	0	0
čtení A přes *dbus*	00	0	00	**1**	0
čtení B	00	0	00	0	**1**
žádna op.	**00**	**0**	00	0	0
zápis do reg. B z *dbus*					
A = not A	**10**	**1**	**00**	0	0

[1] tučné hodnoty 0,1 jsou pro příslušnou operaci povinné, hodnoty napsané normálním typem písma jsou doporučené pro „udržení klidového stavu“ ve zbývající části jednotky, případně pro zamezení konfliktu na sběrnici *dbus*

Tab. 2 Pořadí kroků prováděných řadičem CTRL[1]

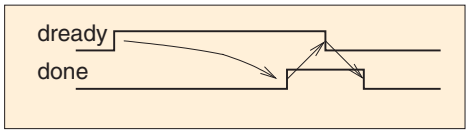
Stav	Příští stav řadiče	ALU operace	dbus_op_sel	done	Krok
RESET	pokud dready = 1, pak OFSET				
	jinak RESET	žádná	00	0	1.1
OFSET	INVERT	zápis A	10	0	1.2
INVERT	SECTI1	A = not A			
		zápis 0x01 do B	01	0	1.3
SECTI1	A_DO_B	A = A + B	00	0	1.4
A_DO_B	ADC	čtení A			
		zápis B	11	0	1.5
ADC	SECTI2	zápis ADC do A	00	0	2
SECTI2	HOTOVO	A = A + B	11	0	3
HOTOVO	pokud dready = 1, pak HOTOVO				
	jinak RESET	čtení A	11	1	4

[1] ve sloupci Krok je číslo kroku výpočtu, jak je popsán v sekci Specifikace; Krok 1 je prováděn více stavy označenými 1.1–1.5



Obr. 1 Blokové schéma navrhovaného obvodu

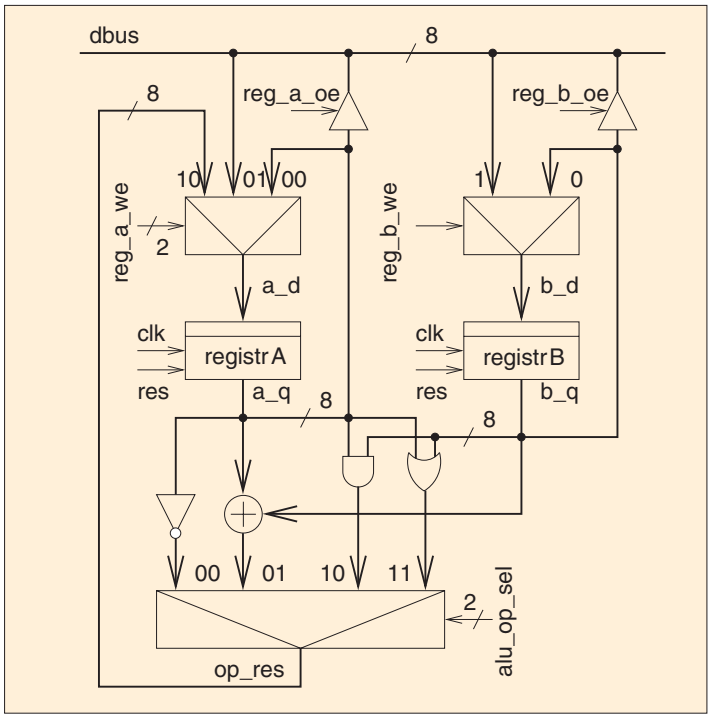
operace obvykle nevyžadují definované hodnoty na všech řídicích a datových vstupech. Některé operace tak lze provádět současně



Obr. 2 Handshake protokol pro předávání dat mezi blokem a okolní logikou (šipky označují pořadí jednotlivých událostí)

například načtení hodnoty do registru B a současně negaci hodnoty v registru A (viz příslušný řádek *tab. 1*).

Jednotka je navržena pro práci s osmibitovými čísly, není ale problém ji upravit tak, aby mohla pracovat s libovolnou (generickou) šířkou slova – viz např. návrh hradla AND v [3], příp. [4], fólie 17 a [2].



Obr. 3 Schéma ALU na úrovni RTL (jména signálů a struktura přímo odpovídají zdrojovému kódu)

Řadič

Řadič obvodu je zodpovědný za řízení datové cesty (ALU) a komunikaci s okolím obvodu. V našem případě je implementovaný jako jednoduchý stavový automat. Sekvence prováděných kroků je zachycena v *tab. 2*.

```
001   LIBRARY IEEE;
002   USE IEEE.std_logic_1164.ALL;
003   USE IEEE.std_logic_unsigned.ALL;
004
005   ENTITY alu IS
006     PORT (
007       res         : IN std_logic;
008       clk         : IN std_logic;
009       dbus        : INOUT std_logic_vector (7 DOWNTO 0);
010       reg_a_oe    : IN std_logic;
011       reg_b_oe    : IN std_logic;
012       reg_a_we    : IN std_logic_vector (1 DOWNTO 0);
013       reg_b_we    : IN std_logic;
014       alu_op_sel : IN std_logic_vector (1 DOWNTO 0)
015     );
016   END ENTITY alu;
017
018   ARCHITECTURE rtl OF alu IS
019     SIGNAL a_d    : std_logic_vector (7 DOWNTO 0);
020     SIGNAL a_q    : std_logic_vector (7 DOWNTO 0);
021     SIGNAL b_d    : std_logic_vector (7 DOWNTO 0);
022     SIGNAL b_q    : std_logic_vector (7 DOWNTO 0);
023     SIGNAL op_res : std_logic_vector (7 DOWNTO 0);
024   BEGIN
025
026     mux_ab : PROCESS (op_res, reg_a_we, reg_b_we, dbus,
a_q, b_q)
027     BEGIN
028       IF    reg_a_we = "10" THEN
029         a_d <= op_res;
030       ELSIF reg_a_we = "01" THEN
031         a_d <= dbus;
032       ELSIF reg_a_we = "00" THEN
033         a_d <= a_q;
034       ELSE --11 nebo jina kombinace
035         a_d <= a_q;
036       END IF;
037       IF reg_b_we = '1' THEN
038         b_d <= dbus;
039       ELSE
040         b_d <= b_q;
041       END IF;
042     END PROCESS mux_ab;
043
044     reg_ab : PROCESS (clk, res)
045     BEGIN
046       IF res = '1' THEN
047         a_q <= (OTHERS => '0');
048         b_q <= (OTHERS => '0');
049       ELSIF clk'EVENT AND clk = '1' THEN
050         a_q <= a_d;
051         b_q <= b_d;
052       END IF;
053     END PROCESS reg_ab;
054
055     mux_op_res : PROCESS (a_q, b_q, alu_op_sel)
056     BEGIN
057       CASE alu_op_sel IS
058         WHEN "00"   =>
059           op_res <= NOT(a_q);
060         WHEN "01"   =>
061           op_res <= a_q + b_q;
062         WHEN "10"   =>
063           op_res <= a_q AND b_q;
064         WHEN "11"   =>
065           op_res <= a_q OR b_q;
066         WHEN OTHERS =>
067           op_res <= (OTHERS => 'X');
068       END CASE;
069     END PROCESS mux_op_res;
070
071     dbus <= a_q WHEN reg_a_oe = '1' ELSE "ZZZZZZZZ";
072     dbus <= b_q WHEN reg_b_oe = '1' ELSE "ZZZZZZZZ";
073
074   END ARCHITECTURE rtl;
```

Obr. 4 Výpis souboru ALU, soubor alu.vhd

```
001   LIBRARY IEEE;
002   USE IEEE.std_logic_1164.ALL;
003   USE IEEE.std_logic_unsigned.ALL;
004
005   ENTITY top_level IS
006     PORT (
007       clk         : IN  std_logic;
008       res         : IN  std_logic;
009
010       dready      : IN  std_logic;
011       done        : OUT std_logic;
012
013       dbus        : INOUT std_logic_vector (7 DOWNTO 0);
014       offset      : IN    std_logic_vector (7 DOWNTO 0);
015       ad          : IN    std_logic_vector (7 DOWNTO 0)
016     );
017   END ENTITY top_level;
018
019   ARCHITECTURE rtl OF top_level IS
020     SIGNAL reg_a_oe_i    : std_logic;
021     SIGNAL reg_b_oe_i    : std_logic;
022     SIGNAL reg_a_we_i    : std_logic_vector (1 DOWNTO 0);
023     SIGNAL reg_b_we_i    : std_logic;
024     SIGNAL alu_op_sel_i  : std_logic_vector (1 DOWNTO 0);
025     SIGNAL dbus_op_sel_i : std_logic_vector (1 DOWNTO 0);
026     SIGNAL dbus_i        : std_logic_vector (7 DOWNTO 0);
027
028     COMPONENT alu IS
029       PORT (
030         res         : IN std_logic;
031         clk         : IN std_logic;
032        dbus        : INOUT std_logic_vector (7 DOWNTO 0);
033         reg_a_oe    : IN std_logic;
034         reg_b_oe    : IN std_logic;
035         reg_a_we    : IN std_logic_vector (1 DOWNTO 0);
036         reg_b_we    : IN std_logic;
037         alu_op_sel : IN std_logic_vector (1 DOWNTO 0)
038       );
039     END COMPONENT alu;
040
041     COMPONENT ctrl IS
042       PORT (
043         clk         : IN  std_logic;
044         res         : IN  std_logic;
045
046         dready      : IN  std_logic;
047         done        : OUT std_logic;
048
049         reg_a_oe    : OUT std_logic;
050         reg_b_oe    : OUT std_logic;
051         reg_a_we    : OUT std_logic_vector (1 DOWNTO 0);
052         reg_b_we    : OUT std_logic;
053         alu_op_sel : OUT std_logic_vector (1 DOWNTO 0);
054         dbus_op_sel: OUT std_logic_vector (1 DOWNTO 0)
055       );
056     END COMPONENT ctrl;
057
058   BEGIN
059
060     i_alu : alu
061       PORT MAP (
062         res         => res,
063         clk         => clk,
064         dbus        => dbus_i,
065         reg_a_oe    => reg_a_oe_i,
066         reg_b_oe    => reg_b_oe_i,
067         reg_a_we    => reg_a_we_i,
068         reg_b_we    => reg_b_we_i,
069         alu_op_sel => alu_op_sel_i
070       );
071
072     i_ctrl : ctrl
073       PORT MAP (
074         clk         => clk,
075         res         => res,
076
077         dready      => dready,
078         done        => done,
079
080         reg_a_oe    => reg_a_oe_i,
081         reg_b_oe    => reg_b_oe_i,
082         reg_a_we    => reg_a_we_i,
083         reg_b_we    => reg_b_we_i,
084         alu_op_sel => alu_op_sel_i,
085         dbus_op_sel=> dbus_op_sel_i
086       );
087
088     dbus_i <= "00000001" WHEN dbus_op_sel_i = "01" ELSE
089               ad         WHEN dbus_op_sel_i = "00" ELSE
090               offset     WHEN dbus_op_sel_i = "10" ELSE
091               "ZZZZZZZZ"; --bez buzeni
092
093     dbus   <= dbus_i;
094   END rtl;
```

Obr. 5 Blok na nejvyšší úrovni hierarchie, soubor top_level.vhd.

Logické úrovně

Od jazyka určeného k návrhu a modelování integrovaných obvodů očekáváme schopnost modelovat základní logické úrovně – log. 1 a log. 0. Ty jsou pro jednoduché logické simulace postačující, pro profesionální práci však potřebujeme mít možnost pracovat i se stavem vysoké impedance Z pro návrh a modelování třístavových budičů sběrnic. Dále je užitečné mít možnost modelovat zkrat na sběrnici – situaci, kdy dva budiče budí jeden spoj opačnými logickými úrovněmi. Knihovny VHDL pro tento účel zavádějí log. úroveň X. Navíc lze pracovat s úrovněmi H, L (slabá log. 1 a 0 generovaná slabým budičem, např. výstupem s otevřeným kolektorem a *pull-up* rezistorem), W, (tzv. *don't care* – libovolná logická úroveň) a U (neinicializovaná hodnota). Podpora pro devítistavovou logiku je implementována v balíčku *std_logic_1164* v knihovně IEEE. Všimněte si, že knihovna je připojena všemi ukázkovými kódy, např. řádky 1 a 2 na *obr. 4*. Logický signál, který má popsaných hodnot nabývat, je pak definován jako signál typu *std_logic* nebo *std_logic_vector*, pokud se jedná o sběrnici. Pomocí nich modelujeme standardní propojení logických prvků – je to (velmi lapidárně řečeno) „obyčejný kus drátu".

Popis rozhraní bloku

První krok, který je při vlastním návrhu číslicové logiky nutné udělat, je popis rozhraní navrhovaného bloku. V jazyce VHDL je pro každý blok definováno rozhraní ve formě popisu entity (řádky 5–16 v *obr. 4*). V tomto jednoduchém případě vidíme v deklaraci entity seznam v/v signálů uvozený klíčovým slovem *PORT*. Pro každý ze signálů je uvedeno jeho jméno (např. *clk*), směr vzhledem k bloku (*IN* – vstup, *OUT* – výstup, *INOUT* – obousměrný signál) a typ – buď *std_logic*, nebo *std_logic_vector*. U sběrnice je nutné definovat šířku a rozsah číslování bitů, což je provedeno zápisem (*7 DOWNTO 0*). Sběrnice má pak 8 bitů číslovaných 7, 6, 5, 4, 3, 2, 1, 0. V identifikátorech nejsou rozlišována malá a velká písmena.

Každá entita má přiřazenou svou implementaci – „vnitřek bloku" – architekturu. K jedné entitě může být přiřazeno i více architektur najednou (např. implementací na různých úrovních abstrakce). V našem případě máme ke každé entitě jen jednu architekturu, jejich spárování je provedeno v definici architektury (řádek 18 v *obr. 4*). Použitý zápis definuje architekturu s názvem *rtl* implementující entitu ALU. Dále v kódu následuje deklarativní část architektury, kde jsou deklarovány signály a komponenty (vnořené bloky) užité vlastní implementací (řádky 19–23 na *obr. 4*, příp. 20–57 na *obr. 5*). Signál v jazyce VHDL představuje model fyzického spoje v integrovaném obvodu. Pro přiřazení hodnoty používáme operátor <=. Změna hodnoty signálu se neprojeví okamžitě, ale nejdříve po tzv. delta zpoždění, protože ani skutečné spoje nemohou měnit svoje stavy nekonečně rychle [2]. Blok popisující implementaci začíná klíčovým slovem *BEGIN* a končí klíčovým slovem *END ARCHITECTURE* na řádku 74 na *obr. 4*.

Popisujeme-li hardware, musíme být schopni zachytit vysokou míru paralelismu, která je mu vlastní (souběžně se šířící signály po spojích, současně klopící hradla atd.). To je ve VHDL řešeno velmi elegantně, všechny

příkazy zapsané mezi *BEGIN* a *END ARCHITECTURE* se z pohledu uživatele provádějí paralelně. Příkazem zde může být jednoduché přiřazení (řádky 71 a 72 na *obr. 4*), složitější konstrukce, nebo tzv. proces (např. řádky 26–42 na *obr. 4*). Zatímco příkazy v paralelním prostředí jsou vykonávány hned, jak dojde ke změně jednoho nebo více signálů, na kterých závisí výstup příkazu, příkazy v procesu jsou prováděny v pořadí jejich zápisu [2]. Spouštění provádění procesu řídí tzv. citlivostní seznam (*sensitivity list*). Ten lze nalézt např. na řádku 26 na *obr. 4* v závorkách. CS je seznamem signálů v obvodu, jejichž změna vyvolá vykonávání procesu, tedy způsobí, že se proces provede a vyhodnotí. Poznamenejme, že ne vždy je citlivostní seznam potřebný [5]. Sestavení CS je třeba věnovat zvýšenou pozornost. Popisujeme-li kombinační logickou funkci, zapisujeme do CS všechny signály, které se vyskytují na pravé straně přiřazení v procesu, v podmínkách příkazu *IF*, jako řídicí signály u příkazu *CASE* atd. Pro sekvenční logickou funkci (viz dále popis registrů) obsahuje CS typicky hodinový signál a signály, na které má blok reagovat asynchronně (např. reset a set). Ponecháme-li například v CS na řádku 26 pouze signál *op_res*, bude se proces vyhodnocovat pouze při změně signálu *op_res* a příslušné výstupní signály nebudou během simulace reagovat na změny na ostatních signálech. Skutečný obvod se bude nicméně chovat správně, protože nástroje pro syntézu citlivostní seznamy ignorují a vytvoří kombinační logický obvod. Dostaneme se tak do situace kdy má kód na RTL úrovni funkčně odlišné chování od výsledného hardwaru. Naštěstí na tuto situaci nástroje pro syntézu samy upozorňují varovnými hlášeními v žurnálu syntézy. Jelikož procesem popisujeme odezvu obvodu na konkrétní událost a obvod modelujeme na úrovni RTL, proces na změny na vstupu reaguje „nekonečně rychle“. Proto se také změny hodnot signálů projeví až po dokončení výpočtu příslušného procesu (na konci simulačního cyklu simulace založeného na událostech po tzv. zpoždění delta). Více o procesech a paralelním prostředí lze nalézt např. v [2].

Strukturní popis

Číslicový systém je obvykle navrhován ve formě hierarchického propojení bloků. Jazyk VHDL proto potřebuje kompletní podporu pro zachycení takové struktury. Na *obr. 1* a *5* lze nalézt implementaci nejvyšší

```
001     LIBRARY IEEE;
002     USE IEEE.std_logic_1164.ALL;
003     USE IEEE.std_logic_unsigned.ALL;
004
005     ENTITY ctrl IS
006       PORT (
007         res         : IN  std_logic;
008         clk         : IN  std_logic;
009
010         dready      : IN  std_logic;
011         done        : OUT std_logic;
012
013         reg_a_oe    : OUT std_logic;
014         reg_b_oe    : OUT std_logic;
015         reg_a_we    : OUT std_logic_vector (1 DOWNTO 0);
016         reg_b_we    : OUT std_logic;
017         alu_op_sel  : OUT std_logic_vector (1 DOWNTO 0);
018         dbus_op_sel: OUT std_logic_vector (1 DOWNTO 0)
019       );
020     END ENTITY ctrl;
021
022     ARCHITECTURE rtl OF ctrl IS
023       TYPE tstate IS (RESET, OFSET, INVERT, SECTI1, A_DO_B,
ADC, SECTI2, HOTOVO);
024       SIGNAL curr_state  : tstate;
025       SIGNAL next_state  : tstate;
026       SIGNAL dready_sync : std_logic_vector (1 DOWNTO 0);
027     BEGIN
028
029       state_reg : PROCESS (res, clk)
030       BEGIN
031         IF res = '1' THEN
032           curr_state <= RESET;
033         ELSIF clk = '1' AND clk'EVENT THEN
034           curr_state <= next_state;
035         END IF;
036       END PROCESS state_reg;
037
038       dready_sync_reg : PROCESS (res, clk)
039       BEGIN
040         IF res = '1' THEN
041           dready_sync <= "00";
042         ELSIF clk = '1' AND clk'EVENT THEN
043           dready_sync <= dready_sync(0) & dready;
044         END IF;
045       END PROCESS dready_sync_reg;
046
047       output_logic : PROCESS (curr_state)
048       BEGIN
049         done        <= '0';
050         reg_a_oe    <= '0';
051         reg_b_oe    <= '0';
052         reg_a_we    <= "00";
053         reg_b_we    <= '0';
054         alu_op_sel  <= "00";
055         dbus_op_sel <= "11";
056         CASE curr_state IS
057           WHEN RESET  =>
058           WHEN OFSET  =>
059             reg_a_we    <= "01";
060             dbus_op_sel <= "10";
061           WHEN INVERT =>
062             reg_a_we    <= "10";
063             alu_op_sel  <= "00";
064             reg_b_we    <= '1';
065             dbus_op_sel <= "01";
066           WHEN SECTI1 =>
067             reg_a_we    <= "10";
068             alu_op_sel  <= "01";
069           WHEN A_DO_B =>
070             reg_b_we    <= '1';
071             reg_a_oe    <= '1';
072           WHEN ADC    =>
073             reg_a_we    <= "01";
074             dbus_op_sel <= "00";
075           WHEN SECTI2 =>
076             reg_a_we    <= "10";
077             alu_op_sel  <= "01";
078           WHEN HOTOVO =>
079             done        <= '1';
080             reg_a_oe    <= '1';
081         END CASE;
082       END PROCESS output_logic;
083
084       next_state_logic : PROCESS (dready_sync, curr_state)
085       BEGIN
086         next_state <= curr_state;
087         CASE curr_state IS
088           WHEN RESET  =>
089             IF dready_sync(1) = '1' THEN
090               next_state <= OFSET;
091             END IF;
092           WHEN OFSET  =>
093             next_state <= INVERT;
094           WHEN INVERT =>
095             next_state <= SECTI1;
096           WHEN SECTI1 =>
097             next_state <= A_DO_B;
098           WHEN A_DO_B =>
099             next_state <= ADC;
100           WHEN ADC    =>
101             next_state <= SECTI2;
102           WHEN SECTI2 =>
103             next_state <= HOTOVO;
104           WHEN HOTOVO =>
105             IF dready_sync(1) = '0' THEN
106               next_state <= RESET;
107             END IF;
108         END CASE;
109       END PROCESS next_state_logic;
110
111     END ARCHITECTURE rtl;
```

Obr. 6 Řadič datové cesty, soubor ctrl.vhd

úrovně hierarchie (*top-level*) ukázkového příkladu. Blok na nejvyšší úrovni hierarchie zapouzdřuje jednotlivé podbloky, popisuje jejich propojení a jeho rozhraní jsou primárními vstupy/výstupy systému.

Deklarace komponent na řádcích 28–56 říkají simulátoru, jaké bloky budou použity. Vlastní deklarace získáme jednoduchým zkopírováním entit příslušných bloků a nahrazením klíčového slova *ENTITY* slovem *COMPONENT*. Na řádcích 60–86 jsou potom komponenty použity (instancovány). Klauzule *PORT MAP* popisuje propojení skutečných portů entit s vnitřními signály bloku, např. řádek 64 uvádí, že port *dbus* komponenty ALU je zapojen na signál *dbus_i* v bloku *top_level*.

Bývá dobrým zvykem na nejvyšší úrovni hierarchie pouze instancovat komponenty a neimplementovat žádnou logickou funkci. Z důvodů jednoduchosti jsme zde zařadili i multiplexer vstupních signálů.

Třístavový budič

Jedním z nejjednodušších prvků je třístavový budič sběrnice. Dva takové budiče obsahuje i ALU (*obr. 3*). Budiče jsou řízeny signály *reg_a_oe* a *reg_b_oe* a umožňují číst obsahy registrů A a B. Jejich syntetizovatelný popis je uveden na *obr. 4* na řádcích 71–72. Funkce je dostatečně zřejmá z uvedeného kódu. Pokud např. *reg_a_oe = 0*, je výstup příslušného budiče ve stavu vysoké impedance, a na všechny bity sběrnice dbus je tak přivedena hodnota Z.

Základní logické funkce

Pro podporu implementace základních logických funkcí nabízí VHDL celou škálu operátorů (AND, OR, NOT, XOR, NAND atd.). V našem obvodu jsme chtěli mít v ALU možnost realizovat inverzi akumulátoru a logický součet/součin akumulátoru s obsahem registru B. Syntetizovatelná realizace logických operací je zřejmá z popisu (řádky 59, 63 a 65 na *obr. 4*). Poznamenejme zde, že logické operátory definované v balíčku *std_logic_1164* nad typem *std_logic* implementují kompletní sadu operací s devítistavovou logikou, např. 0 AND X = 0, *1 AND X = X* atd. [2].

Multiplexer

Ve zdrojových kódech ukázkového obvodu jsou použity tři základní způsoby, jak implementovat multiplexer. Nejjednodušší způsob lze nalézt v *obr. 5* na řádcích 88–91. Použili jsme zde paralelní konstrukci *WHEN-ELSE*. Kromě multiplexeru uvedený zápis realizuje i třístavový budič zapojený na jeho výstupu (alternativa pro *dbus_op_sel_i = 11*).

Druhou možností je zápis pomocí klauzule *IF-THEN-ELSE* použité v procesu (řádky 28–41 na *obr. 4*, které implementují vstupní multiplexery registrů A a B v ALU). Zde bychom rádi čtenáře upozornili na jedno z úskalí používání VHDL. Začátečníkovi se může lehce stát, že z nepozornosti vynechá větev ELSE na řádcích 34–36 či 39–41. Výsledný kód tak může vypadat například takto:

```
IF reg_b_we = '1' THEN
  b_d <= dbus;
END IF;
```

Na chybu se simulacemi vůbec nemusí přijít, i když došlo k velmi závažné změně chování obvodu. Z původně kombinační logické funkce se stala funkce sekvenční. Pokud je totiž *reg_b_we = 0*, hodnota na sběrnici *b_d* zůstává zachována. Získali jsme tak hladinový klopný obvod. Naštěstí nástroj pro syntézu v ISE na uvedenou konstrukci upozorňuje prostřednictvím varování „*found latch*“ v žurnálu.

Třetí možností je užití konstrukce *CASE-WHEN* ekvivalentní stejnojmenné konstrukci jazyka Pascal. Použití příkazu *CASE* vidíme v procesu na řádcích 55–69 na *obr. 4*. V příkazu si všimněte povinné alternativy *WHEN OTHERS*. Jazyk VHDL si vynucuje definování reakce na všechny alternativy, které mohou nastat. Jelikož pracujeme s devítistavovou logikou, může signál *alu_op_sel* nabývat i hodnot XX, UU a mnohých dalších. Abychom je nemuseli všechny vypisovat, použijeme konstrukci WHEN OTHERS. Blok kódu za ní uvedený je vykonán, pokud není vykonána žádná z předchozích alternativ, tedy pokud *alu_op_sel* není 00, 01, 10 ani 11.

Aritmetické operace

Při implementaci aritmetických operací návrhář zpravidla nejvíce ocení výhody návrhu na úrovni RTL. Jazyk VHDL spolu se standardizovanými knihovnami umožňuje velmi elegantní implementaci (sčítačka v ALU, řádek 61 na *obr. 4*). Podobně jednoduchá by byla i implementace násobení, jen místo operátoru „+“ bychom použili operátor „*“. Implementace aritmetických operací však má i svá úskalí, o kterých se zmíníme v jednom z dalších příspěvků.

Obecná kombinační funkce

Nástroje pro syntézu jsou schopny korektně rozpoznat a vygenerovat i obvody pro implementaci významně složitějších logických funkcí, než jsme si doposud ukázali. S kombinačními logickými funkcemi často pracujeme ve formě pravdivostní tabulky či booleovských rovnic, během fáze specifikace obvodu ovšem funkce často popisujeme na behaviorální úrovni například ve formě implikací „jestliže ..., pak ...“. Jazyk VHDL sice umožňuje zápis kombinační funkce pomocí booleovských rovnic, ale tato forma se používá jen zřídka. Častěji se lze setkat s popisem funkce pomocí tabulky , tedy vlastně s implementací logické funkce pamětí ROM. Bezesporu nejoblíbenější je třetí možnost implementace obecné logické funkce – popis na úrovni chování. Využijeme přitom prezentovaných konstrukcí, které lze volně kombinovat (řádky 26–42 na *obr. 4*, či složitější příklad na řádcích 47–82 na *obr. 6*). Zde užíváme zápisu pomocí procesů.

Paměťový prvek

Základním typem paměťového elementu je jednoduchý klopný obvod typu D. Jeho VHDL popis lze nalézt například na řádcích 44–53 na *obr. 4*. Implementujeme zde dva osmibitové registry s asynchronním resetem (řádek 46, 47 a 48). Registr reaguje na náběžnou hranu hodin; to lze vyčíst z podmínky na řádku 49. Podmínka *clk'event* je splněna právě tehdy, když na hodinách došlo k události (výskyt hrany) a druhá část podmínky *clk = 1* vybere náběžnou hranu (po hraně je clk v log. 1).

Pomocí prostředků jazyka VHDL lze na FPGA jednoduše navrhovat i složitější struktury, jako jsou např. synchronní a asynchronní paměti [6].

Stavový automat

Řadič je implementován jako stavový automat (FSM). Implementace FSM již nevyžaduje žádné další jazykové konstrukce, neboť je prostým spojením dvou kombinačních logických funkcí realizujících výpočet příštího stavu, aktuálního výstupu a stavového registru.

Podíváme-li se blíže na zdrojový kód řadiče (*obr. 6*), rozpoznáme zde všechny tyto tři základní struktury realizované procesy *state_reg*, *output_logic* a *next_state_logic*. Dále zde nalezneme proces *dready_sync_reg*, který zajišťuje korektní resynchronizaci vstupního asynchronního signálu *dready*. Všimněme si také způsobu, jakým je implementován stav FSM (signály *curr_state* a *next_state*) a stavový registr automatu. Nepoužili jsme zde signály typu *std_logic_vector*, ale vlastní výčtový typ *tstate*, který nabývá hodnot pojmenovaných jako jednotlivé stavy automatu (řádek 23 na *obr. 6*). Oproti ručnímu návrhu automatu nám jazyk VHDL opět ulehčuje práci. Jednak nemusíme ručně minimalizovat dílčí logické funkce, a dále za nás syntéza provede i automatické zakódování stavů automatu. Použití výčtového typu významně zjednodušuje i simulaci a ladění – v časových průbězích přímo vidíme zvolená symbolická jména stavů.

Náměty k zamyšlení

Pro zájemce o hlubší pochopení prezentované problematiky jsme připravili několik jednoduchých cvičení. Odpovědi na položené otázky jsou spolu se zdrojovými kódy příkladu, kompilačními a simulačními skripty pro ModelSim a VHDL kódem simulačního prostředí s automatickou kontrolou odezev uvedeny v souboru reseni_cviceni.pdf v balíčku [7].

Cvičení 1 – analýza chování bezchybného obvodu: Spusťte verifikační simulaci (VS). Prohlédněte si časové průběhy hodnot signálů a všimněte si, jak je modelována třístavová sběrnice a funkce řídicího automatu. Spusťte kompletní implementaci v ISE WebPack (KI), jako cílový obvod zvolte xc2s15-5cs144. Jak velký je výsledný obvod? Jaká je maximální dosažitelná hodinová frekvence a kudy prochází kritická cesta v obvodu? Jak jsou kódovány stavy řídicího automatu ve skutečném obvodu?

Cvičení 2 – chybné vložení hladinového klopného obvodu: Ve zdrojovém kódu v *obr. 4* zakomentujte řádky 39 a 40 – větev ELSE příkazu IF. Spusťte VS. Odhalila VS změnu chování? Spusťte KI a prostudujte výstup z nástroje pro syntézu. Najděte varování, které upozorňuje na chybu v kódu.

Cvičení 3 – chybný citlivostní seznam: Ponechte na řádku 26 ve výpisu 1 v CS pouze signál op_res, ostatní vymažte. Spusťte VS i KI. Odhalila VS změnu v chování návrhu? Prohlédněte si také výstup z nástroje pro syntézu. Upozorňuje syntezátor na potenciální rozdíl v chování mezi návrhem na úrovni RTL a skutečným obvodem?

Závěr

Článek prezentuje některé elementární jazykové konstrukce, pomocí nichž lze vytvořit složitější číslicové systémy. Vzhledem k omezenému rozsahu příspěvku se ale na mnoho aspektů jazyka VHDL nedostalo, proto odkazujeme čtenáře k dalšímu studiu (seznam použité literatury).

Návrh číslicového systému pomocí jazyka VHDL je velmi příjemný, často ale svádí k tvorbě složitých konstrukcí a zbytečně velkých obvodů s dlouhými kombinačními cestami a vysokou spotřebou. Mějte proto vždy na paměti, že neprogramujete software, ale navrhujete integrovaný obvod. Dobrým zvykem je nejprve navrhovaný obvod nakreslit na papír a posléze během psaní kódu průběžně spouštět implementační proces a sledovat technické parametry návrhu. Umožní to případně ihned přepracovat příliš divoké jazykové konstrukce či opravit některé chyby, a tak ušetřit čas v pozdějších fázích návrhu. Cit pro návrh a zkušenosti se správným užitím jazyka lze bohužel získat pouze dlouholetou praxí. Tato práce byla podpořena výzkumným záměrem MSM6840770012 – Transdisciplinární výzkum v biomedicínském inženýrství 2.

Ing. Jakub Šťastný, Ph.D.
Katedra teorie obvodů ČVUT v Praze
ASICentrum, s. r. o.

Literatura k článku je na str. 487

Generátor ZUS22 – jednoduché řešení pro údržbu a servis

Od roku 2002 firma Dodávky automatizace dodává na český trh jednoúčelové průmyslové kalibrátory a zdroje unifikovaných signálů typové řady ZUS. Tyto přístroje jsou určeny pro servisní a provozní praxi měření a regulace (MaR) a automatizační systémy řízení technologických procesů (ASŘ TP), především pro účely nastavování měřicích a regulačních obvodů, převodníků, řídicích systémů atd. Vycházejí z jednotné koncepce, pro niž jsou typické shodná konstrukce a ovládání, stejně tak jako i shodný design. Jsou řešeny jako přenosné (kapesní) s bateriovým napájením. Prvním přístrojem z typové řady ZUS je zdroj proudu a napětí ZUS22. Je určen především pro servisní činnost systémů MaR, elektro a ASŘ TP, které pracují v rozsahu 4 až 20mADC a 0 až 10 V DC.

V základním režimu pracuje ZUS22 jako generátor proudu v aktivním a pasivním režimu s rozlišením 10 μA nebo jako generátor napětí s rozlišením 10 mV. Další možností je zadávání žádaných hodnot po 25 % z daného rozsahu (pět nastavených hodnot). Chyba generování je ve všech třech režimech generace ±0,25 % z rozsahu. Přístroj zobrazuje na displeji LCD poruchové provozní stavy, při nichž generovaná hodnota neodpovídá požadované hodnotě. K těmto poruchovým stavům patří pokles napětí baterie a zkrat na výstupu v režimu generace napětí nebo zvětšení odporu proudové smyčky v režimu generace proudu. Zatěžovací odpor musí být v režimu generace napětí větší než 10 kΩ. V režimu generace proudu musí být odpor (v ohmech) proudové smyčky naopak menší než 50 U_{NAP} – 8, kde U_{NAP} je napájecí napětí proudové smyčky. Pracovní teplota přístroje je 0 až 40 °C. Přístroj je určen pro běžnou údržbu a kontrolu nastavených parametrů.

zs

Kontakt:
Dodávky automatizace, spol. s r. o.
1. máje 34/120
706 02 Ostrava-Vítkovice
tel.: 596 600 111, fax: 596 600 116
e-mail: mikroelektronika@daas.cz
www.daas.cz

Nový v/v modul společnosti Siemens

Společnost Siemens rozšiřuje nabídku jednotek vzdálených vstupů, resp. výstupů řady Simatic ET200S o nový modul s vestavěnou procesorovou jednotkou a komunikačním rozhraním Profinet, který nese označení IM151-8 PN/DP CPU. Modul zvládá řídicí úlohy i komunikaci po síti Ethernet a výkonově odpovídá centrále Simatic S7-300 CPU 314. Přístroj bude k dispozici také v provedení pro bezpečnostní řízení (až SIL3, resp. Cat 4), v této verzi bude mít typové označení IM151-8 PN/DP F-CPU.

Nový modul IM151-8 PN/DP CPU je plnohodnotnou procesorovou jednotkou, kterou lze rozšířit až o 63 vstupně/výstupních modulů řady ET200S. V novém modulu s vestavěným CPU je integrován přepínač

(*switch*), který nabízí tři porty pro napojení do komunikační sítě Ethernet. Stanice ET200S osazená modulem IM151-8 PN/DP CPU může komunikovat po síti Ethernet pomocí protokolů Profinet IO, Profinet IRT, TCP/IP, UDP/IP a ISO-on-TCP. V první fázi bude nový modul fungovat jako Profinet IO-controller (řídí až 128 podřazených stanic typu Profinet IO-device), ve druhé fázi bude následovat doplnění funkce Profinet IO-device (podřazená stanice na síti Ethernet). Samozřejmostí je i podpora S7-komunikace a rozhraní Profinet CBA. Nová procesorová jednotka navíc obsahuje integrovaný web-server, jenž slouží zejména pro účely diagnostiky, monitorování proměnných apod.

js

Kontakt: Siemens, s. r. o.
Evropská 33a, 160 00 Praha 6
tel.: 800 122 552
fax: 233 032 492
e-mail: adprodej.cz@siemens.com
www.siemens.cz/ad

Literatura k článku na str. 440–445:

LITERATURA

[1] Internet: *http://en.wikipedia.org/wiki/VHDL* [kontrolováno 21. 6. 2008]

[2] SPIEGEL, J., VHDL Tutorial. *www.seas.upenn.edu/~ese201/vhdl/vhdl_primer.html* [kontrolováno 21. 6. 2008]

[3] Jednoduché logické bloky – ukázky VHDL kódů. *http://amber.feld.cvut.cz/fpga/prednasky/fpga_navrh/fpga_navrh.html* [kontrolováno 21. 6. 2008]

[4] ŠŤASTNÝ, J., Úvod do syntetizovatelného VHDL – prezentace z přednášky. *http://amber.feld.cvut.cz/fpga/prednasky/fpga_navrh/fpga_navrh.html* [kontrolováno 21. 6. 2008]

[5] Hamburg VHDL Tutorial. *www.vhdl-online.de/tutorial* [kontrolováno 21. 6. 2008]

[6] KOLOUCH, J., Možnosti realizace pamětí RAM v obvodech FPGA. *Elektrorevue* 2003, č. 27.

[7] Ukázkový příklad návrhu. *http://amber.feld.cvut.cz/fpga/prednasky/FPGA_navrh/fpga_navrh.html* [kontrolováno 21. 6. 2008]

[8] CHU, P. P., *RTL Hardware Design Usign VHDL*. Coding for Efficiency, Portability and Scalability, John Wiley & sons, 2006.

[9] KŘEMEČEK, M., *Implementace základních logických funkcí na FPGA. Bakalářská práce.* Praha : FEL ČVUT, KTO, FPGA Laboratoř, 2006. *http://amber.feld.cvut.cz/fpga/publications/BP_Kremecek_2006.pdf* [kontrolováno 21. 6. 2008]