Podstawy VHDL

(aktualizacja: 2021-08-09)

VHDL jest jednym z najpopularniejszych języków opis sprzętu. Służy do opisu (na różnych poziomach abstrakcji) cyfrowego układu elektronicznego, celem symulacji jego działania oraz automatycznej syntezy rzeczywistego układu (w postaci schematu złożonego z układów dyskretnych, projektu układu scalonego, czy też wsadu do układów programowalnych FPGA itp). Najbardziej abstrakcyjne poziomy opisu mają problemy z syntezowalnością, dlatego jeżeli myślimy o wykorzystaniu VHDL do syntezy najbezpieczniej jest operować poziomem przesłań między-rejestrowych (register transfer level). Ponadto należy się stosować do pewnych szablonów tak aby narzędzia syntezy bezproblemowo interpretowały kod. Istnieją także HDL dla układów analogowych oraz kilka innych języków HDL dla układów cyfrowych (np. Verilog, AHDL).

Trochę innym podejściem jest SystemC, będący w istocie biblioteką dla C++ umożliwiającą opisywanie (i symulowanie tak zbudowanych systemów) sprzętu w ramach kodu C++. Kod taki może być automatycznie konwertowany na VHDL i z niego może być dokonywana synteza.

własny pakiet

Tworzenie własnego pakietu, deklaracje funkcji i procedur, konwersja pomiędzy liczbami całkowitymi a wektorami bitów:

```
-- uwidaczniamy bibliotekę IEEE
library ieee;

-- używamy z tej biblioteki wszystkich elementów z pakietu std_logic_1164
-- pozwala to na pisanie std_logic zamiast ieee.std_logic_1164.std_logic
use ieee.std_logic_1164.all;
-- oraz z pakietu numeric_std
use ieee.numeric_std.all;


-- deklarujemy nasz własny pakiet
package moj_pakiet is
        -- w ramach niego będzie funkcja to_bits()
        function to_bits(liczba, dlugosc_wektora : in integer) return  std_logic_vector;
        -- oraz to_int()
        function to_int(wartosc_bitowa : in std_logic_vector) return integer;
        -- i procedura zwieksz
        -- procedury (w odróżnieniu od funkcji) mogą modyfikować przekazane do nich argumenty, co zobaczymy dalej
        procedure zwieksz(dane: inout std_logic_vector; ovf: out std_logic);
end package;

-- definiujemy zawartość naszego pakietu
package body moj_pakiet is
        function to_bits(liczba, dlugosc_wektora : in integer) return  std_logic_vector is
                begin
                        return std_logic_vector(to_unsigned(liczba,dlugosc_wektora));
        end function;

        function to_int(wartosc_bitowa : in std_logic_vector) return integer is
                begin
                        return to_integer(unsigned(wartosc_bitowa));
        end function;

        procedure zwieksz(dane: inout std_logic_vector; ovf: out std_logic) is
                begin
                        -- pętla for po całym zakresie binarnym wektora wejściowego/wyjściowego
                        -- to coś po ' nazywa się atrybutem, na przykład:
                        --  * xx'range zwraca zakres xx'left to xx'right
                        --  * xx'right jest maksymalnym indeksem wektora
                        F1: for i in dane'range loop
                                -- przypisanie do zmiennej
                                dane(i) := not dane(i);
                                -- przerwanie petli (tak jak break w C)
                                exit when dane(i) = '1';
                                -- pomiędzy exit a when można podać etykietę pętli z której chcemy wyjść ...

                                -- jeżeli pętla doszła do końca
                                -- nie osiągając warunku przerwania to mamy przepełnienie
                                if i = dane'right then
                                        ovf := '1';
                                else
                                        -- galąź else jest obowiązkowa gdy nie chcemy tworzyć zatrzasków
                                        ovf := '0';
                                end if;
                        end loop;
        end procedure;
end;
```

multiplekser na kilka sposobów

Deklaracja prostego komponentu na przykładzie multipleksera, korzystanie z parametrów ogólnych, kierunki portów:

```
library ieee;
use IEEE.STD_LOGIC_1164.all;

-- deklarujemy multiplexer 1 bitowy o M wejściach
entity multiplexer is
	-- parametr ogólny, modyfikuje zachowanie jednostki projektowej
	-- na etapie jej umieszczania w projekcie (coś jak #define w C)
	generic (
		M: natural := 2
	);
	-- porty wejściowe i wyjściowe jednostki
	-- przy ich pomocy komunikujemy się z innymi elementami systemu
	-- kierunek portu oprócz zaprezentowanego in i out może być:
	--  * inout - port dwukierunkowy (jak GPIO w AVR)
	--  * buffer - port wyjściowy z którego możemy odczytać wpisana do niego
	--             wartość (przerzutnik wyjściowy),
	--             tryb ten może stwarzać problemy z kompatybinością
	port (
		-- tryb jest liczba całkowita z zakresu 0-M
		MODE: in integer range 0 to M-1;
		-- wejścia zgrupowane sa w postaci wektora M-1 bitowego
		DATA_IN: in std_logic_vector (M-1 downto 0);
		-- wyjściem jest pojedynczy bit
		DATA_OUT: out std_logic
	);
end;

-- definiujemy architekturę naszego multipleksera
architecture logic of multiplexer is
	begin
		DATA_OUT <= DATA_IN(MODE);
end;
```

Wykorzystanie utworzonej jednostki w tworzeniu innej, zaprezentowanie rożnych instrukcji wyboru (innych pomysłów na multiplekser):

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

-- używamy naszego pakietu
use work.moj_pakiet.all;


-- deklarujemy multiplexer 3 wejściowy
entity mux_3to1 is
	port (
		-- tryb jest wartością 2 bitowa
		MODE: in std_logic_vector (1 downto 0);
		-- mamy 3 wejścia
		a,b,c: in std_logic;
		-- i jedno wyjście
		DATA_OUT: out std_logic
	);
end;


-- definiujemy architekturę naszego multipleksera
architecture z_N_bitowego_arch of mux_3to1 is
	-- gdy chcemy wykorzystać jakiś element (w tym wypadku multiplexer
	-- opisany w multiplexer.vhdl) musimy przytoczyć
	-- (lekko zmodyfikowana) treść jego entity w postaci opisu komponentu:
	component mux is
		generic (
			M: natural := 4
			-- nadpisujemy domyślna ilość bitów
		);
		port (
			MODE: in integer range 0 to M-1;
			DATA_IN: in std_logic_vector (M-1 downto 0);
			DATA_OUT: out std_logic
		);
	end component;

	-- deklarujemy wewnętrzne pomocnicze sygnały
	signal tryb: integer range 0 to 2;
	signal wejscie: std_logic_vector (3 downto 0);

	begin
		wejscie <= '-' & c & b & a;
		-- musimy podać tez najwyższy bit (aby jego bitowość była 2^N)
		-- jako ze nie jest on nam potrzebny a korzystamy z std_logic to podajemy "don't cary"

		-- przy pomocy funkcji to_int z moj_pakiet konwertujemy wartość binarna na integer
		tryb<=to_int(MODE);

		-- mapujemy odpowiednie wejścia i wyjścia dla egzemplarza komponentu multiplexer
		-- określonego etykieta mux_31
```

```
		mux_31: mux port map (MODE=>tryb, DATA_IN=>wejscie, DATA_OUT=>DATA_OUT);
		-- etykiety możemy podać w każdej linii vhdl, ale tylko w niektórych wypadkach jest
		-- to wymagane (właśnie taka sytuacja ma tutaj miejsce)
end;


-- można także okreslić konfiguracje komponentów używanych do budowy architektury
configuration z_N_bitowego_conf of mux_3to1 is
	for z_N_bitowego_arch -- dla architektury "z_N_bitowego_arch"
		for mux_31: mux -- w mux_31 jako mux
			use entity work.multiplexer(logic); --- używamy entity multiplexer z architekturą logic
							    --- (są zdefiniowane w multiplexer.vhdl)
		end for;
		-- tu mógłby być kolejny for dla kolejnych komponentów
	end for;
	-- tu mógłby być kolejny for dla kolejnej architektury
end;
-- gdyby w architekturze zamiast mux podać nazwę komponentu zgodna z jego entity
-- to blok ten byłby niepotrzebny bo użyta byłaby konfiguracja domyślna:


architecture z_N_bitowego of mux_3to1 is
	-- gdy chcemy wykorzystać jakiś element (w tym wypadku multiplexer
	-- opisany w multiplexer.vhdl) musimy przytoczyć
	-- (lekko zmodyfikowana) treść jego entity w postaci opisu komponentu:
	component multiplexer is
		generic (
			M: natural := 4
			-- nadpisujemy domyslna ilosc bitow
		);
		port (
			MODE: in integer range 0 to M-1;
			DATA_IN: in std_logic_vector (M-1 downto 0);
			DATA_OUT: out std_logic
		);
	end component;

	-- deklarujemy wewnętrzne pomocnicze sygnały
	signal tryb: integer range 0 to 2;
	signal wejscie: std_logic_vector (3 downto 0);

	begin
		wejscie <= '-' & c & b & a;
		-- musimy podać tez najwyższy bit (aby jego bitowość była 2^N)
		-- jako ze nie jest on nam potrzebny a korzystamy z std_logic to podajemy "don't cary"

		-- przy pomocy funkcji to_int z moj_pakiet konwertujemy wartość binarna na integer
		tryb<=to_int(MODE);

		-- mapujemy odpowiednie wejścia i wyjścia dla egzemplarza komponentu multiplexer
		-- określonego etykieta mux_31
		mux_31: multiplexer port map (MODE=>tryb, DATA_IN=>wejscie, DATA_OUT=>DATA_OUT);
end;



-- architektur może być kilka ...
architecture wybierany_case of mux_3to1 is
	begin
		with MODE select DATA_OUT <=
			a when "00",
			b when "01",
			c when "10",
			'-' when others;
			-- w przypadku std_logic formalnie (część kompilatorów tego nie wymaga)
			-- praktycznie zawsze konieczne jest podawanie "when others"
			-- gdyż mamy inne niż 0 i 1 wartości sygnału
end;

architecture proces_case of mux_3to1 is
	begin
		process (a,b,c,MODE)
			begin
				case MODE is
					when "00" => DATA_OUT <= a;
					when "01" => DATA_OUT <= b;
					when "10" => DATA_OUT <= c;
					when others => DATA_OUT <= '-';
				end case;
		end process;
end;

-- UWAGA ze względu na kolejność wykonywania instrukcji w poniższych konstrukcjach
-- oba układy opisane poniżej mogą się zsyntezować jako hierarchiczna struktura
-- multiplekserów 2 wejściowych
--
-- NIE JEST to zatem zalecany opis multipleksera 3 na 1
```

```
architecture wybierany_if_else of mux_3to1 is
	begin
		DATA_OUT <=
			a when MODE = "00" else
			b when MODE = "01" else
			c when MODE = "10" else
			'-';
end;

architecture proces_if_else of mux_3to1 is
	begin
		process (a,b,c,MODE)
			begin
				if MODE="00" then
					DATA_OUT <= a;
				elsif MODE="01" then
					DATA_OUT <= b;
				elsif MODE="10" then
					DATA_OUT <= c;
				else
					DATA_OUT <= '-';
				end if;
		end process;
end;
```

oczekiwanie, logika, pętle

Operacje logiczne, oczekiwanie (procesy z wait), zmienne, operacje matematyczne, pętle:

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

entity nor3 is
	port (
		wej: in std_logic_vector(0 to 2);
		wyj: out std_logic
	);
end;


architecture logic of nor3 is
	begin
		wyj <= not (wej(0) or wej(1) or wej(2));
		-- ze względu na to iż funkcjonalność XOR opisuje się w taki prosty sposób
		-- w praktycznych zastosowaniach raczej nie ma sensu robienia osobnego komponentu XOR ...
end;

-- możemy jednak użyć innych sposobów implementacji XOR
-- do zademonstrowania procesów z pętlami i oczekiwaniem:

architecture proces_while of nor3 is
	begin
		process
			variable i: integer range -1 to 3;
			variable x: std_logic;
			begin
				-- czekamy na zmianę któregoś z sygnałów
				wait on wej;
				-- jest to podobne do listy podawanej po słowie proces ...
				-- tyle ze wtedy oczekiwanie jest umieszczane na końcu ciała procesu ...
				-- należy wspomnieć ze jest też wait until (czekanie na spełnienie warunku)

				x := '0';
				i := -1;

				-- petla typu while
				L0: while i<2 loop
					i := i+1;

					-- instrukcja next powoduje przejście do następnego kroku pętli
					next when wej(i) = '0';
					-- pomiędzy next a when można podać etykietę pętli której iteracje chcemy zakończyć ...

					x := '1';
				end loop;

				wyj <= not x;
		end process;
end;


architecture proces_loop of nor3 is
	begin
		process (wej)
			variable i: integer range -1 to 2;
			variable x: std_logic;
			begin
```

```
                                x := '0';
                                i := -1;
                                -- pętla prosta
                                L0: loop
                                        exit when i=2;
                                        i := i+1;
                                        next when wej(i) = '0';
                                        x := '1';
                                        exit;
                                end loop;

                                wyj <= not x;
                end process;
end;
```

użycie elementów i testbench

testbench

Przykład pisania testbench'ów, uwagi na temat syntezowalności, używanie literałów, wygodne wstawianie komponentu:

```
library ieee;
use ieee.std_logic_1164.all;
use std.env.finish;

entity tester is
end;

architecture logic of tester is
        signal CLK,RST,a,b,c: std_logic := '1';
        -- uwaga takie przypisanie wartości początkowej nie podlega syntezie (!)
        --       podobnie nie da się syntezować opóźnień czasowych,
        --       ale to jest testbanch wiec nam to nie przeszkadza
        signal tryb: std_logic_vector (1 downto 0);
        signal wej1, wej2: std_logic_vector (2 downto 0);
        -- w miejscu tym warto wspomnieć o możliwości definiowania aliasów np:
        --  alias pocz_wej1 : std_logic_vector (1 downto 0) is wej1 (1 downto 0);
        -- spowoduje że nazwa pocz_wej1 będzie się odnosiła do dwuelementowego wektora
        -- tożsamego z pierwszymi dwoma elementami wej1

        begin
                -- zegar
                process
                        begin
                        wait for 2 ns;
                        CLK <= not CLK;
                end process;

                -- zatrzymanie symulacji
                process
                        begin
                        wait for 100 ns;
                        report "koniec symulacji";
                        finish;
                        -- starszym rozwiązaniem na takie przerwanie jest:
                        --   assert FALSE report "koniec symulacji" severity FAILURE;
                        -- działa w starszych wersjach VHDL,
                        -- jednak powoduje on wypisywanie informacji o zakończeniu z błędem
                end process;

                RST <= '0', '1' after 0.5 ns;

                a <= '0' after  5 ns, '1' after  25 ns;
                b <= '0' after  9 ns, '1' after  21 ns;
                c <= '0' after 13 ns, '1' after  17 ns;

                tryb <= "00", "10" after 11 ns, "01" after 22 ns;

                mux1: entity work.mux_3to1(z_N_bitowego) port map(MODE=>tryb, a=>a, b=>b, c=>c);
                mux2: entity work.mux_3to1(wybierany_case) port map(MODE=>tryb, a=>a, b=>b, c=>c);
                mux3: entity work.mux_3to1(proces_case) port map(MODE=>tryb, a=>a, b=>b, c=>c);
                mux4: entity work.mux_3to1(wybierany_if_else) port map(MODE=>tryb, a=>a, b=>b, c=>c);
                mux5: entity work.mux_3to1(proces_if_else) port map(MODE=>tryb, a=>a, b=>b, c=>c);
                mux6: configuration work.z_N_bitowego_conf port map(MODE=>tryb, a=>a, b=>b, c=>c);

                wej1 <= a&b&c;
                wej2 <= O"5";
                -- zapis 3 bitowej wartości w notacji oktalnej ...
                -- 8 bitowy wektor w notacji szesnastkowej możemy wypełnić poprzez X"fa"
                --
                -- z kolei 2#0# oznacza ze pomiędzy # jest zapisana liczba stałopozycyjna (integer)
                -- w notacji o podstawie 2 (w systemie dwójkowym)

                nor1: entity work.nor3(logic) port map(wej=>wej1);
                nor2: entity work.nor3(proces_while) port map(wej=>wej1);
                nor3: entity work.nor3(proces_loop) port map(wej=>wej1);
```

```
                przes1: entity work.rejestr_przesuwny(logic)
                        generic map(N=>4)
                        port map(D=>a,CLK=>CLK,RST=>RST,ENABLE=>'1');
                -- należy wspomnieć że zarówno w generic jak i port map możemy  używać pozycyjnego jak i
                -- nazewniczego sposobu podawania przypisań jednak jeżeli jest ich więcej niż jedno to
                -- sposób poprzez nazwa=>... ("nazewniczy") jest czytelniejszy i wygodniejszy ...
                przes2: entity work.rejestr_przesuwny(modularna)
                        generic map(4)
                        port map(D=>a,CLK=>CLK,RST=>RST,ENABLE=>'1');
        -- na koniec warto wspomnieć o tym iż do tworzenia rozbudowanych testbench'ow można
        -- wykorzystać mechanizmy vhdl do obsługi pliku, dzięki czemu pobudzenia testbencha mogą
        -- być umieszczane w zewnętrznym pliku (np. generowanym przez jakieś inne oprogramowani)
        -- podobnie ewentualne wyniki zostaną zapisane do plików co umożliwi ich analizę
        -- zewnętrznymi narzędziami
end;
```

analiza elaboracja i uruchomienie

Proces uruchomienia symulacji z kodu VHDL składa się z kilku etapów. Celem jego łatwiejszego przeprowadzenia wykorzystać można make z następującym plikiem Makefile:

```
GHDL_OPT = --std=08

tester:
        # dodanie do biblioteki
        ghdl -i $(GHDL_OPT) *.vhdl
        # analiza/kompilacja rzeczy potrzebnych do uruchomienia jednostki (unit'a) $@
        ghdl -m $(GHDL_OPT) $@
        # uruchomienie (symulacji) jednostki (unit'a) $@
        ghdl -r $(GHDL_OPT) $@ --wave=$@.ghw
        # wyświetlenie wyniku
        gtkwave $@

clean:
        rm -f *.ghw *.cf
```

pamięć

przerzutnik, zatrzask

Podstawowy element pamiętający:

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

entity przerzutnik is
        port (
                D,CLK,RST,ENABLE: in std_logic;
                Q: buffer std_logic
        );
end;

architecture logic of przerzutnik is
        begin
                process (CLK, RST)
                begin
                        if (RST='0') then
                                Q<='0';
                        -- reagowanie na opadające (CLK='0') zbocze zegara (CLK'event)
                        -- powoduje utworzenie przerzutnika a nie zatrzasku
                        -- który uzyskalibyśmy pisząc: if (ENABLE='1') then Q<=D; end if;
                        elsif (CLK='0' and CLK'event) then
                                if (ENABLE='1') then
                                    Q<=D;
                                end if;
                        end if;
                end process;
end;
```

rejestr przesuwny (i składanie z wielu komponentów)

Budowa rejestru przesuwnego w oparciu o proces oraz w oparciu o łączenie przerzutników – pokazuje wybieranie podzestawu bitów z wektora bitowego oraz wykorzystanie generate-for oraz generate-if do złożenia wielu jednakowych podkomponentów w większą całość:

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

entity rejestr_przesuwny is
        generic (
                N: natural := 1
        );
        port (
                CLK,RST,ENABLE: in std_logic;
```

```
                D: in std_logic;
                P_OUT: buffer std_logic_vector (N-1 downto 0)
        );
end;

architecture logic of rejestr_przesuwny is
        begin
                process (CLK, RST)
                begin
                        if (RST='0') then
                                for i in 0 to N-1 loop
                                        P_OUT(i) <= '0';
                                end loop;
                                -- sprytniejszym zapisem tego jest:
                                --  P_OUT <= (others => '0');
                                -- korzystamy tu z tzw agregacji umożliwiającej wygodne nadawanie wartości wektorom np.:
                                --  wektor <= (1|3 => '1', 2|4 =>'0');
                                --  wektor <= (1 => '1', 3 => '1', 2|4 =>'0');
                                --  wektor <= ('1', '0', '1', '0');
                                --  wektor <= "1010"
                                -- są całkowicie równoważnymi zapisami, jednak pierwszy z nich jest zdecydowanie
                                -- wygodniejszy dla długich wektorów
                                -- ponadto dzięki agregacji możemy łatwo wpisać wektor do zestawu sygnałów poprzez:
                                --  (skalar, vektor_2, skalar) <= wektor_4
                        elsif (CLK='0' and CLK'event) then
                                if (ENABLE='1') then
                                        P_OUT <= D & P_OUT(N-1 downto 1);
                                        -- tak jak do pojedynczych bitów możemy odwołać się do zakresów
                                        -- o kolejności bitów z zakresu decyduje użycie to / downto
                                end if;
                        end if;
                end process;
end;

architecture modularna of rejestr_przesuwny is
        begin
                -- używamy generate for aby wykorzystać odpowiednia liczbę 1 bitowych multiplekserów
                G0: for i in 0 to N-1 generate
                        -- możemy także warunkować fragmenty tego co umieszczamy w generate
                        G01: if i=N-1 generate
                                mux: entity work.przerzutnik(logic)
                                -- powyższy zapis oszczędza podawania deklaracji komponentu
                                -- oraz ewentualnego podawania konfiguracji ...
                                        port map (D=>D,CLK=>CLK,RST=>RST,ENABLE=>ENABLE,Q=>P_OUT(i));
                        end generate;
                        G02: if i<N-1 generate
                                mux: entity work.przerzutnik(logic)
                                        port map (D=>P_OUT(i+1),CLK=>CLK,RST=>RST,ENABLE=>ENABLE,Q=>P_OUT(i));
                        end generate;
                end generate;
end;
```

automat

Przykład jawnej konstrukcji automatu:

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

entity automat is
        port (
                wejscia: in std_logic_vector (1 downto 0);
                clk, reset: in std_logic;
                wyjscia: out std_logic_vector (1 downto 0)
        );
end;

architecture logic of automat is
        type STANY is (
                S0,
                S1,
                S2
        );

        signal STAN_OBECNY, STAN_NASTEPNY: STANY := S0;

begin
        process (clk, reset)
        begin
                if (reset='1') then
                        STAN_OBECNY <= S0;
                elsif (clk='1' and clk'event) then
                        STAN_OBECNY <= STAN_NASTEPNY;
                end if;
        end process;

        process (STAN_OBECNY, wejscia)
```

```
	begin
		case STAN_OBECNY is
			when S0 =>
				wyjscia <= "01";
				-- wyjścia zależą tylko od stanu, więc automat Moora

				if (wejscia = "01") then
					STAN_NASTEPNY <= S1;
				else
					STAN_NASTEPNY <= S2;
				end if;
			when S1 =>
				wyjscia <= "11";

				if (wejscia = "01" or wejscia = "10") then
					STAN_NASTEPNY <= S2;
				elsif (wejscia = "11") then
					STAN_NASTEPNY <= S1;
				else
					STAN_NASTEPNY <= S0;
				end if;
			when S2 =>
				wyjscia <= "00";

				if (wejscia(0) = '0') then
					STAN_NASTEPNY <= S1;
				else
					STAN_NASTEPNY <= S2;
				end if;
		end case;
	end process;
end;
```

kody źródłowe

Wszystkie kody źródłowe pokazywane w tym artykule dostępne są także do pobrania w postaci archiwum tar/gzip.

licencja