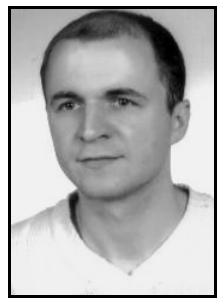
Mirosław MOŚCICKI
POLITECHNIKA SZCZECIŃSKA, WYDZIAŁ INFORMATYKI, KATEDRA TECHNIK PROGRAMOWNIA

Generowanie równań boolowskich dla instrukcji mapowania języka VHDL

Mgr inż. Mirosław MOŚCICKI

Jest absolwentem Wydziału Informatyki Politechniki Szczecińskiej (2000). Obecnie pracuje na stanowisku asystenta w Katedrze Technik Programowania. Jest specjalistą w zakresie tworzenia oraz testowania oprogramowania. Od 1997 roku uczestniczył w wielu projektach informatycznych jako programista, projektant oraz tester.



e-mail: mmoscicki@wi.ps.pl

Streszczenie

W przedstawionym opracowaniu zaprezentowany został sposób generowania równań boolowskich dla wielokrotnie powtarzających się mapowań na tą samą jednostkę. Algorytm ten opiera się na zapisie raz wygenerowanych równań dla mapowanej jednostki w odpowiednim metapliku. Dla każdej jednostki może istnieć wiele metaplików zawierających równania. Oprócz plików z równaniami tworzony jest dodatkowy plik zawierający informacje o mapowanych sygnałach jednostki. W omówionym algorytmie pełny proces generowania równań boolowskich dla takich samych argumentów odbywa się tylko raz.

Słowa kluczowe: język VHDL, FPGA, równania boolowskie, kompilatory.

Boolean Equations Generation For 'map' Instruction In VHDL Language

Abstract

In this paper is proposed and discribed a Boolean Equation generation for multiple map. The algorithm is based on writing generated equations for map entity in meta file. There is a possibility of existing for one entity many meta files with equations. If map process on the same entity appears multiple, then full Boolean equations generation process is done only once.

Keywords: VHDL language, FPGA, Boolean equations, compilers.

1. Wstęp

Od kilku lat w Katedrze Technik Programowania Wydziału Informatyki Politechniki Szczecińskiej realizowany jest projekt [3], którego celem jest stworzenie kompilatora dokonującego konwersji pliku zawierającego program napisany w języku VHDL na równania boolowskie. Równania boolowskie są bardzo dobrym materiałem wyjściowym do dalszej obróbki ponieważ jest to forma matematyczna. Na ich podstawie można stworzyć układy cyfrowe realizujące określone zadania, możemy je również poddać minimalizacji. Równania boolowskie mogą być wykorzystywane przy produkcji układów FPGA [1, 9].

Rozwiązując bardziej złożony problem wyodrębnia się zwykle pewne jego części, dla których formułuje się rozwiązania oddzielnie.. Wydzielenie poszczególnych problemów ma istotne zalety, gdyż umożliwia prowadzenie rozumowania na ustalonych poziomach abstrakcji. We wszystkich językach programowania istnieją mechanizmy umożliwiające dzielenie rozwiązywanego problemu na części. W języku VHDL możemy korzystać z podprogramów oraz komponentów. Podprogramy umożliwiają definiowanie algorytmów, które jako oddzielne moduły programu mogą reprezentować wybrany element zachowania się układu. Najczęściej występują dwa rodzaje podprogramów: procedury i funkcje.

Język VHDL służy do projektowania cyfrowych układów logicznych. Modułowa budowa większości układów cyfrowych zachęca nas do stosowania podziału projektowanego układu na komponenty. Specyfikacja komponentów jest szczegółowo opisana w wielu pozycjach. Projektowanie komponentów jest prostą czynnością i umożliwia projektantom wielokrotne wykorzystanie raz zaprojektowanego układu. Ponadto wiele firm oferuje gotowe komponenty które możemy wykorzystać w projektowanych układach.

Komponent może być zaprojektowany w sposób uniwersalny i z tego powodu może być wielokrotnie wykorzystywany w różnych układach [2]. Ze względu na złożoność zadań jakie wykonują komponenty ich rozmiar może być znaczny, dlatego też pojawia się problem szybkiego generowania równań boolowskich dla nich. W niniejszym opracowaniu zostanie omówiony problem generowania równań boolowskich dla komponentów.

2. Mechanizm mapowania

Składnia mapowania w języku VHDL jest następująca [6]:

```
Instance_name: component_name
[ generic map (
                    generic_name => expression
                    { , generic_name => expression }
) ]
port map (
          [ port_name => ] expression
          { , [ port_name => ] expression }
);
```

Podczas mapowania może występować słowo kluczowe „generic", które powoduje przypisanie wartości do odpowiedniej zmiennej zadeklarowanej w jednostce. Dla każdej deklaracji komponentu musi istnieć jednostka o tej samej nazwie. Część wiążąca składnik z jednostką projektową może zawierać także informacje o mapowaniu parametrów i portów składnika. Do tego celu służą listy mapować parametrów oraz portów. Występowanie obu list jest opcjonalne. Jeżeli sposób połączenia portów między komponentem a jednostką nie został jawnie podany, wówczas dla każdego portu składnika musi znaleźć się odpowiedni port w jednostce. Port musi mieć taką samą nazwę oraz taki sam typ, w przeciwnym razie wymagana jest jawna deklaracja sekcji mapowania portów[11]. Analogiczna zasada obowiązuje dla wartości parametrów. Przy mapowaniu możemy korzystać z notacji pozycyjnej (ang. positional notation) lub przez nazwę (ang. name notation). Proces mapowania zostanie opisany na podstawie poniższego przykładu (przykład 1):

Przykład 1. Program w języku VHDL
Example 1. VHDL language program

Jednostka:

```
entity RAM is
          generic( inout_range : integer );
          port (
                    CLK: in STD_LOGIC;
                    RST: in STD_LOGIC;
                    OUT_DATA: out STD_LOGIC_VECTOR
          (inout_range -1  downto 0)
          );
end RAM;
```

Komponent:

```
component  RAM is
          generic( inout_range : integer );
          port (
                    CLK: in STD_LOGIC;
                    RST: in STD_LOGIC;
```

```
                OUT_DATA: out STD_LOGIC_VECTOR
        (inout_range -1  downto 0)
        );
end component ;
```

Przykładowe mapowania:

```
        U_RAM_0:  RAM
        generic map (inout_range => 8)
        port map(
                CLK => CLK,  RST =>RST,
                OUT_DATA => Out_DataR0
                );
        U_RAM_1: RAM
        generic map (inout_range => 8)
        port map(
                CLK => CLK,  RST =>RST,
                OUT_DATA => Out_DataR1
                );
```

Przed przystąpieniem do generowania równań boolowskich program źródłowy w języku VHDL musi zostać poddany analizie leksykalnej [10], syntaktycznej oraz semantycznej [4, 5]. Proces kompilacji programu przebiega w kilku krokach:

- Gdy w programie występuje mapowanie to musimy sprawdzić czy istnieje komponent oraz jednostka o odpowiedniej nazwie.
- Następnie sprawdzana jest zgodność sygnałów zadeklarowanych wewnątrz komponentu i jednostki, oraz zgodność sygnałów występujących przy mapowaniu z sygnałami komponentu.
- Jeśli sygnały są zgodne to można przeprowadzić proces mapowania.

Podczas mapowania generowane są równania boolowskie dla mapowanej jednostki (przykład 2).

Przykład 2. Program w języku VHDL
Example 2. VHDL language program

```
library IEEE;
use IEEE.std_logic_1164.all;
entity test is
        port (
                x: in integer;
                y: out integer
        );
end test;

architecture test of test is
begin
  y<= x;
end test;
-------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity blask is
        port (
                x1: in integer;
                y1: inout integer
        );
end blask;

architecture test of blask is
begin
  y1<=x1;
end test;
-----------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity gtest is
        generic (s: integer:=1);
        port (
                x: in integer;
                y: out integer;
                x1: in integer;
                y1: inout integer
        );
end gtest;

architecture gtest of gtest is
component test
        port (
                x: in integer;
                y: out integer
        );
end component;

component blask
        port ( x1: in integer;
        y1: inout integer);
end component;
signal s1,s2,s3,s4 : integer;

begin
  m1: test port map(x=>s1,y=>s2);
  m2: blask port map(x1=>s2,y1=>s3);
end gtest;
```

Dla powyższego przykładu zgodnie z przedstawionym algorytmem zostaną wygenerowane następujące równania boolowskie (przykład 3):

Przykład 3. Równania boolowskie
Example 3. Boolean equations

```
--port_map equations begin, line: 62, file: Test_0013.vhd
s2(31)=s1(31);
s2(30)=s1(30);
s2(29)=s1(29);
s2(28)=s1(28);
...
...
...
s2(3)=s1(3);
s2(2)=s1(2);
s2(1)=s1(1);
s2(0)=s1(0);
--port_map equations end, line: 14, file: Test_0013.vhd
--port_map equations begin, line: 63, file: Test_0013.vhd
s3(31)=s2(31);
s3(30)=s2(30);
s3(29)=s2(29);
s3(28)=s2(28);
...
...
...
s3(4)=s2(4);
s3(3)=s2(3);
s3(2)=s2(2);
s3(1)=s2(1);
s3(0)=s2(0);
--port_map equations end, line: 30, file: Test_0013.vhd
```

W przedstawionym przykładzie dla każdej instrukcji mapowania były generowane równania boolowskie bazując na podstawie pełnego procesu kompilacji źródeł programu.

W równaniach boolowskich powstających w procesie kompilacji wykorzystujemy tylko trzy podstawowe operacje: or (|), and (&), not (!) [8]. Związane jest to z pierwotnym zastosowaniem wygenerowanych równań. Ich głównym przeznaczeniem była symulacja i weryfikacja na komputerze wieloprocesorowym opracowanym specjalnie do tego zadania. Jednostka taka składała się z wielkiej liczby procesorów zdolnych do wykonywania bardzo prostych operacji. Z tego też względu w istniejącym formacie wyjściowym nie można korzystać z podprogramów. Taki format został przyjęty na życzenie firmy ALDEC dla której początkowo był opracowywany kompilator języka VHDL.

Jeśli na tą samą jednostkę występuje kilka identycznych mapować to kilka razy muszą być wygenerowane równania dla tej samej jednostki. Jedyna różnica jaka występuje między równa-

niami to zmienione nazwy. Tak zakłada specyfikacja przyjęta podczas powstawania kompilatora. Czas potrzebny na wygenerowanie równań dla jednego mapowania zależy od złożoności mapowanej jednostki i jej architektury. Może to być kilka sekund lub kilkadziesiąt minut. Jeśli dla każdego mapowania będziemy przeprowadzali pełny proces przekładu źródła VHDL na równania boolowskie, to wielokrotne mapowanie na tą samą jednostkę może wykluczyć taki kompilator z możliwości przemysłowego zastosowania z powodu zbyt długiego czasu kompilacji. W takiej sytuacji należy pominąć najbardziej czasochłonne operacje.

Gdy choć raz wygenerujemy równania boolowskie dla danej jednostki to możemy wielokrotnie korzystać z tych równań dokonując w nich niezbędnych zmian. Wykorzystanie gotowych równań jest możliwe tylko w przypadku gdy wartość generic'a jest taka sama jak dla wygenerowanych poprzednio równań [7]. Oczywiście musimy zmienić nazwy we wszystkich zmiennych występujących w równaniach, związane jest to z wymogami zapisanymi w specyfikacji: przypisanie do tej samej zmiennej może występować tylko raz. Modyfikacją nazw zajmuje się jedna z funkcji. Zastępuje ona nazwy sygnałów występujących w mapowanej jednostce oraz modyfikuje nazwy pozostałych zmiennych. Wykorzystanie wygenerowanych raz równań umożliwia skrócenie czasu kompilacji. W takim przypadku czas kompilacji zależy od złożoności mapowanych jednostek. Jeśli architektura mapowanej jednostki zawiera niewiele instrukcji to przyspieszenie może pozostać niezauważone. Związane to jest z czasem potrzebnym na sprawdzenie czy dla danej jednostki były już generowane równania oraz jakie były wartości generic'a. Przykład 4 zawiera fragment układu cyfrowego z różnymi wartościami generic. W tym przypadku nie jest możliwe zastowanie przedstawionego algorytmu. Jeśli przy mapowaniu wartości generic byłyby takie same to przy drugim mapowaniu moglibyśmy skorzystać z opisanego algorytmu.

Przykład 4. Mapowanie Generic
Example 4. Generic map

Jednostka

```
entity g_lut is
   -- synopsys template
   generic ( N : integer := 4;
      INIT1 : integer := 1;
      INIT2 : integer := 0;
      INIT3 : integer := 0;
      INIT4 : integer := 0);
   port ( INP : std_logic_vector ( N-1 downto 0 );
      O : out std_logic);
end g_lut;
```

Komponent

```
   component g_lut
      generic ( N : integer;
         INIT1 : integer;
         INIT2 : integer;
         INIT3 : integer;
         INIT4 : integer);
      port ( INP : std_logic_vector ( N-1 downto 0 );
         O : out std_logic);
   end component;
```

Mapowanie

```
C16 : g_lut
generic map ( N => 3 , INIT1 => 234 , INIT2 => 0 , INIT3 => 0,
                        INIT4 => 0)
port map ( INP => LUT_C16_sig , O => C0_N9);

C17 : g_lut
generic map ( N => 4 , INIT1 => 202 , INIT2 => 0 , INIT3 => 0,
                    INIT4 => 0)
port map ( INP => LUT_C17_sig , O => syn251);
```

Przy tak skonstruowanym algorytmie należy szczególną uwagę zwrócić na funkcję modyfikującą nazwy. Musi być ona napisana z wykorzystaniem algorytmów których czas działania jest bardzo krótki, gdyż tylko to umożliwia przyspieszenie czasu kompilacji rozbudowanych projektów.

3. Podsumowanie

Bardzo istotną sprawą podczas generowania równań boolowskich jest opracowanie szybkich i skutecznych algorytmów wykonujących mapowanie. Na podstawie przeanalizowanych komercyjnych projektów wynika, że mapowania występują znacznie częściej niż wywołania podprogramów. Dla kilkunastu komercyjnych projektów napisanych w języku VHDL poddanych procesowi kompilacji przyspieszenie w najbardziej korzystnym przypadku wyniosło 4 razy w stosunku do pierwotnej wersji kompilatora. Jest to satysfakcjonująca wartość, która może być jeszcze powiększona poprzez modyfikację formatu w jakim są przechowywane wyniki kompilacji poszczególnych jednostek. Jeśli opracowalibyśmy odpowiedni format w którym przechowywane byłyby równania boolowskie np. rozbicie nazw zmiennych w równaniach na leksemy, to proces przekładu powinien ulec dalszemu skróceniu. Związane jest to z tym, że w zaimplementowanym algorytmie procesor dość dużo czasu spędza na zamianie nazw zmiennych. Dla dużej liczby równań musimy wykonać wiele zmian nazw, to powoduje wykonywanie wielu operacji na pamięci, które są bardzo czasochłonne. Rozmiar plików z równaniami dla niektórych jednostek może wynosić nawet kilka megabajtów. Jak widać istnieją jeszcze sposoby modyfikacji procesu generowania równań boolowskich dla instrukcji mapowania. Tylko szybkie algorytmy umożliwią przemysłowe zastosowanie kompilatora powstającego w Katedrze Technik Programowania Wydziału Informatyki Politechniki Szczecińskiej.

4. Literatura

[1] Adamski M., Specyfikacja, analiza i synteza reprogramowalnych sterowników logicznych, Materiały II Krajowej Konferencji Naukowej RUC'99, Szczecin, 14-16 kwietnia 1999, s. 11-20

[2] Aho A. V., Sethi R., Ullman J. D., Kompilatory. Reguły, metody i narzędzia, Wydawnictwo Naukowo-Techniczne, Warszawa, 2002

[3] Bielecki W., Hayduke S., Drążkowski R., Liersz M., Radziewicz M., Błaszyński P.: Organizacja kompilatora do syntezy układów logicznych z syntezowalnego podzbioru języka VHDL, Materiały IV Sesji Naukowej Informatyki, INFORMA, Szczecin 1999.

[4] Błaszyński P, Drążkowski R.: Organizacja analizatora semantycznego kompilatora języka VHDL do syntezy układów logicznych, P., R., Materiały III krajowej konferencji naukowej RUC'2000, INFORMA, Szczecin 2000.

[5] Błaszyński P.: Generacja i wyszukiwanie wartości semantycznych w kompilatorze języka VHDL służącym do generacji równań boolowskich, Materiały V Sesji Naukowej Informatyki, INFORMA, Szczecin 2000.

[6] FPGA Express, VHDL Reference Manual, 1997.

[7] Perry D. L., VHDL: Programming By Example, McGraw-Hill Professional, 4th edition, May 12, 2002

[8] Radziewicz M.: Translacja instrukcji sekwencyjnych języka VHDL, Pomiary Automatyka Kontrola 7/2006, 54-56

[9] Sołdek J., Miejsce układów reprogramowalnych w informatyce, Materiały I Krajowej Konferencji Naukowej. Reprogramowalne układy cyfrowe.

[10] Srikant Y. N., Shankar Priti, The Compiler Design Handbook: Optimizations & Machine Code Generation, CRC Press, 1st edition, September 25, 2002

[11] Wrona W., VHDL język opisu i projektowania układów cyfrowych, Wydawnictwo pracowni komputerowej Jacka Skalmierskiego, Gliwice 1998.

Artykuł recenzowany