Mariusz KAPRUZIAK
ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY, KATEDRA ARCHITEKTURY KOMPUTERÓW I TELEKOMUNIKACJI,
ul. Żołnierska 49, 71-210 Szczecin

Aplikacja wspomagająca projektowanie struktury procesorów programowalnych w układach FPGA

Dr inż. Mariusz KAPRUZIAK

Obronił prace doktorską na Politechnice Szczecińskiej w 2006r. na temat "Opracowanie koncepcji i realizacja dedykowanego procesora radia programowalnego". Obecnie prowadzi badania dotyczące procesorów o dynamicznie modyfikowalnej strukturze wewnętrznej w aplikacjach widzenia maszynowego.

e-mail: mkapruziak@wi.ps.pl

Streszczenie

Przejście z implementacji systemu na mikroprocesorze do wykorzystania układu FPGA jest często trudne. Zazwyczaj nie opłaca się poświęcenie czasu na przepisanie kodu już zaimplementowanych algorytmów. Skuteczniejszym rozwiązaniem jest przeniesienie samej struktury oryginalnego procesora do wnętrza układu FPGA. Zadanie przeniesienia struktury można częściowo zautomatyzować i przyspieszyć proponując właściwą aplikację wspomagającą. Aplikacja taka mogłaby także być efektywną pomocą dydaktyczną do prezentacji i eksperymentowania na różnych architekturach komputerów. W artykule przedstawiono propozycją właśnie takiej aplikacji.

Słowa kluczowe: procesor programowalny, FPGA.

FPGA soft processor design tool

Abstract

While improving current projects, transition form microprocessor based system to FPGA is often not straightforward. Time spent on code rewriting is not usually considered cost-effective. It seems to be more effective to implement the structure of a considered processor directly on FPGA and transfer the code unmodified. The task of cloning a real processor into FPGA structure could be partly automated and shortened by the right programming environment. Such environment could also serve as a helpful and efficient teaching tool, allowing students to see architecture at work and experiment with its own modifications. In the paper such an environment is presented. It is partially inspired by LISA project [1], but opposed to that the author tries not to put a user too far away from the resulting code. This environment is rather a time-saving code generator for schematical tasks (Fig. 1). As such, it allows defining the general structure of the resulting Verilog code (Fig. 3) and the parameters for ALU, control unit and bus address space (Figs. 4, 5, 6). Figs. 7 and 8 show examples of the resulting codes. The application is currently mainly used for teaching purposes but is planned to be developed to help in automatic project transformation from microcontrollers to FPGA SoC designs.

Keywords: soft processor, FPGA.

1. Wstęp

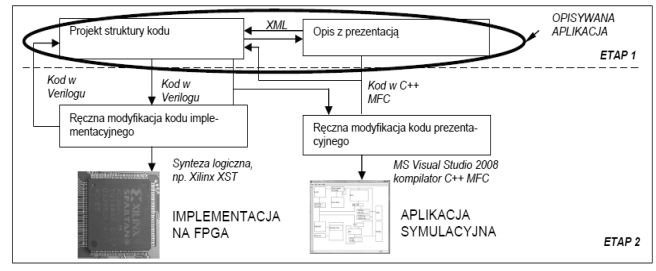
Projekt, który oryginalnie zainspirował autora pracy, jest LISA (Language for Instruction Set Architecture), stworzony na uniwersytecie w Aachen [1]. W wyniku prac na projektem powstało narzędzie, które na podstawie opisu w specyficznym opracowanym do tego celu języku, umożliwia wygenerowanie struktury procesora w FPGA. Dla tego procesora można pisać własne kody w jednym ze znanych języków proceduralnych, jak na przykład C czy C++. W artykule autor przedstawia swoją zmodyfikowaną wizję tak postawionego zadania. Modyfikacje dotyczą dwóch aspektów: (1) łatwej i przyspieszonej integracja starszych projektów z ich oryginalnymi kodami w jednym chipie FPGA oraz (2) dydaktycznego aspekt testowania różnych rozwiązań organizacji procesora, prezentacja wyników oraz przede wszystkim praktyczne pokazanie wpływu tych rozwiązań na ostateczną efektywność.

Aspekt pierwszy wynika z praktycznych doświadczeń z uaktualnianiem i modyfikacją wcześniejszych projektów, tak aby przenieść rozwiązanie na współczesne układy FPGA. Istnieje szereg zalet takiego podejścia, jak integracja kilku wcześniejszych procesorów/modułów w jednym elemencie oraz łatwa możliwość rozszerzania systemu o nową funkcjonalność. Aspekt drugi dotyczy natomiast użyteczności dydaktycznej. Istnieją dwa typy kursów, architektura systemów komputerowych oraz technika mikroprocesorowa, na których problem dogłębnego wytłumaczenia specyfiki różnych aspektów funkcjonowania procesora wymaga w zasadzie jego implementacji i przekonania się samemu co do efektywności tych rozwiązań. Oryginalnie opisywane w pracy narzędzie powstało na bazie doświadczeń z prowadzenia kursu techniki mikroprocesorowej dla trzeciego roku studentów informatyki ZUT. Na kursie realizuje się implementację różnych mechanizmów procesora w języku Verilog i implementacje ich na układach FPGA. Okazuje się, że istnieje część modułów, których kodowanie podlega pewnym schematom, możliwym do zautomatyzowania. Istnieje także część modułów na tyle specyficznych, że ich automatyzowanie nie opłaca się i tylko niepotrzebnie usztywniłoby strukturę kodu.

Do realizacji obu aspektów można próbować wykorzystać istniejące komercyjne lub naukowe rozwiązania. Przede wszystkim może to być wspomniany wcześniej projekt LISA. Jego wadą, z punktu widzenia założeń autora, jest fakt bardzo wyraźnego rozdzielenia opisu architektury od jego kodu w Verilogu. Istnieje także cały szereg skutecznych i łatwych w obsłudze narzędzi dedykowanych dla konkretnych architektury, jak Tensilica Xtensa [2] czy Coware ARM Amba Designer [3]. Dedykacja tych narzędzi dla konkretnej architektury wykluczyła jednak je z rozważań na tym zagadnieniem. Skłoniło to autora do podjęcia prac nad stworzeniem własnego środowiska.

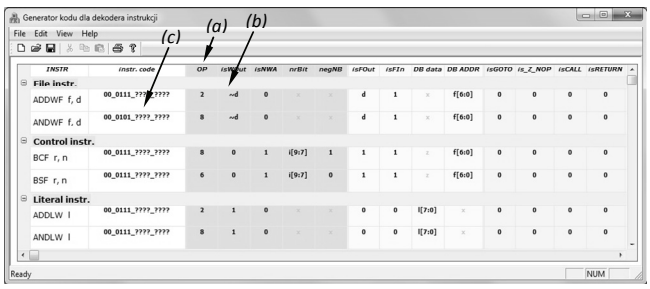
2. Proces projektowania procesora

Proces projektowania odbywa się w dwóch etapach (rys. 1). W pierwszym projektuje się ogólną strukturę kodu oraz fragmenty kodu, które można wygenerować na podstawie predefiniowanych szablonów. W wyniku generowane są kody w Verilogu oraz kody symulacyjne do Visual Studio 2008. Drugi etap dotyczy ręcznej modyfikacji tych kodów i wymaga do dalszej syntezy zewnętrznych narzędzi jak na przykład Xilinx XST [4] lub Altera Quartus2 [5] oraz Visual Studio 2008 do kompilacji kodu symulacyjnego. Kody z drugiego etapu mogą być następnie wczytane do pierwszego etapu, gdzie można zmodyfikować części szablonu i iteracyjnie dochodzić do ostatecznego rozwiązania. Zmiany dokonane w drugim etapie są widziane w etapie pierwszym.

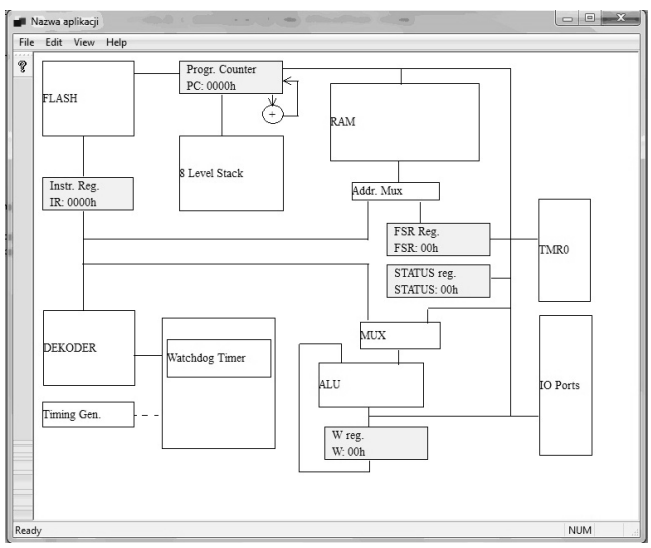


Rys. 1. Proces projektowania procesora
Fig. 1. Processor design process

Główna aplikacja dotyczy etapu pierwszego. Na rys. 3 przedstawiono przykładowy ekran z projektem procesorem PIC16F84A [6]. W kwadratowych lub okrągłych obiektach znajdują się moduły (rys. 3.a). Te zaznaczone ciemniejszym kolorem nie posiadają odpowiedniego generatora kodu (wymagają edycji ręcznej). Linie oznaczają szyny. Szyny ukryte mogą być zaznaczone linią przerywaną (rys. 3.b) lub wcale i tylko oznaczone w szczegółach modułu i na liście szyn.



Rys. 2. Tabela opisu dla generatora dekodera instrukcji, (a) nazwy kombinacji szyny sterującej, (b) przepisanie fragmentów IR na szynę sterującą, (c) definicja formatów instrukcji
Fig. 2. Description table for instruction decoder generator, (a) names for control bus constants, (b) assignment of IR signals to control bus signals, (c) instruction format definitions



Rys. 3. Ekran aplikacji projektowania struktury procesora, (a) przykładowe moduły, (b) normalne oznaczenie szyn, (c) szyny ukryte
Fig. 3. Application window for structure design, (a) modules for model processor, (b) normal bus symbol, (c) hidden bus

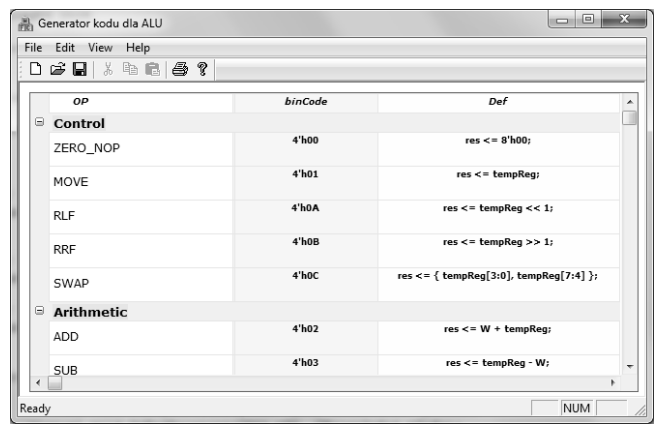
Z modelu generowany jest kod tak, że każdemu blokowi odpowiada dokładnie jeden moduł w kodzie o takiej samej nazwie. Część z modułów jest od razu wypełniana kodem, na podstawie opisu w bibliotekach generatorów. W aplikacji przewidziano na stan obecny cztery takie generatory:

- Generator modułu dekodera instrukcji
- Generator jednostki ALU
- Generator pamięci programu
- Generator szyny z układem mapowania adresów (mapa pamięci).

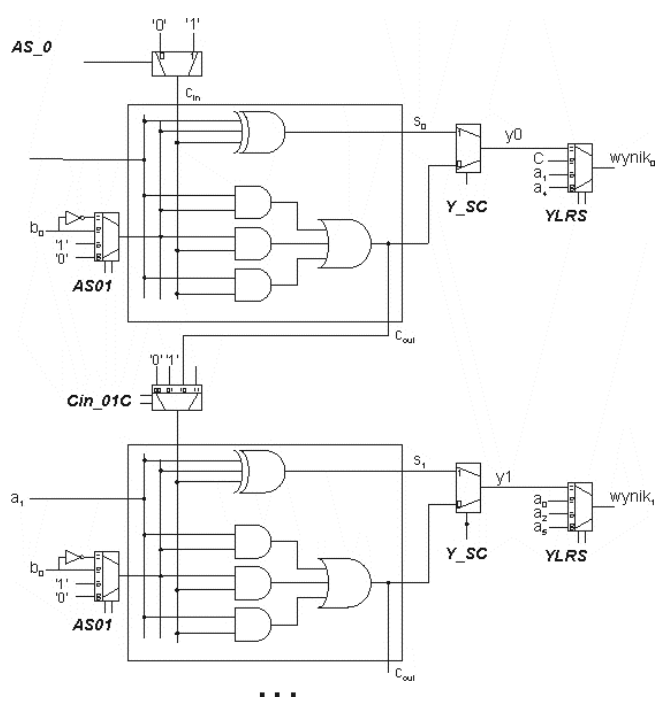
Kod dla modułu dekodera instrukcji generuje się za pomocą tabeli (rys. 2), gdzie każdej instrukcji określonej jednoznacznie przez jej kod (dwie pierwsze kolumny, rys. 2) odpowiada właściwa wartość kodu operacji ALU, sterowania multipleksera podawania stałej itp., stosownie do konstrukcji szyny sterującej. Zakłada się niejednorodność szyny sterującej, co jest jawnie opisywane przez nazwy kolumn w tabeli. Uwzględniono dodatkowo następujące rozszerzenia:

- Możliwość nazywania pewnych kombinacji logicznych, na przykład OP = MOVE, co odpowiada w tym przypadku kombinacji 00h (rys. 2.a).
- Możliwość nazwania pojedynczych bitów kodu maszynowego i użycia ich bezpośrednio przy określaniu wartości szyny (rys. 2.b)
- Możliwość określania wzorców dekodowania, tak aby odpowiadały formatom instrukcji dla różnych trybów adresowania(rys. 2.c).

Każdej instrukcji musi odpowiadać stosowna operacja do wykonania na ALU. Lista takich operacji definiowana jest w tabelce generatora jednostki ALU (rys. 4). Przewidziano w programie dwie metody generowania ALU: jedna oparta na strukturze z rys. 5 (nazwanej ALU0), druga oparta na behawioralnej instrukcji case (generator na rys. 4) . W przypadku ALU opartego na strukturze ALU0 automatycznie sugerowana jest lista operacji z możliwością wykasowania lub dodania nowej poprzez wpisanie właściwych wartości sygnałów sterujących multiplekserami. W przypadku ALU opartego na instrukcji case należy dla każdej operacji wpisać ręcznie kod implementacyjny (rys. 4).



Rys. 4. Tabela opisu dla generatora jednostki ALU
Fig. 4. Description table for ALU generator

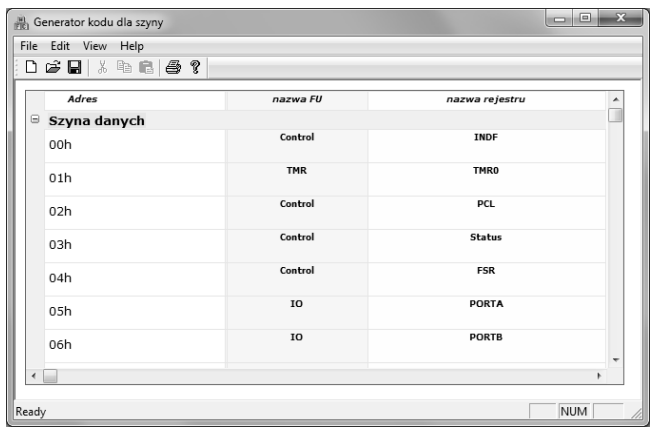


Rys. 5. Struktura ALU0
Fig. 5. ALU0 module structure

Generator pamięci programu pobiera plik HEX na wejściu i generuje odpowiadający kod na wyjściu. Jest to moduł, który po

każdej kompilacji wymaga ponownego wykonania. Przewidziano wybór umieszczenia pamięci programu w zewnętrznej pamięci Flash lub w wewnętrznej pamięci FPGA. Jeśli kod umieszczony jest w zewnętrznej pamięci należy podać przestrzeń adresów w których kod może być umieszczony.

Generator szyny z układem mapowania adresów dotyczy przypisania nazwom rejestrów ich adresów w przestrzeni adresowej wybranej szyny (może to być jedna szyna danych lub też wiele niezależnych szyn). Generowany jest na podstawie tabeli przypisania adresów (rys. 6), gdzie każdemu rejestrowi z każdego podłączonego do szyny modułu można przypisać konkretny adres. Kod wynikowy rozproszony jest na wszystkie moduły podłączone do szyny (rys. 8).



Rys. 6. Tabela opisu dla generatora szyny z układem mapowania adresów
Fig. 6. Description table for bus generator with address mapping

3. Kody procesora i schematy generowania kodów

Kody wynikowe formatowane są w taki sposób, aby były najbardziej czytelne i łatwe do modyfikacji. Dla przykładowego procesora PIC16F84A generator dekodera wygenerował kod tak jak na rys. 7. Dla każdej instrukcji przewidziany jest osobny wpis wypełniający szynę sterującą (na przykład określający operację dla ALU, rys. 7.a).

```
      ...
always @( posedge uP_CLK ) begin
  uP_CNT <= uP_CNT + 1;
  if( uP_CNT == 0 )  begin
     casex( IR[13:0] )
       // ..........................................
       14'b00_0111_????_????: begin   // ADDWF .....
         OP              <= 4'h2;          // 02 - ADD
         isFIn           <= 1;
         isWOut          <= ~IR[7];
         isFOut          <= IR[7];
         is_NOTW_ALU     <= 0;
         nrBit           <= 3'bxxx;
         negNrBit        <= 1'bx;
         isGOTO          <= 0;
         is_Z_NOP        <= 0;
         isCALL          <= 0;
         isRETURN        <= 0;
         outDataBUS      <= 8'bxxxxxxxx;
         if( IR[6:0] != 7'h00 )
               dataBUS_ADDR <= { STATUS_RP0, IR[6:0] };
         else  dataBUS_ADDR <= FSR;
          end
            ...
```

(a)

Rys. 7. Przykładowy fragment kodu dekodera instrukcji,
(a) zapis kodu operacji
Fig. 7. A passage from a model code for instruction decoder,
(a) code for ALU operation

Ze względu na niesyntezowalność na danym etapie kodów TLM (Transaction Level Modeling [7]) przez klasyczne narzędzia firmy Xilinx [8] kod szyny musi być rozbity na wszystkie moduły. Szyna podzielona jest jawnie na dwie części: szynę adresową (końcówka nazwy _ADDR) i szynę przenoszącą dane (końcówka nazwy _DATA). Każdemu modułowi przydzielona jest pewna przestrzeń adresów (dla przypadku z rys.8 od 0Ch a 7Fh). W modelu szyny uwzględniane są również zależności związane z etapami cyklu instrukcji (rys. 8, uP_CNT) jak również zależności od sygnałów sterujących (rys. 8, isFIn).

```
      ...
always @( posedge uP_CLK ) begin
   if( uP_CNT == 0 ) begin
   end else if( uP_CNT == 1 ) begin
   end else if( uP_CNT == 2 ) begin
   end else if( uP_CNT == 3 ) begin
    if( isFOut ) begin
       if( ( dataBUS_ADDR > 8'h0B ) &&
          ( dataBUS_ADDR < 8'h80 ) )
             mem[ laddr ] <= dataBUS_DATA;
    end
   end
end
      ...
```

Rys. 8. Przykładowy fragment kodu dotyczący obsługi szyny danych (w module pamięci RAM)
Fig. 8. A passage from a model code for bus implementation (in RAM module)

4. Podsumowanie

Artykuł ma głównie charakter implementacyjny i narzędziowy. Proponowana w pracy aplikacja przyspiesza i ułatwia prace przy implementacji architektur podstawowych procesorów popularnych w projektach typu system alarmowy czy sterownikach silników. Pozwala także na efektywne przygotowanie symulacji do różnych architektur procesorów jak również pozwala na szybkie przetestowanie różnych mechanizmów funkcjonowania procesora.

5. Literatura

[1] Schliebusch O., Hoffmann A., Nohl A., Braun H., Meyr H.: Architecture Implementation Using the Machine Description Language LISA, ASP-DAC/VLSI Design 2002, ISBN: 0-7695-1441-3.
[2] Gonzalez R. E., Xtensa: A configurable and extensible processor, IEEE Micro, vol. 20, pp. 60–70, Mar./Apr. 2000.
[3] Agrawal A., Gupta N.: Architecture Oriented Performance Optimizations for Bus Based System-on-Chip Designs Using TLM, CoWare, Inc. Tech. Rep, 2005.
[4] Xilinx Inc, XST User Guide 10.1, Xilinx 2008.
[5] Altera, Quartus-II Software Development Kit, Altera 2009.
[6] Barnett R. H., O'Cull L.: Embedded C programming and the microchip PIC, SA Cox – 2003.

otrzymano / received: 07.05.2010
przyjęto do druku / accepted: 04.06.2010 *artykuł recenzowany*