JAN KOWALSKI[*]

INFRASTRUKTURA DO ZRÓWNOLEGLANIA OBLICZEŃ NA UKŁADACH FPGA

INFRASTRUCTURE FOR PARALLEL COMPUTING ON FPGA BASED SYSTEMS

S t r e s z c z e n i e

W artykule przedstawiono projekt oraz wyniki implementacji wybranej metody zrównoleglenia obliczeń na układach FPGA. Całość została zaimplementowana używając jedynie języka syntezy wysokiego poziomu, Scharakteryzowano problemy związane z zarządzaniem takimi obliczeniami oraz zaproponowano rozwiązanie w postaci systemu opartego o układ ARM

Słowa kluczowe: FPGA, obliczenia równoległe, systemy wbudowane, HLS

A b s t r a c t

The article shows the project and implementation of chosen method of paralleling calculations on FPGA systems. Implementation was done using only High-Level Synthesis language. Problems related to the management of such calculations were characterized and a solution in the form of an arm-based system was proposed.

Keywords: FPGA, parallel computing, embedded system, HLS

---

[*] inż. Jan Kowalski, Instytut Teleinformatyki, Wydział Fizyki Matematyki i Informatyki,

Politechnika Krakowska

1. Wprowadzenie

Od lat 70 XX w. przewidywania Gordona Moore'a (nazwane później na jego cześć prawem Moore'a) były aktualnym (zgrubnym) wyznacznikiem wydajności systemów komputerowych [1]. Sytuacja ta zmieniła się w ciągu ostatnich kilku lat. Producenci układów scalonych wraz ze zmniejszeniem procesu produkcyjnego zbliżyli się do technologicznych (jak i fizycznych) granic możliwości, jakie niesie ze sobą wykorzystanie krzemu jako surowca bazowego [2]. Coraz trudniejsze staje się przyśpieszanie taktowania pojedynczego procesora - pojawia się problem związany z odprowadzaniem energii wydzielanej w formie ciepła co w połączeniu ze ścieżkami o rozstawie rzędu 7-14 nm powoduje efekty zniekształcające wyniki operacji. Z tych przyczyn producenci skierowali swoją uwagę w stronę innych rozwiązań aby zapewnić przyśpieszenie obliczeń przy zapewnieniu wysokiego stopnia skalowalności rozwiązań.

Odpowiedzią na ten problem okazało się prowadzenie jednoczesnych obliczeń na wielu jednostkach obliczeniowych. Implementowane jest ono na kilka sposobów. Najpopularniejsze są maszyny wieloprocesorowe, którymi dysponuje w dniu dzisiejszym każdy z nas. Do profesjonalnych zastosowań, gdzie wykonuje się obliczenia wysokiej wydajności wykorzystywane są maszyny wielokomputerowe działające w klastrach – nic nie stoi na przeszkodzie aby były to również komputery wieloprocesorowe.

Często rozwiązania te działają pod kontrolą pełnoprawnego systemu operacyjnego, czasowo niedeterministycznego. Dodatkową wadą jest użycie procesorów ogólnego zastosowania do prowadzenia specjalistycznych obliczeń.

W tym artykule przedstawiono inne podejście – użycie fizycznej architektury dedykowanej do rozwiązywania danego zagadnienia. Problem zarządzania rozwiązano dzięki wprowadzeniu zarządcy w formie mikrokontrolera opartego o architekturę ARM, który komunikuje się z zewnętrzną infrastrukturą. Część obliczeniowa – specjalizowana – jest implementowana z użyciem układu FPGA. Zarządca może działać pod kontrolą systemu czasu rzeczywistego lub bez niego tzw. '*bare metal'* co w połączeniu z macierzą bramek programowalnych umożliwia osiągnięcie dokładności czasu wykonywania obliczeń mierzonej w skrajnych przypadkach nawet w cyklach procesora.

Nowością jest użycie HLS (ang. *High-Level Synthesis Language*), który pozwala zastąpić skomplikowane języki opisu sprzętu (Verilog/VHDL) językiem C/C++ i udostępnia interfejs analogiczny do OpenMP. Został on rozszerzony mając tym samym o wiele większy wpływ na samą strukturę fizyczną układu a nie tylko na jej wykorzystanie.

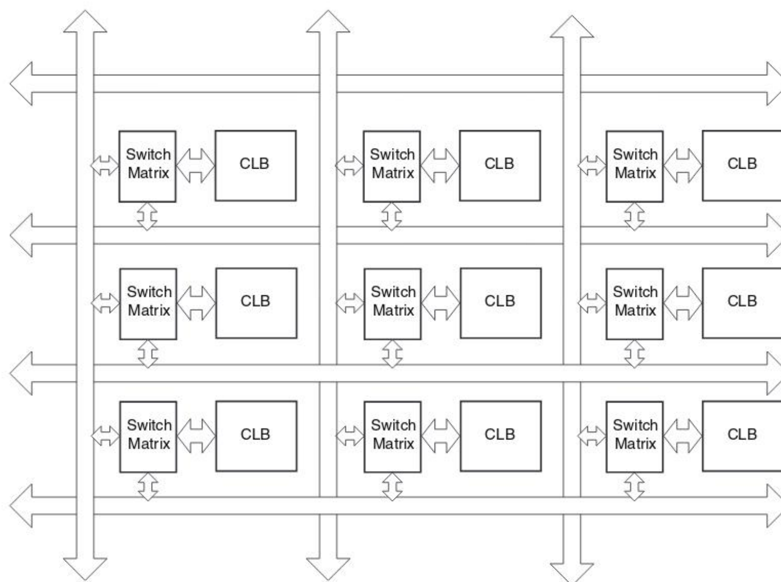
Ekosystem Vivado dostarczany przez firmę Xilinx udostępnia pełna gamę rozwiązań umożliwiając integrację podsystemów w jednym chipie co ogranicza koszt oraz skomplikowanie docelowych urządzeń. W wyniku takiego połączenia można uzyskać urządzenia o (teoretycznie) nieskończonych możliwościach zrównoleglenia fragmentów obliczeń, warunkiem jest niezależność obliczeń od siebie. Niewątpliwie zaletą takiego rozwiązania jest fakt energooszczędności układów FPGA (wykonują te same operacje co inne niespecjalizowane procesory ale używając do tego o rząd wielkości niższego taktowania). Jest to rozwiązanie lepsze niż używanie układów ASIC, ponieważ niweluje ich główną wadę w tego typu zastosowaniach – konieczność przeprojektowywania fizycznej struktury całego układu przy jakiejkolwiek zmianie.

2. Wprowadzenie do FPGA

Początek historii układów FPGA sięga lat 70 XX wieku. Pierwsze próby firmy Phillips z 'programowaniem' sprzętowej logiki dały owoc w wyniku FPLA (ang. *Field-Programmable Logic Array*) [3]. Były to struktury wyposażone w dwie 'przestrzenie', z logiką typu AND oraz typu OR. Umożliwiono ich programowe łączenie co pozwalało wykonać kilka nieskomplikowanych operacji logicznych.

W wyniku technologicznej ewolucji na czoło ze swoim nowatorskim pomysłem wyszła w roku 1985 firma Xilinx kładąc podwaliny pod rozwój całej branży. Ich idea była stosunkowo prosta i intuicyjna. Można ją zawrzeć w trzech postulatach:

1) zbudujmy blok logiczny, którego działanie możemy dowolnie zmieniać – zależnie od zapotrzebowania niech będzie w stanie wykonywać odmienne operacje logiczne (AND/OR/XOR/NAND/…)
2) umieśćmy wiele jego instancji w obrębie jednego układu tworząc macierz elementów logicznych
3) zezwólmy na dowolne programowanie połączeń pomiędzy tymi blokami (nie ograniczając inwencji programisty).



Rys. 1. Schemat ideowy macierzy bramek logicznych

Fig. 1. Schematic diagram of field-programmable gate array

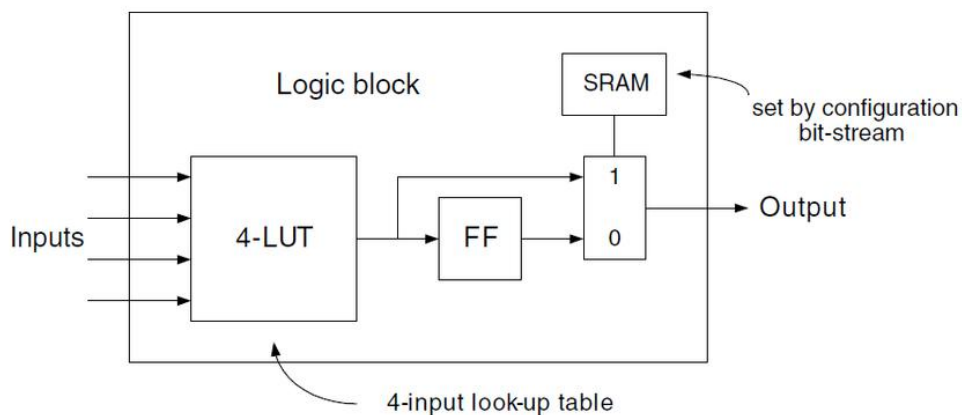
Realizacją tego celu stał się układ XC2064 wyposażony w 64 elementy logiczne, co porównując do dzisiejszych układów mających po dziesiątki (lub setki) tysięcy elementów, jest liczbą mało imponującą.

Ważną kwestią jest (często) błędne wyobrażenie o budowie samej 'macierzy bramek'. Nazwa wskazuje, jakoby miałaby ona się składać z podstawowych bramek logicznych.

Każdy inżynier ma przed oczami bramki typu OR, AND itp. W rzeczywistości jednak taka implementacja rozwiązania problemu byłaby wielce nieefektywna. Sprawę rozwiązano używając tzw. *Look-Up Table* mających 4 wejścia – pozwala to zaimplementować dowolny element logiki boolowskiej. Najpopularniejsze rozwiązania dodają to takiej komórki logicznej:

- przerzutnik typu D lub rejestr – wprowadza on możliwość przechowywania informacji
- multiplekser – pozwalający przełączać pomiędzy trybami pracy: logiką boolowską, przechowywaniem danych oraz wykonywaniem operacji arytmetycznych.

Dodatkowym elementem znajdującym się w układach FPGA są bloki I/O umożliwiające komunikacje z peryferiami. Zawierają one w sobie zintegrowane rezystory pociągające/obniżające, bufory, inwertery itd. – oczywiście również konfigurowalne.



Rys. 2 Schemat wnętrza pojedynczego bloku logicznego

Fig. 2. Interior diagram of a single logic block

3. "Programowanie" FPGA

W tej kwestii warto wyróżnić dwa aspekty:

- „programowanie" oraz przechowywanie „programu"
- języki programowania.

W swoim założeniu (w przeciwieństwie do układów mikroprocesorowych) układy FPGA nie posiadają pamięci ROM. Sprawia to, że po każdym odcięciu zasilania istnieje konieczność ponownego wgrania „programu". Nie mają one, dokładniej rzecz biorąc, nawet zdefiniowanej architektury.

Posługując się przykładem: programujemy 8-bitowy mikrokontroler typu AtMega328p oprogramowaniem avrdude. Posiadamy gotowy plik .hex, do budowy którego użyto *toolchaina*, skryptów linkera (z określonymi zakresami pamięci i ich zastosowaniami) oraz

plików konfiguracyjnych dedykowanych dla tego układu (zawierają one adresy poszczególnych rejestrów, ustawienia zegarów itd.). Wszystkie te podstawowe parametry są unormowane specyfikacją lub przynajmniej mają określone zakresy zmienności. Intuicyjne jest również to, że po zaprogramowaniu takiego układu możemy dowolną ilość razy go restartować. Program wywoływany przez *bootloader* będzie niezmiennie funkcjonować do czasu kolejnej zmiany *softu*.

Rzecz ma się zgoła odmiennie w przypadku układów FPGA. Zaczynami z 'białą kartką'. W naszej gestii jest przemyślenie i zaimplementowanie całej logiki. Jeśli np. dodamy rejestr, to nie będzie on miał przypisanego domyślnie adresu – będzie po prostu odseparowanych elementem. Tak samo (jeśli nie bardziej) problematyczna jest sprawa domen zegarowych. W układach ASIC programista ma do wyboru (najczęściej) kilka częstotliwości zależnych od dozwolonych mnożników w rejestrach konfiguracyjnych. Przykładem może być zmiana *fuse bitów* celem zmiany częstotliwości pracy całego układu AVR lub zmiana taktowania zewnętrznych magistral, takich jak SPI, I2C, I3C. Najczęściej problemy w takich wypadkach pojawiają się przy próbie komunikacji z zewnętrznymi sensorami lub innymi mikrokontrolerami. W FPGA problemy tego typu pojawiają się wewnątrz układu na porządku dziennym. Elementy działają zgodnie z cyklem zegara – zależnie od konfiguracji zachowania, cykle mogą być wywoływane różnymi przejściami lub stanami sygnału zegarowego. Z racji różnego czasu propagacji sygnału przez poszczególne bloki trzeba liczyć się z potrzebą synchronizacji. Problematyczna jest kwestia łączenia fragmentów działających z różnymi częstotliwościami (np. magistrale I/O). Aby uniknąć komplikacji związanych z tymi rozbieżnościami często używa się kolejek FIFO, które pełnią rolę buforów.

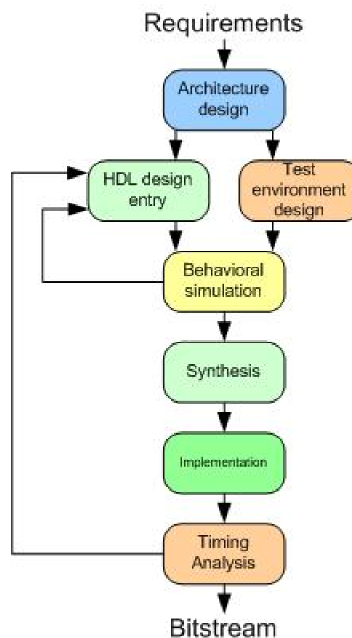
W wyniku tych wszystkich zależności w macierzy bramek programowalnych „program" nie jest nigdzie zapisywany na stałe. Chcąc używać poprawnej nomenklatury musimy wprowadzić 2 dodatkowe pojęcia zastępujące dotychczas bardzo dobrze ugruntowane w naszej świadomości:

1) ~~program~~ bitstream – w przypadku FPGA nie dostarcza się „programu", który analogicznie jak w komputerach klasy PC wykorzystuje procesor i rejestry do wykonywania predefiniowanych operacji. Tutaj wgrywa się schemat opisujący jak ma wyglądać fizyczna architektura, jak mają być skonfigurowane oraz połączone bloki
2) ~~programowanie~~ konfiguracja – elementy elektroniczne nie mogą być „programowane", można za to je konfigurować aby działały w pewien ustalony sposób

Skoro jesteśmy przy samym sprzęcie przedstawmy idee języków opisu sprzętu - HDL (*ang. Hardware Description Language)*. Powstały one celem uproszczenia procesu projektowania układów cyfrowych. Niemożliwe byłoby rysowanie schematów układów w skali miniaturyzacji VLSI. Słowny opis zapewnia lepszą skalowalność prezentacji.

Aktualnie liczą się dwa główne języki HDL:

- Verilog
- VHDL



Rys. 3. Proces rozwijania oprogramowania na układy FPGA

Fig. 3. Process of developing software for FPGA based systems

Nie wdając się dokładnie w różnice pomiędzy nimi (nie jest to głównym celem tekstu [5]), często można spotkać się z opinią, że Verilog jest lepszym językiem pierwszego kontaktu. Ciekawa jest idea budowania „programów". Wszystko powinno być modularne i skalowalne. Dany fragment struktury elektronicznej „zamyka" się w tzw. *IP Core'y*. Są to rdzenie zawierające w sobie własność intelektualną twórcy. W ten sposób można rozdystrybuować (również za opłatą) gotowy moduł np. procesora, nie martwiąc się o to, że kupujący przywłaszczy sobie nasz pomysł rozwiązania problemu. Biorąc pod uwagę potencjalnie nieskończoną dowolność programisty/elektronika każdy problem może mieć wiele takich rozwiązań.

Kolejnym krokiem na drodze ewolucji rozwijania układów FPGA stały się kompilatory języków wysokiego poziomu. Pierwsze akademickie próby stworzenia takich narzędzi sięgają 1994 roku. Dopiero jednak w 2004 zaczęły pojawiać się pierwsze komercyjne rozwiązania implementujące w akceptowalnym stopniu ten pomysł. Producenci sprzętu tacy jak Xilinx i Altera (Intel) upublicznili swoje implementacje w odpowiednio 2013 i 2017 roku [6].

Idea polega na udostępnieniu niskopoziomowym programistom x86 sposobu wykorzystania ich zdobytych dotychczas umiejętności. Kod napisany w języku (najczęściej) C/C++ jest „kompilowany" – przechodząc cały skomplikowany proces – do języków opisu sprzętu. Obniża to znacznie próg wejścia dla nowych programistów FPGA.

Wraz ze wzrostem skomplikowania powstających *IP Core'ów* sami producenci dostrzegli problem skalowalności ludzkiej pracy. Powoli niemożliwe staje się dla większości ludzi zaprojektowanie całego układu od podstaw – widoczny trend kieruje się ku zostawianiu coraz większej ilości „zbędnej" pracy automatowi.

W poniższym tekście spróbowano wykorzystać zalety jakie niesie ze sobą używanie języków HLS (dokładnie Xilinx *VivadoHLS*)

```
1  int idamax(int n, REAL dx[DIM1], int incx)
2
3  /*
4   finds the index of element having max. absolute value.
5   jack dongarra, linpack, 3/11/78.
6   */
7  {
8  #pragma HLS INLINE
9      REAL dmax;
10     int i, ix, itemp;
11
12     if (n < 1)
13         return (-1);
14     if (n == 1)
15         return (0);
16     itemp = 0;
17     dmax = fabs((double) dx[0]);
18     for (i = 1; i < PROBLEM_SIZE; i++) {
19 #pragma HLS PIPELINE
20
21         if (i < n) {
22             if (fabs((double) dx[i]) > dmax) {
23                 itemp = i;
24                 dmax = fabs((double) dx[i]);
25             }
26         }
27
28     }
29     return (itemp);
30 }
```

Rys. 4. Przykład użycia HLS (język C)

Fig. 4. Example of using HLS (C language)

4. Zrównoleglenie x86 a FPGA

Istnieje kilka sposobów zrównoleglenia obliczeń na platformie x86. Dwa najpopularniejsze to:

- rozszerzenie OpenMP
- standard MPI.

Pierwsze jest stosowane do uruchamiania programu równolegle na kilku rdzeniach. Drugi używany jest wykonywania programu wieloprocesorowo – od strony programisty nie ma różnicy czy program jest wykonywany lokalnie na komputerze czy na maszynie zdalnej.

Proces optymalizacji kodu pisanego z użyciem HLS jest w znacznym stopniu podobny do OpenMP. Obydwa rozwiązania starają się w analogiczny sposób – patrząc od strony programisty – zrównoleglić obliczenia. Mając gotowy kod w języku C dodajemy pragmy.

Są to instrukcje modyfikujące sposób działania kompilatora – dzięki nim możemy wymusić specyficzne zachowania względem poszczególnych elementów

OpenMP pozwala nam np. rozwinąć daną pętlę, wyrównując nakład obliczeniowy przypadający na poszczególne rdzenie. Dodatkowo możemy zdefiniować w jaki sposób ma być zorganizowana dostępność poszczególnych zmiennych – współdzielenie lub prywatne kopie.

```
1  startTime = omp_get_wtime(); //Start licznik czasu
2  omp_set_num_threads(n_threads); //Deklaracja liczby maksymalnie wykorzystanych watkow
3  #pragma omp parallel for schedule(static) private(i, j, x, y, mask_index, blueSum, greenSum, redSum, intensity)
4  for(i = 3*3; i<((image.rows-3)*(3*(image.cols-3))); i+=3){
5      blueSum = 0;
6      greenSum = 0;
7      redSum = 0;
8      mask_index = 0;
9      for(x = -2; x<=2; x++){
10         for(y = -2 ; y<=2; y++){
11             blueSum += *(image.data+i+(x+y)*3)*scales[mask_index];
12             greenSum +=  *(image.data+i+1+(x+y)*3)*scales[mask_index];
13             redSum +=  *(image.data+i+2+(x+y)*3)*scales[mask_index];
14             mask_index++;
15         }
16     }
```

Rys. 5. Przykład użycia openMP

Fig. 5. Example of using openMP

VivadoHLS idzie podobną drogą, istnieje jednak zasadnicza różnica. W wypadku OpenMP możemy ręcznie wymusić ile instancji pętli ma zostać wywołane. Po przekroczeniu liczby dostępnych fizycznie nieobciążonych rdzeni zaobserwujemy negatywne przyśpieszenie. Będzie to spowodowane coraz większymi narzutami systemowymi na proces zarządzania i synchronizowania programu niż samymi obliczeniami. Układy FPGA nie mają tej wady. Przykładowo mając pętlę, która powinna wykonać się 100x (zakładając, iż iteracje są niezależne od siebie i innych czynników) bez problemu można „rozwinąć" pętlę 100x. W HDL zostanie wygenerowane 100 instancji wykonujących tą samą operację (w tych samych cyklach zegarowych). Minusem tego rozwiązania będzie pojawianie się innych wąskich gardeł, z których istnienia programista musi sam sobie zdawać sprawę. Będą to m.in.

a) dostęp do pamięci – bloki mają ograniczona ilość interfejsów I/O, w grę wchodzi znajdywanie kompromisu między rozmiarami BRAM'ów, wielkością używanych słów oraz wykorzystaniem elementów logicznych
b) potokowanie/buforowanie danych – używając danych ze źródeł o innym taktowaniu/czasie propagacji trzeba przeciwdziałać opróżnianiu kolejki
c) ilość danych, dynamiczna alokacja pamięci – posługując się HLS prosto zapomnieć, że wynik naszej pracy zostanie finalnie przeniesiony bezpośrednio na sprzęt. Prowadzi to często do prób dynamicznej alokacji struktur. Jest to niestety niemożliwe. HDL wymaga znajomości rozmiarów buforów, tablic i szerokości interfejsów przed rozpoczęciem procesu syntezy.

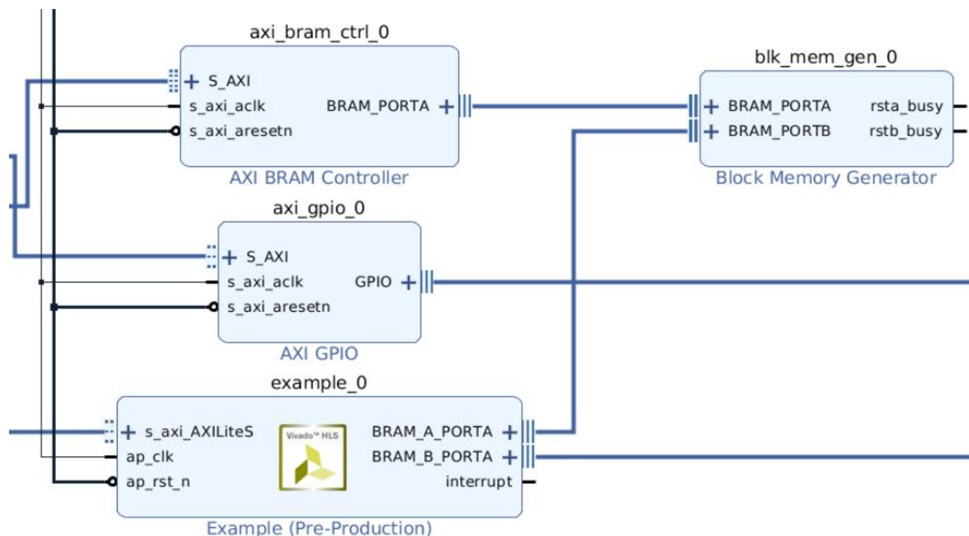
```
void copy_to_bram(volataile int *AXI, volataile int bram_a[50]){
#pragma HLS interface bram port=bram_a name=BRAM_A
#pragma HLS INTEFACE axis register both depth=1 port=AXI name=AXI_STREAM
#pragma HLS INTERFACE s_axilite port=return

    for(int i=0;i<50;i++)
    {
        (*bram_a) = AXI[i];
        (*bram_a)++;
    }
}
```

Rys. 6. Definiowanie interfejsów z użyciem pragm HLS

Fig. 6 Defining interfaces using HLS pragms

HLS posiada jeszcze jedną specyficzną cechę wyróżniającą go na tle innych rozwiązań. Musi istnieć sposób dodania interfejsów potrzebnych do późniejszego połączenia z resztą ekosystemu układu FPGA. Używając dodatkowych pragm możemy wyspecyfikować owe interfejsy. Jest to ważne z kilku przyczyn. Warto cofnąć się do idei rozwijania oprogramowania w językach opisu sprzętu. Wszystko na być dobrze skalowalne. Z tego powodu gotowy kod – po przetłumaczeniu na Verilog/VHDL – jest eksportowany do *IP Core'a*. Daje nam to kolejną 'skrzynkę', która poza oznaczeniem nie różni się niczym od każdego innego modułu pisanego w innych językach. Umożliwia to bezproblemową integracją wielu rozwiązań (w tym zbudowanych w przeszłości schematów) bez konieczności ich żmudnego przepisywania.



Rys. 7. Bezproblemowe łączenie modułów pisanych w różnych językach

Fig. 7. Seamless integration of modules written in different languages

5. Problem zarządzania zdalnego

Analizując poprzednie podrozdziały można wysnuć pytanie: jak zarządzać układem bez procesora? W dzisiejszych czasach wszystko zmierza do zapewnienia możliwości zdalnego nadzorowania i sterowania. Układy FPGA zdają się przeczyć temu trendowi. Zwracając uwagę na cały tor rozwijania oprogramowania – schematów budowy - można natrafić na IP Core'y dodające funkcje interakcji z zewnętrznymi interfejsami. Mamy np. dostępne za darmo bloki dodające obsługę standardu Fast Ethernet. Jest to jednak tylko i wyłącznie część fizyczna tego interfejsu. Napisanie całej logiki obsługującej transmisję od strony „aplikacji" zostaje w gestii programisty.

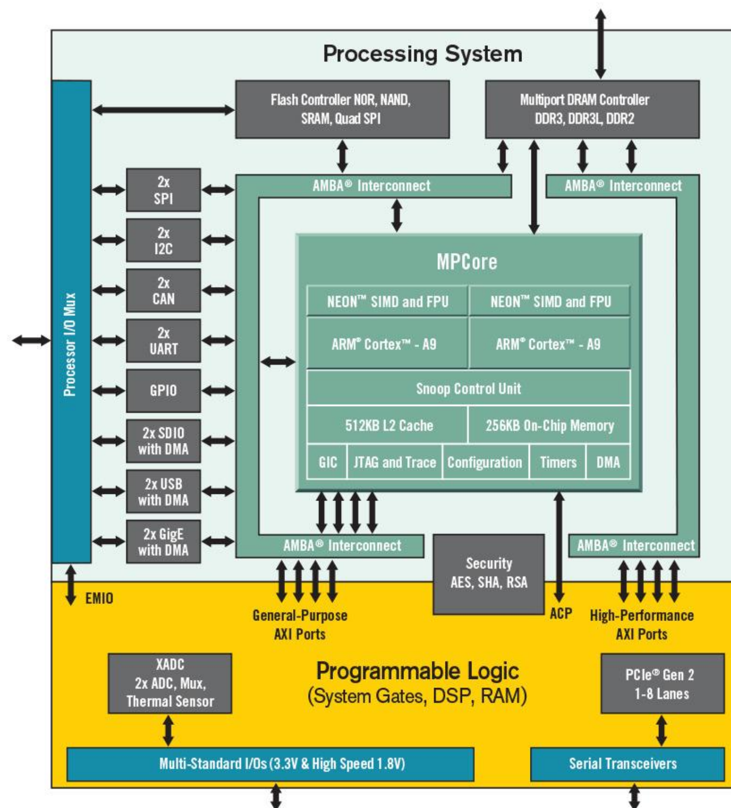
Jest to oczywiście wykonalne ale znowu zbliżamy się do tego, od czego staramy się odchodzić - implementowania wszystkiego od podstaw. Warto nadmienić jeszcze jedną istotną kwestię. Czas kompilacji programu, który moglibyśmy użyć na maszynie będącej zarządcą, jest niewspółmiernie mniejszy w stosunku do czasu syntezowania schematu i generowania bitstreamu. Mówimy tutaj przykładowo o generowaniu programu na architekturę ARM w czasie rzędu kilku minut oraz generowaniu schematu na układ FPGA począwszy od kilkunastu minut, nawet na godzinach kończąc. Czasochłonny dodatkowo (przy zastosowaniu specjalistycznych pragm dotyczących dostępu do BRAMu i jego organizacji) jest proces rozmieszczania fragmentów schematu w logice.

Istnieje możliwość zbudowania mikroprocesora „wewnątrz" układu FPGA. Niesie to jednak ze sobą wiele wad takich jak:

- znaczne ograniczenie dostępnej ilości elementów logicznych
- brak „namacalnego" procesora – utrudnione debugowanie
- brak retencji rejestrów/ustawień po wyłączeniu zasilania
- konieczność wyboru jednej z 2 ścieżek:
    1) użycie *softprocessora* od producenta – uzależnienie od ekosystemu i implementacji producenta
    2) użycie darmowego *IP Core'a* – tylko stosunkowo proste układy są dostępne w takiej formie.

6. Idea rozwiązania bazującego na platformie ARM

Po przedstawieniu często napotykanych problemów pora na omówienie potencjalnego rozwiązania. Do rozważań wybrano produkowany przez firmę Xilinx układ Zynq 7000. Łączy on w obrębie jednej obudowy macierz bramek programowalnych oraz mikrokontroler ARM.



Rys. 8. Schemat hybrydowego układu Xilinx Zynq 7000

Fig. 8. Scheme of Xilinx Zynq 7000 hybryd system

Tak „kompaktowe" umiejscowienie upraszcza proces tworzenia skomplikowanych połączeń pomiędzy logiką a procesorem. Jeśli spróbowalibyśmy zaprojektować płytkę PCB, która miałaby pomieścić jakikolwiek inny procesor i układ FPGA, napotkalibyśmy następujące problemy:

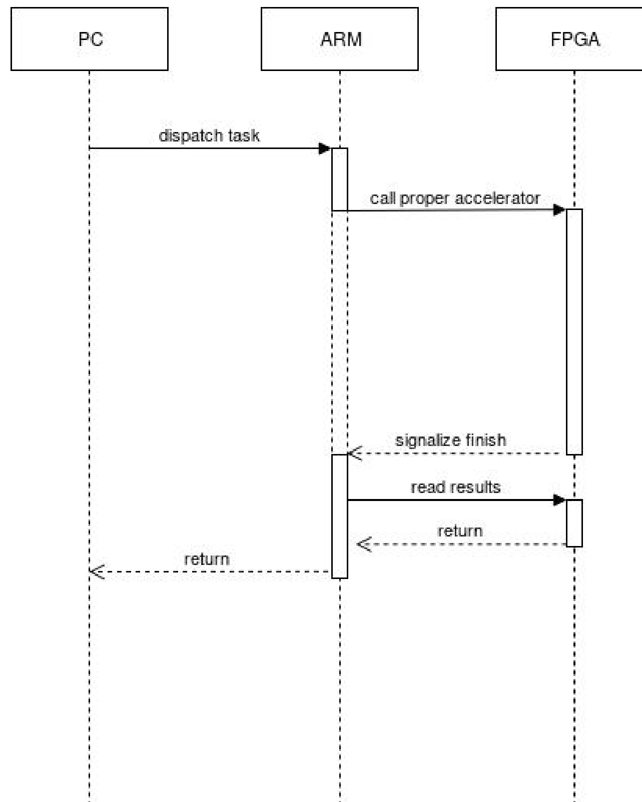
- przy dużych częstotliwościach (powyżej kilku MHz) ścieżki stają się bardzo wrażliwe na elektromagnetyczne zakłócenia transmisji
- jakakolwiek zmiana (np. nóżki mikrokontrolera) wymusza przebudowę płytki co wiąże się z kosztami.

Użycie zintegrowanego rozwiązania umożliwia skupienie się na części programistycznej, tym samym upraszczając prototypowanie części elektronicznej. Ponadto Xilinx oferuje dedykowane interfejsy, które bezproblemowo łączą się z *IP Core'ami* osiągając przy tym zadowalające prędkości.

Przejdźmy do samej platformy ARM. Mamy tutaj dostępne trzy osoby rozwijania oprogramowania:

a. *Bare Metal* – pisany kod jest wykonywany bezpośrednio na mikrokontrolerze
b. *FreeRT OS* – kod jest uruchamiany na systemie operacyjnym czasu rzeczywistego, dodaje on wsparcie dla wielowątkowości
c. *PetaLinux* – pełnoprawny system operacyjny.

Zależnie od tego, którą opcję wybierzemy, będziemy w stanie różnym nakładem pracy skomunikować się z zewnętrznymi urządzeniami bądź siecią. W tym momencie dostajemy do ręki narzędzia, których szukaliśmy. ARM staje się zarządcą, interfejsem ze światem. Posiada on bezpośredni dostęp do logiki (dzięki wsparciu producenta), co po przygotowaniu odpowiedniego programu pozwoli np. zacząć pomiar czasu i wywołać odpowiedni akcelerator do zaistniałych warunków. Natomiast układ FPGA wystarczy, że nada sygnał po skończeniu pracy - wtedy zarządca może odczytać wyniki obliczeń.



Rys. 9. Diagram użycia systemu z mikrokontrolerem ARM jako zarządcą

Fig. 9. Use diagram of system with the ARM microcontroller as a manager

W takim zastosowaniu praca akceleratora trwa pewien (z góry wiadomy – poznany w trakcie kompilacji komponentu) okres czasu. Odciąża to główny procesor pozwalając np. obsłużyć większą ilość zapytań od klientów.

7. Podsumowanie

Wraz z rozwojem architektury FPGA i dojrzewaniem środowisk dostarczanych przez producentów można zauważyć interesujący trend. Coraz chętniej są dostarczane rozwiązania zmniejszające nakład pracy programisty mającego zaimplementować daną funkcjonalność. Zmiana czegoś tak pozornie trywialnego jak język pisania komponentów znacząco skraca czas prototypowania i debugowania.

Nie można jednak zapomnieć, że o ile języki syntezy wysokiego poziomu są w stanie skrócić czas pisania modułu, to nie zwalniają programisty od myślenia. Cały czas jego wiedza i doświadczenie jest niezastąpione w momencie optymalizacji działania projektu. Zmiana kilku opcji może zmniejszyć czas oczekiwania pomiędzy kolejnymi operacjami o setki cykli dając znaczące przyśpieszenie.

Na rynku istnieje kilka popularnych rozwiązań wykorzystywanych do zrównoleglenia obliczeń. Różnią się one w głównej mierze wykorzystywaną architekturą. W przypadku x86 używany jest standard przesyłania komunikatów między procesami – MPI. Niewątpliwą jego zaletą jest transparentność od strony programistycznej. Identyczny program może zostać uruchomiony na pojedynczym komputerze oraz na klastrze obliczeniowym. Od kilku lat na popularności zyskują również technologie bazujące na kartach graficznych. Prym tutaj wiedzie platforma CUDA dostarczana przez firmę Nvidia. Umożliwia ona wykorzystanie znacznie większej ilości procesorów CUDA, które pomimo niższych częstotliwości taktowania osiągają nieporównywalnie lepsze całościowe osiągi. Limitacją staje się jednak często problem dostępu do podręcznej pamięci oraz wymóg obecności komputera wyposażonego w szereg dodatkowych komponentów.

W artykule przedstawiono przykład rozwiązania bazującego na ekosystemie firmy Xilinx. Tok tworzenia oprogramowania jest możliwy od odtworzenia na tańszych i mniej rozbudowanych hybrydowych układach. W profesjonalnych zastosowaniach (przy użyciu układów o większej ilości elementów logicznych) dochodzi kolejny sposób rozwoju systemów takiego typu. Programista dostaje „uproszczony" sposób zarządzania procesem dewelopmentu. W aplikacji *Vivado SDSoc* pisząc kod na rdzenie ARM dostarczana jest opcja wydzielania funkcji. Polega ona na odseparowaniu fragmentu funkcjonalności oraz oddelegowaniu go do logiki – używa się tutaj ponownie *Vivado HLS*. Daje to jeszcze lepsze rezultaty w kwestii optymalizacji czasu integracji owych rozwiązań.

Użycie architektury ARM w połączeniu z układami FPGA jest zdecydowanie wartym uwagi rozwiązaniem w miejscach, gdzie wykonywane są krytyczne czasowo operacje. Problemem jednak może okazać się koszt, który również stanowi ważny aspekt biznesowy wszelakich przedsięwzięć.

Literatura

[1] *Wpis w encyklopedii*, [online]:
britannica.com/technology/Moores-law [dostęp: 26.03.2019].

[2] Lieven Eeckhout, *Is Moore's Law Slowing Down? What's Next,* 2017 [online]: ieeexplore.ieee.org/document/8013504 [dostęp: 26.03.2019]

[3] Peter Cheung, *Introduction to FPGAs*, 2014 [online] ee.ic.ac.uk/pcheung/teaching/ee2_digital/Lecture%202%20-%20Introduction%20to%20FPGAs.pdf [dostęp: 16.04.2019]

[4] Anshul Thakur, *FPGA: An Overview*, [online], engineersgarage.com/articles/fpga-tutorial-basics [dostęp: 16.04.2019]

[5] Rob Dekker, *What's the Difference Between VHDL, Verilog, and SystemVerilog?, 2014* [online], electronicdesign.com/what-s-difference-between/what-s-difference-between-vhdl-verilog-and-systemverilog [dostęp: 22.04.2019]

[6] Nane, R.; Sima, V. M.; Pilato, C.; Choi, J.; Fort, B.; Canis, A.; Chen, Y. T.; Hsiao, H.; Brown, S. ,*A Survey and Evaluation of FPGA High-Level Synthesis Tools, 2016,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 35 (10): 1591–1604.[online], ieeexplore.ieee.org/document/7368920 [dostęp: 22.04.2019]

[7] Cem Ünsalan, Bora Tar, *Digital System Design with FPGA: Implementation Using Verilog* and *VHDL*, 2017