|  |  |
| --- | --- |
|  | **AKADEMIA GÓRNICZO-HUTNICZA** |

Raport z projektu

**Sprzętowa implementacja algorytmu Karatsuba**

z przedmiotu

**Systemy dedykowane w układach programowalnych**

Elektronika i Telekomunikacja, Systemy Wbudowane, rok I

*Marcin Maj, Jakub Zimnol*

semestr letni 22/23, grupa poniedziałek 18:00

prowadzący: Sebastian Koryciak

19.06.2023

# 1. Podstawy teoretyczne

Algorytm Karatsuby to rekurencyjny algorytm szybkiego mnożenia typu divide-and-conquer. W algorytmie Karatsuby ilość potrzebnych do wykonania operacji mnożenia jest mniejsza niż w klasycznym algorytmie mnożenia. Jego złożoność obliczeniowa wynosi , podczas gdy złożoność obliczeniowa zwykłego mnożenia wynosi . Z racji rekurencyjności algorytm Karatsuby jest szybszy od klasycznego algorytmu tylko dla odpowiednio dużych liczb.

Oprócz szybkości, w kontekście FPGA zaletą algorytmu jest mniejsza ilość mnożeń, dzięki czemu wykorzystanie algorytmu redukuje zużycie bloków DSP.

**Zasada działania algorytmu:**

Aby pomnożyć dwie liczby n-cyfrowe (lub n-bitowe) x i y o podstawie B, należy je rozdzielić na dwie części:

gdzie n = 2m, oraz x2 i y2 są mniejsze niż Bm. Wynik mnożenia wynosi wtedy:

gdzie:

Standardową metodą byłoby pomnożenie czterech czynników osobno i dodanie ich po odpowiednim przesunięciu. Daje to algorytm o złożoności . Algorytm Karatsuby polega na policzeniu czynnika z1 w następujący sposób:

A więc ostatecznie:

Pozwala to na zredukowanie całkowitej ilości mnożeń z 4 do 3.

Pseudokod algorytmu (źródło: [Karatsuba algorithm - Wikipedia](https://en.wikipedia.org/wiki/Karatsuba_algorithm)):

function karatsuba(num1, num2)

**if** (num1 < 10 or num2 < 10)

**return** num1 × num2 */\* fall back to traditional multiplication \*/*

*/\* Calculates the size of the numbers. \*/*

m = max(size\_base10(num1), size\_base10(num2))

m2 = floor(m / 2) */\* m2 = ceil (m / 2) will also work \*/*

*/\* Split the digit sequences in the middle. \*/*

high1, low1 = split\_at(num1, m2)

high2, low2 = split\_at(num2, m2)

*/\* 3 recursive calls made to numbers approximately half the size. \*/*

z0 = karatsuba(low1, low2)

z1 = karatsuba(low1 + high1, low2 + high2)

z2 = karatsuba(high1, high2)

**return** (z2 × 10 ^ (m2 × 2)) + ((z1 - z2 - z0) × 10 ^ m2) + z0

# 2. Założenia projektu

Założeniem projektu jest stworzenie IP-core implementującego algorytm mnożenia „Karatsuba”, porównanie wyników syntezy dla klasycznego mnożenia i karatsuby oraz zaimplementowanie komponentu jako moduł współpracujący z procesorem ARM za pomocą interfejsu AXI-Lite.

Plan zakłada implementację mnożenia liczb:

* stałoprzecinkowych
* dodatnich
* ujemnych
* 64, 128, 256 bitowych

# 3. Opis funkcjonalny

Moduł Karatsuba komunikuje się z procesorem ARM za pomocą interfejsu AXI Slave poprzez protokół AXI-Lite. Zaimplementowane zostało mnożenie dodatnich liczb 64 bitowych stałoprzecinkowych.

# 4. Opis rejestrów

IP karatsuba zawiera 10 rejestrów, każdy rejestr jest 32 bitowy. Opis rejestrów przedstawiono   
w tabeli 1. Parametr BASEADDR zadeklarowany jest w pliku main.c, również parametry z offsetem dla poszczególnych rejestrów zostały zapisane w pliku main.c

Tabela 1. Opis rejestrów IP karatsuba

|  |  |  |  |
| --- | --- | --- | --- |
| **Nazwa rejestru** | **Opis** | **Adres  (adres bazowy + offset)** | **Nazwa parametru zawierającego wartość offsetu** |
| slv\_reg0 | Rejestr do zapisu i odczytu, zawiera bity [31:0] pierwszego czynnika mnożenia | BASEADDR + 0x00 | A\_LOW |
| slv\_reg1 | Rejestr do zapisu i odczytu, zawiera bity [63:32] pierwszego czynnika mnożenia | BASEADDR + 0x04 | A\_HIGH |
| slv\_reg2 | Rejestr do zapisu i odczytu, zawiera bity [63:32] drugiego czynnika mnożenia | BASEADDR + 0x08 | B\_HIGH |
| slv\_reg3 | Rejestr do zapisu i odczytu, zawiera bity [31:0] drugiego czynnika mnożenia | BASEADDR + 0x0C | B\_LOW |
| slv\_reg4 | Rejestr tylko do odczytu zawierający bity [31:0] wyniku | BASEADDR + 0x10 | RESULT\_0 |
| slv\_reg5 | Rejestr tylko do odczytu zawierający bity [63:32] wyniku | BASEADDR + 0x14 | RESULT\_1 |
| slv\_reg6 | Rejestr tylko do odczytu zawierający bity [95:64] wyniku | BASEADDR + 0x18 | RESULT\_2 |
| slv\_reg7 | Rejestr tylko do odczytu zawierający bity [127:96] wyniku | BASEADDR + 0x1C | RESULT\_3 |
| slv\_reg8 | Rejestr tylko do odczytu, jest to wewnętrzny rejestr statusowy. W użyciu są bity [3:0], gdzie każdy z bitów sygnalizuje zapis do odpowiedniego rejestru slv\_reg | BASEADDR + 0x20 | REG8 |
| slv\_reg9 | Rejestr tylko do odczytu, najmłodszy bit rejestru sygnalizuje zakończenie mnożenia i gotowość do odczytu | BASEADDR + 0x24 | STATUS\_REG |

# 5. Schematy blokowe

Poniżej znajdują się dwa schematy, pierwszy z nich obrazuje cały system zawierający procesor ARM, moduł Karatsuba oraz potrzebne komponenty. Drugi schemat przedstawia sposób implementacji algorytmu.

## a) Schemat top-level

A diagram of a computer

Description automatically generated with low confidence

Rys 1. Schemat systemu

## b) Schemat blokowy modułu karatsuba, link do obrazka w lepszej jakości: [maj77/karatsuba.svg](https://raw.githubusercontent.com/maj77/Karatsuba_project/main/karatsuba_64bit_final.svg)

Rys 2. Schemat bloku realizującego algorytm mnożenia „Karatsuba”

# 6. Weryfikacja

Do zweryfikowania poprawności działania modułu Karatsuba stworzony został testbench oraz skrypt generujący dane wejściowe dla algorytmu. Skrypt generuje dane wejściowe, które zapisuje do pliku, następnie testbench ładuje zawartość pliku do tablicy. Testbench przekazuje liczby z tablicy do modułu Karatsuba oraz oblicza iloczyny, które są porównywane z wynikami z modułu Karatsuba.

Przykładowe logi z symulacji:

A picture containing text, screenshot, font

Description automatically generated

Rys 3. Log informujący o prawidłowym wyniku

A picture containing text, screenshot, font

Description automatically generated

Rys 4. Log informujący o nieprawidłowym wyniku

Obliczenie wyniku zajmuje 2 cykle zegara:

A screenshot of a video game

Description automatically generated with medium confidence

Rys 5. Przebiegi otrzymane podczas testowania modułu Karatsuba

# 7. Wyniki syntezy

W celu porównania zajętości zasobów przeprowadzono syntezy dla samego modułu Karatsuba oraz dla modułów wykonujących klasyczne mnożenie. Najbardziej interesuje nas ilość wykorzystanych bloków DSP. Pełne raporty z syntezy dla wersji 64 bit (klasyczna i Karatsuba) znajduje się na końcu raportu.

Tabela 2. Zajętość bloków DSP

|  |  |  |
| --- | --- | --- |
| **Typ modułu** | **Szerokość bitowa wejść** | **Ilość użytych bloków DSP** |
| Karatsuba | 64 | 10 |
| Karatsuba | 128 | 301) |
| Karatsuba | 256 | 901) |
| Klasyczny | 64 | 16 |
| Klasyczny | 128 | 64 |
| Klasyczny | 256 | 225 |

1) jest to przybliżona wartość obliczona na podstawie ilości wykorzystanych bloków DSP dla modułu mnożącego liczby 64 bitowe, w praktyce może różnić się o kilka bloków, prawdopodobnie 5-15 bloków więcej, w zależności od wersji (128/256bit).

Rys 6. Porównanie wykorzystania bloków DSP

# 8. Podsumowanie

Aktualna wersja projektu posiada zaimplementowany algorytm Karatsuby pozwalający na mnożenie 64 bitowych dodatnich liczb stałoprzecinkowych. Podstawową zaletą algorytmu jest zmniejszone zapotrzebowanie na bloki DSP, w przypadku liczb 64 bitowych różnica wynosi tylko 4. Różnica ta może być zadowalająca w przypadku gdy układ FPGA ma bardzo małą ilość bloków DSP i każdy blok jest „na wagę złota”. Algorytm może zastąpić klasyczne mnożenie kiedy wystąpi konieczna potrzeba zredukowania ilości wykorzystanych bloków DSP.

W przypadku mnożenia liczb 128 i 256 bitowych róznica między klasycznym mnożeniem a Karatsubą jest już zadowalająca i wynosi ona odpowiednio ~30 i ~120 bloków DSP.

**Podsumowując:** W przypadku gdy w projekcie wystąpiłaby potrzeba mnożenia liczb 64 bitowych zdecydowalibyśmy się na zaimplementowanie 64 bitowej wersji algosrytmu Karatsuba tylko w ostateczności ponieważ zysk jest niewielki. W przypadku gdy w projekcie wystąpiłaby potrzeba mnożenia liczb 256 bitowych rozważylibyśmy implementację algorytmu Karatsuby w pierwszej kolejności ponieważ różnica 120 bloków DSP jest znaczna, również w kontekście poboru mocy przez układ.

**Możliwości rozwoju projektu:**

* Implementacja mnożenia liczb ujemnych
* Implementacja mnożenia liczb 128 i 256 bitowych
* Analiza algorytmu w kontekście czasu wykonywania operacji

# Dodatek – Wyniki syntezy algorytmu Karatsuba oraz klasycznego mnożenia

1. Karatsuba

A screenshot of a computer

Description automatically generated with low confidence A picture containing text, screenshot, font, number

Description automatically generated A screenshot of a computer

Description automatically generated with medium confidence

1. Klasyczne mnożenie

A picture containing text, screenshot, font, number

Description automatically generated

A picture containing text, screenshot, font, number

Description automatically generated

A screenshot of a computer

Description automatically generated with low confidence