AGH University of Krakow

Faculty of Computer Science, Electronics and Telecommunication

Institute of Electronics

Systemy dedykowane w układach programowalnych

Implementacja oraz porównanie algorytmu Bitconic sort w układzie FPGA

> **NOTE:** This tutorial shows how to start your adventure with the PYNQ system using the AMD-Xilinx KRIA KV260 development platform.
> After rebuilding the project in Vivado, all examples can be run on any FPGA SOC platform like Zedboard, Zybo.

Spis treści

ver 0.2.1

Cel projektu

Celem projektu jest stworzenie sprzętowej implementacji algorytmu sortowania bitconic z wykorzystaniem układu Zynq, który łączy elastyczność procesora ARM z mocą programowalnej logiki FPGA. Projekt ten ma na celu przeniesienie obliczeń z warstwy programowej do sprzętowej. Zynq pozwala na stworzenie dedykowanej architektury równoległej, która idealnie pasuje do natury bitonic sort. Kluczowe elementy algorytmu zostaną odwzorowane jako struktury sprzętowe pracujące synchronicznie i deterministycznie. W projekcie przewiduje się integrację części logicznej z systemem

operacyjnym zainstalowanym na płytce rozwojowej Kria KV260, co umożliwi łatwą konfigurację i kontrolę procesu sortowania. Efektem końcowym będzie akcelerator sprzętowy, który może znaleźć zastosowanie w systemach wymagających szybkiego przetwarzania danych.

Wprowadzenie do algorytmy sortującego

Wstęp

Bitonic Sort to algorytm sortowania oparty na tworzeniu tzw. sekwencji bitonicznych (ciągów, które najpierw rosną, a potem maleją (lub odwrotnie)). Algorytm ten w kolejnych etapach dzieli i porównuje elementy, aż uzyska w pełni posortowany ciąg(proces składa się z 6 stanów). Bitconic sort dzięki swojej strukturze dobrze nadaje się do implementacji sprzętowej i potokowej(szczególnie w FPGA). Algorytm ten działa z czasem $O(\log^2 n)$.

Zasada działania

Zasada działania algorytmu opiera się o dwa główne bloki funkcjonalne - minMax oraz Maxmin z których składa się główny moduł sortujący. Ich zadaniem jest porównywanie dwóch 8-bitowych liczb i przypisywanie mniejszej i większej z nich do odpowiednich wyjść. Różnica między nimi polega na kierunku sortowania:

1. maxMin - moduł który jest wykorzystywany w etapach sortowania, w których tworzone są sekwencje rosnące.
2. minMax - moduł który jest wykorzystywany w etapach sortowania, w których tworzone są sekwencje malejące.

Takie podejście pozwala elastycznie budować sekwencje bitoniczne, w których kluczowe jest odpowiednie przemieszczanie danych zgodnie z ustalonym wzorcem porównań i zamian.

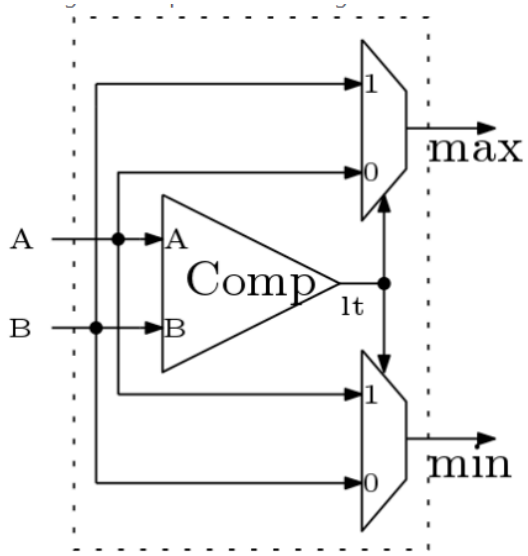


Figure 1: BN block diagram

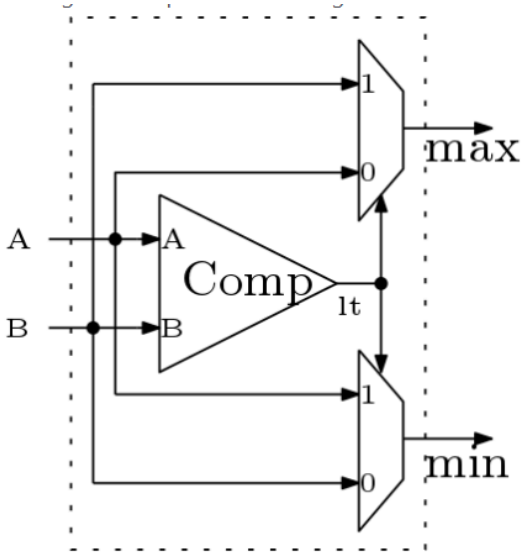


Figure 1: BN block diagram

Mając tak zaprojektowane moduły zaprojektowano główny moduł układu sortujacego zgodnie z schematem

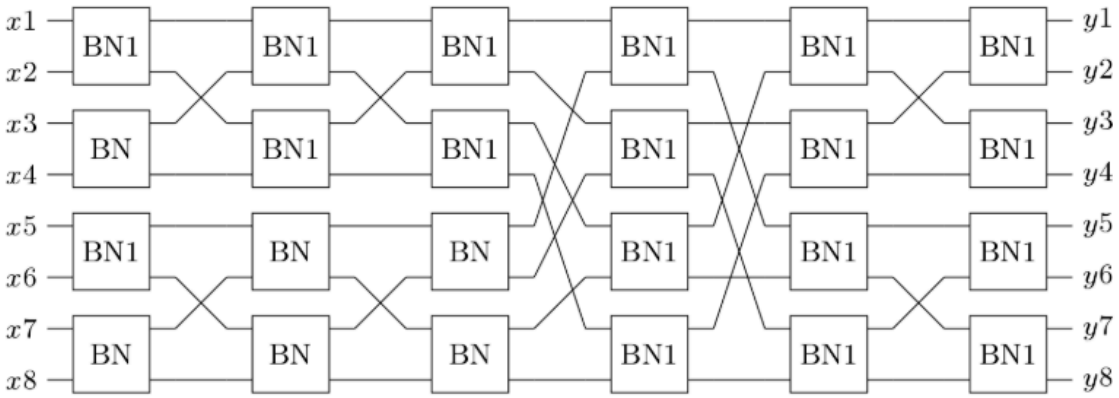
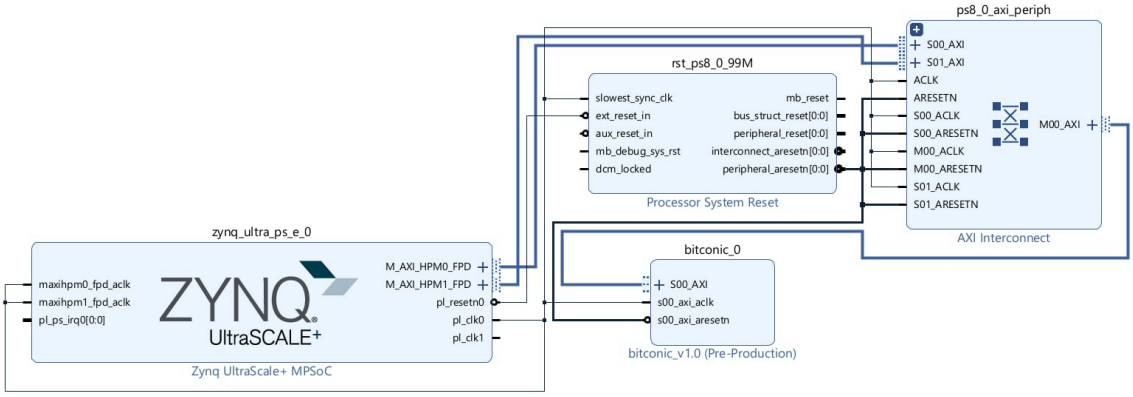


Figure 3: Bitonic Sorter for n=8

Projekt systemu sortującego - integracja z AXI

Kolejnym etapem projektu była integracja zaprojektowanego algorytmu z płytką deweloperską Kria kv260. Połączenie tych dwóch rzeczy możliwe było dzięki wykorzystaniu magistrali AXI Lite, dzięki której możliwa jest komunikacja zaprojektowanego IP Core z Zynq UltraScale+.

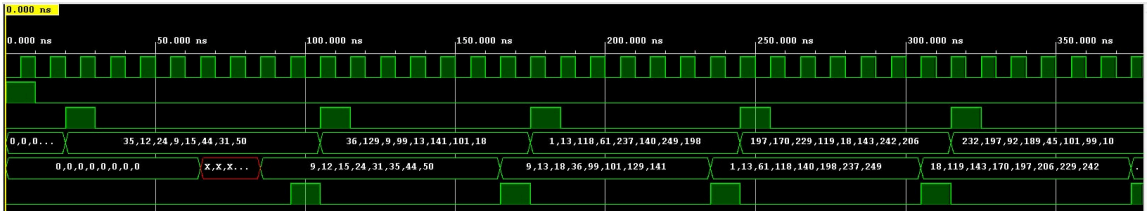
Zaprojektowany z wykorzystaniem środowiska Vivado układ przedstawiono poniżej.



Testy zaprojektowanego systemu

Projekt systemu sortującego - symulacja behawioralna w Vivado

Pierwszym wykonanym testem zaraz po zaprojektowaniu systemu była symualcja behawioralna w Vivado, jej wynik przedstawiono poniżej



Projekt systemu sortującego - symulacja behawioralna w języku Python

Drugim wykonanym testem była implementacja algorytmu bitconic w języku Python. Dzięki takiemu podejściu można łatwo porównać czy sprzętowa implementacja algorytmu przyniosła pozytywne efekty takie jak np. przyśpieszenie obliczeń itp. Zaprojektowany algorytm w języku Python przedstawiono poniżej.

In [1]:
```
def compare_and_swap(arr, i, j, direction):
    if (direction == 1 and arr[i] > arr[j]) or (direction == 0 and arr[i] <
        arr[i], arr[j] = arr[j], arr[i]

def bitonic_merge(arr, low, cnt, direction):
    if cnt > 1:
        k = cnt // 2
        for i in range(low, low + k):
            compare_and_swap(arr, i, i + k, direction)
        bitonic_merge(arr, low, k, direction)
        bitonic_merge(arr, low + k, k, direction)

def bitonic_sort_recursive(arr, low, cnt, direction):
    if cnt > 1:
        k = cnt // 2
        bitonic_sort_recursive(arr, low, k, 1)
        bitonic_sort_recursive(arr, low + k, k, 0)
        bitonic_merge(arr, low, cnt, direction)

def bitonic_sort(arr, ascending=True):
    n = len(arr)
    if n & (n - 1) != 0:
        raise ValueError("Array length should be power of 2.")
    bitonic_sort_recursive(arr, 0, n, 1 if ascending else 0)
    return arr


a = [3, 7, 4, 8, 6, 2, 1, 5]
bitonic_sort(a)
```

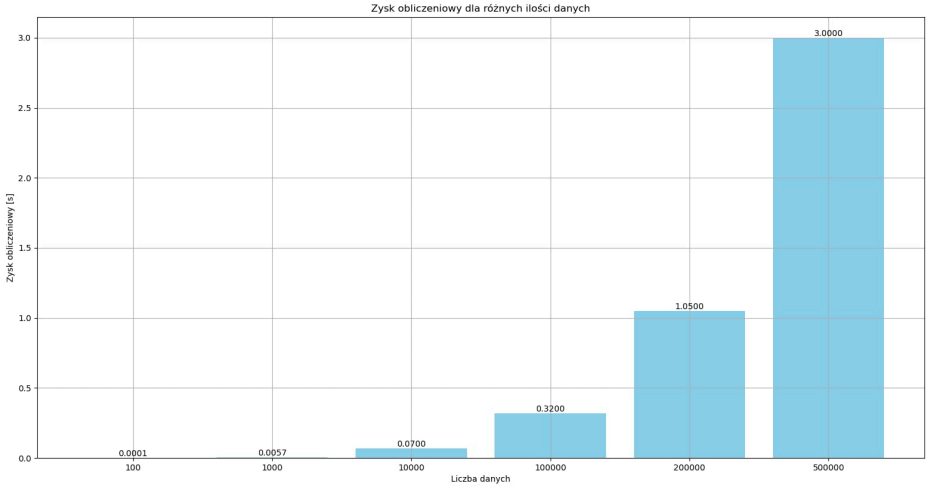
Out[1]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Zysk obliczeniowy algorytmu sortującego

W celu weryfikacji uzyskanych efektów, szczególnie czasu trwania algorytmu dla zadanych ilosci danych przpeorwdzono serię pomiarów których wynik przedstawiono poniżej.

liczba danych	Python [s]	Bitconic [s]	Zysk obliczeniowy[s]
100	0.01397	0.0139	7E-05
1000	0.1297	0.124	0.0057

liczba danych	Python [s]	Bitconic [s]	Zysk obliczeniowy[s]
10000	1.27	1.2	0.07
100000	12.62	12.3	0.32
200000	25.11	24.06	1.05



Integracja z modułem FPGA z wykorzystaniem Jupiter Notebook

Załadowanie modułu:

In [2]:
```
from pynq import Overlay
from time import sleep
import random

kv260_sdup_ov = Overlay("bitconic_v2.xsa")
```

Przetestowanie załączenia:

In [3]:
```
kv260_sdup_ov
```

Out[3]:
```
<pynq.overlay.Overlay at 0xffffa83e3c10>
```

Po poprawnym załączeniu modułu należy przetestować podstawowe działanie

In [4]:
```
kv260_sdup_ov.bitconic_0.write(0, 0x04_03_02_01)
kv260_sdup_ov.bitconic_0.write(4, 0x05_06_07_08)
kv260_sdup_ov.bitconic_0.write(4*4, 1)


i0 = kv260_sdup_ov.bitconic_0.read(4*0)
i1 = kv260_sdup_ov.bitconic_0.read(4*1)
valid = kv260_sdup_ov.bitconic_0.read(4*4)
print(f"i0: {i0}, i1: {i1}, valid: {valid}")
```

```
i0: 67305985, i1: 84281096, valid: 1
```

In [5]:
```
o0 = kv260_sdup_ov.bitconic_0.read(4*2)
o1 = kv260_sdup_ov.bitconic_0.read(4*3)
valid = kv260_sdup_ov.bitconic_0.read(4*5)
print(f"o0: 0x{o0:X}, o1: 0x{o1:X}, valid: {valid}")
```

```
o0: 0x1020304, o1: 0x5060708, valid: 0
```

In [6]:
```
o0 = kv260_sdup_ov.bitconic_0.read(4*2)
o1 = kv260_sdup_ov.bitconic_0.read(4*3)
valid = kv260_sdup_ov.bitconic_0.read(4*5)
print(f"o0: 0x{o0:X}, o1: 0x{o1:X}, valid: {valid}")
```

```
o0: 0x1020304, o1: 0x5060708, valid: 0
```

Jak widać układ sortujący pracuje poprawnie.

Funkcje dodatkowe, poprawiające czytelność danych

In [7]:
```
number = 8

def genRandom():
    L = range(256)

    tab = [random.choice(L) for _ in range(number)]
    # print(f"Input:  {tab}")

    return tab

def encode(tab):
    i0 = 0
    i1 = 0
    for i in range(number//2):
        i0 = (i0 << 8) + tab[i]
        i1 = (i1 << 8) + tab[i + number//2]
    return i0, i1

def decode(o0, o1):
    ou0 = []
    ou1 = []
    for i in range(number//2):
        ou0.append(o0 & 0xFF)
        o0 = o0 >> 8

        ou1.append(o1 & 0xFF)
        o1 = o1 >> 8
    ou0.reverse()
    ou1.reverse()
    tab = [*ou0, *ou1]
    return tab
```

In [8]:
```
def toSort(i0, i1):
    kv260_sdup_ov.bitconic_0.write(4*0, i0)
    kv260_sdup_ov.bitconic_0.write(4*1, i1)
    kv260_sdup_ov.bitconic_0.write(4*4, 1)

def fromSort():
    o0    = kv260_sdup_ov.bitconic_0.read(4*2)
    o1    = kv260_sdup_ov.bitconic_0.read(4*3)
    valid = kv260_sdup_ov.bitconic_0.read(4*5)
    return o0, o1, valid
```

In [9]:
```
tab = genRandom()
i0, i1 = encode(tab)
```

```
print(f"Input: {tab}, {i0, i1}")
```

```
Input: [71, 172, 19, 177, 33, 104, 112, 91], (1202459569, 560492635)
```

In [10]:
```
toSort(i0, i1)
o0, o1, valid = fromSort()
out = decode(o0, o1)
print(f"Output: {out}")
```

```
Output: [19, 33, 71, 91, 104, 112, 172, 177]
```

Test poprawności sortowania dla losowych wektorów

In [11]:
```
for _ in range (20):
    tab = genRandom()
    i0, i1 = encode(tab)

    toSort(i0, i1)
    o0, o1, valid = fromSort()

    hard = decode(o0, o1)

    inArray = decode(i0, i1)
    soft = bitonic_sort(inArray)


    print(f"Output: \n\tHardware: {hard},\n\tSoftware: {soft}")
    if (hard == soft):
        print("Dane posortowane")
    else:
        print("Błąd w sortowaniu")
    print()
```

```
Output:
        Hardware: [25, 39, 101, 110, 159, 160, 224, 239],
        Software: [25, 39, 101, 110, 159, 160, 224, 239]
Dane posortowane

Output:
        Hardware: [15, 47, 157, 166, 189, 192, 205, 254],
        Software: [15, 47, 157, 166, 189, 192, 205, 254]
Dane posortowane

Output:
        Hardware: [15, 16, 87, 88, 120, 146, 199, 200],
        Software: [15, 16, 87, 88, 120, 146, 199, 200]
Dane posortowane

Output:
        Hardware: [4, 64, 68, 71, 89, 169, 188, 253],
        Software: [4, 64, 68, 71, 89, 169, 188, 253]
Dane posortowane

Output:
        Hardware: [8, 24, 31, 48, 54, 85, 182, 229],
        Software: [8, 24, 31, 48, 54, 85, 182, 229]
Dane posortowane

Output:
        Hardware: [0, 12, 75, 84, 112, 135, 135, 238],
        Software: [0, 12, 75, 84, 112, 135, 135, 238]
Dane posortowane

Output:
        Hardware: [19, 61, 72, 74, 178, 240, 246, 250],
        Software: [19, 61, 72, 74, 178, 240, 246, 250]
Dane posortowane

Output:
        Hardware: [61, 98, 105, 110, 131, 163, 186, 255],
        Software: [61, 98, 105, 110, 131, 163, 186, 255]
Dane posortowane

Output:
        Hardware: [6, 31, 50, 58, 98, 131, 133, 249],
        Software: [6, 31, 50, 58, 98, 131, 133, 249]
Dane posortowane

Output:
        Hardware: [36, 55, 76, 103, 151, 158, 182, 215],
        Software: [36, 55, 76, 103, 151, 158, 182, 215]
Dane posortowane

Output:
        Hardware: [13, 24, 41, 132, 160, 205, 221, 255],
        Software: [13, 24, 41, 132, 160, 205, 221, 255]
Dane posortowane

Output:
        Hardware: [37, 107, 149, 169, 203, 206, 225, 229],
        Software: [37, 107, 149, 169, 203, 206, 225, 229]
Dane posortowane

Output:
        Hardware: [25, 60, 68, 83, 117, 186, 216, 237],
        Software: [25, 60, 68, 83, 117, 186, 216, 237]
Dane posortowane
```

```
Output:
        Hardware: [24, 51, 63, 107, 136, 162, 171, 249],
        Software: [24, 51, 63, 107, 136, 162, 171, 249]
Dane posortowane

Output:
        Hardware: [48, 69, 71, 123, 130, 131, 146, 148],
        Software: [48, 69, 71, 123, 130, 131, 146, 148]
Dane posortowane

Output:
        Hardware: [21, 60, 65, 83, 97, 153, 206, 245],
        Software: [21, 60, 65, 83, 97, 153, 206, 245]
Dane posortowane

Output:
        Hardware: [24, 31, 49, 63, 137, 138, 160, 212],
        Software: [24, 31, 49, 63, 137, 138, 160, 212]
Dane posortowane

Output:
        Hardware: [57, 61, 79, 94, 194, 196, 217, 255],
        Software: [57, 61, 79, 94, 194, 196, 217, 255]
Dane posortowane

Output:
        Hardware: [40, 124, 125, 134, 147, 153, 193, 255],
        Software: [40, 124, 125, 134, 147, 153, 193, 255]
Dane posortowane

Output:
        Hardware: [5, 86, 88, 94, 142, 146, 216, 254],
        Software: [5, 86, 88, 94, 142, 146, 216, 254]
Dane posortowane
```

Test dużej ilości danych:

In [12]:
```
from time import perf_counter_ns
count = 100_000
inp = []
i0 = []
i1 = []
outHard = []
outSoft = []

for _ in range(count):
    inp.append(genRandom())
    in0, in1 = encode(inp[-1])
    i0.append(in0)
    i1.append(in1)
```

Pomiar czasu sortowania za pomocą hardware

In [13]:
```
start = perf_counter_ns()
for i in range(count):
    toSort(i0[i], i1[i])
    outHard.append(fromSort())
stop = perf_counter_ns()

print(f"Czas sortowania {count} wektorów za pomocą hardwaru: {(stop - start)
```

```
Czas sortowania 100000 wektorów za pomocą hardwaru: 12.455306447s
```

Pomiar czasu sortowania za pomocą software

In [14]:
```
start = perf_counter_ns()
for i in range(count):
    outSoft.append(bitonic_sort(inp[i]))
stop = perf_counter_ns()

print(f"Czas sortowania {count} wektorów za pomocą softwaru: {(stop - start)
```

```
Czas sortowania 100000 wektorów za pomocą softwaru: 12.567472045s
```

Czas obiegu pętli

In [15]:
```
start = perf_counter_ns()
for i in range(count):
    pass
stop = perf_counter_ns()
print(f"Czas pustej petli {count}: {(stop - start)/1e9}s")
```

```
Czas pustej petli 100000: 0.03556604s
```

Sprawdzenia czy dane zostały posortowane tak samo

In [16]:
```
isOk = True
for i in range(count):
    tab = decode(outHard[i][0], outHard[i][1])
    if (outSoft[i] != tab):
        print("Dane niepoprawnie posortowane")
        print(f"Soft: {outSoft[i]}")
        print(f"Hard: {tab}")
        isOk = False
        break

if isOk:
    print("Poprawnie posortowane")
```

```
Poprawnie posortowane
```

Bibliografia:

- Bitonic Sorter – Digital System Design
- Instrukcja do labolatorium nr. 5 „PYnQ - Wprowadzenie”, autor: Sebastian Koryciak

In [ ]:
```

```