Projekt 2: Sieć Sortująca

Paweł Plewa Tymon Zadara

15 maja 2025

Spis treści

1 Cel projektu **2**

2 Opis zadania **2**

2.1 Logisim-evolution 2

2.2 Intel Quartus Prime Lite 2

3 Realizacja **3**

3.1 Logisim-evolution 3

3.2 Quartus 4

3.2.1 Comp 4

3.2.2 Sn 5

3.2.3 Test 6

3.2.4 Proces Testowania 7

3.2.5 Wave - Szczegółowa Analiza wyników 8

4 Zawartość **14**

5 Wnioski **15**

1 Cel projektu

Celem projektu jest implementacja sieci sortującej. Projekt ma być zrealizowany w dwóch środowiskach:

- Logisim-evolution
- Intel Quartus Prime Lite

Dla naszych numerów indeksu zrealizowaliśmy sieć Hibbard'a $(6 + 7(mod\ 8) + 1 = 6)$. Zakładamy, że sortujemy liczby naturalne od najmniejszej (u góry, wypływa) do największej (na dole, tonie). Zrealizowaną sieć zweryfikowaliśmy na podstawie 6 wektorów o 4-bitowych liczbach:

- 337077
- 337086
- 770733
- 680733
- 777330
- 867330.

2 Opis zadania

2.1 Logisim-evolution

Projekt należy zrealizować wykorzystując element *comparator*, a następnie ułożyć sieć o zadanych parametrach. W ostatnim kroku wykonać test działania sieci za pomocą opcji *Test Vector*.

2.2 Intel Quartus Prime Lite

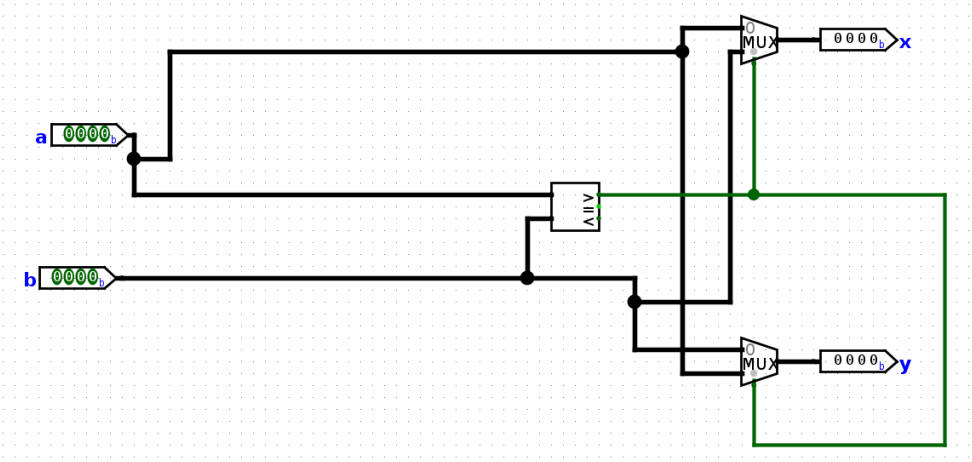
Realizacja części projektu z Quartusa składała się z następujących etapów:

- Stworzenie pliku Comp - pliku zawierającego funkcję komparatora
- Stworzenie pliku Sn - pliku zawierającego skrypt algorytmu sortującego
- Stworzenie pliku Test - pliku testującego sortowanie dla naszych wektorów
- Przetestowanie programu przy użyciu ModelSim

3 Realizacja

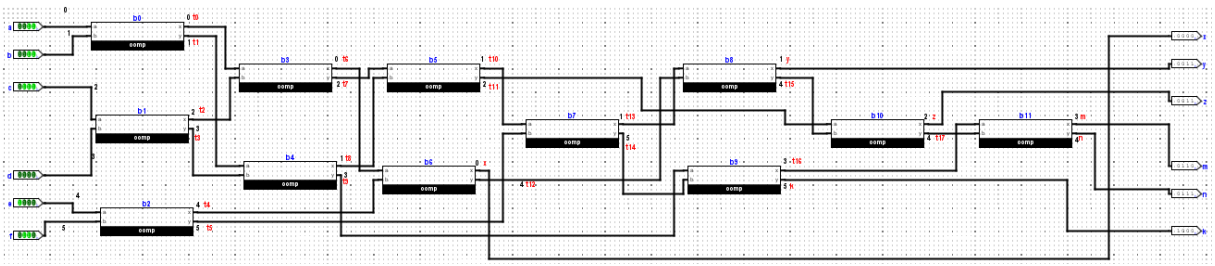
3.1 Logisim-evolution

Zaczęliśmy od wykonania prostego komparatora w następującej postaci:



Rysunek 1: Komparator wykorzystany w sieci sortującej, mniejsza liczba wypływa, większa tonie.

Następnie zgodnie z *algorytmem Hibbard'a* wykonaliśmy poniższą sieć:



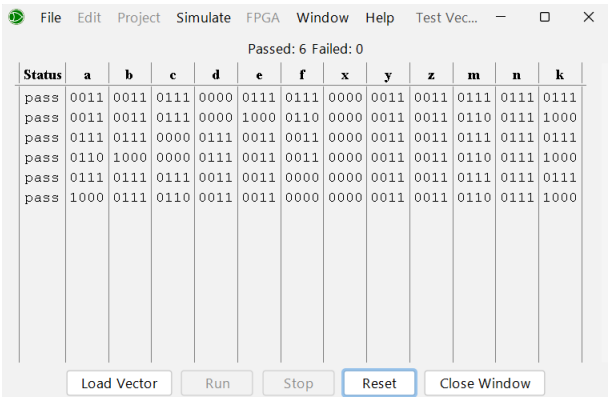
Rysunek 2: Sieć algorytmu Hibbard'a dla 6 cyfr.

Po czym dokonaliśmy testu z wektorami z pliku .txt, o następującej postaci:

a[4]	b[4]	c[4]	d[4]	e[4]	f[4]	x[4]	y[4]	z[4]	m[4]	n[4]	k[4]
0011	0011	0111	0000	0111	0111	0000	0011	0011	0111	0111	0111
0011	0011	0111	0000	1000	0110	0000	0011	0011	0110	0111	1000
0111	0111	0000	0111	0011	0011	0000	0011	0011	0111	0111	0111
0110	1000	0000	0111	0011	0011	0000	0011	0011	0110	0111	1000
0111	0111	0111	0011	0011	0000	0000	0011	0011	0111	0111	0111
1000	0111	0110	0011	0011	0000	0000	0011	0011	0110	0111	1000

Tabela 1: Zawartość pliku .txt.

Otrzymując jedyny możliwy efekt:



Rysunek 3: Wynik testowania na danych z pliku .txt.

3.2 Quartus

3.2.1 Comp

Plik *comp.v* (1) zawiera kod stanowiący funkcję komparatora, podstawę algorytmu sortującego. Plik comp.v został podany w danych plikach do tego projektu i nie wymagał wprowadzenia żadnych zmian.

Listing 1: Moduł *comp* w Verilog, plik *comp.v*

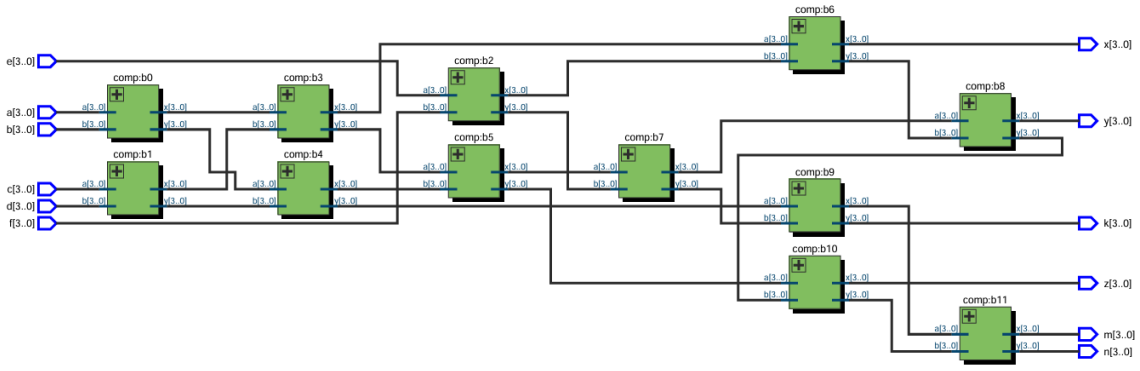
```
module comp
(
        input [3:0] a,b,
        // input [3:0] b,
        output [3:0] x,y
);

assign x = (a > b) ? b : a;
assign y = a > b ? a : b;

endmodule
```

3.2.2 Sn

Plik *sn.v* (2) zawiera kod stanowiący algorytm sortowania Hibbarda dla wektorów o długości 6 znaków. Kod można podzielić na 3 części: deklaracja zmiennych wejściowych oraz wyjściowych, deklaracja zmiennych stanowiących połączenia między komparatorami (*"wire"*) oraz algorytm sortowania składający się z 12 komparatorów.
Plik ten wymagał zmian w stosunku do oryginalnego. Aby plik poprawnie spełniał swoje założenia należało dodać 2 zmienne wejściowe oraz wyjściowe (*"In(e,f), Out(n,k)"*, 12 kabli (*"wire"*) oraz 7 komparatorów.

Listing 2: Moduł *sn* w Verilog, plik *sn.v*

```
module sn
(
        input [3:0] a,b,c,d,e,f
        output [3:0] x,y,z,m,n,k
);

wire [3:0] t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,t14,t15,t16,t17;

comp b0 (.a(a), .b(b), .x(t0), .y(t1));
comp b1 (.a(c), .b(d), .x(t2), .y(t3));
comp b2 (.a(e), .b(f), .x(t4), .y(t5));

comp b3 (.a(t0), .b(t2), .x(t6), .y(t7));
comp b4 (.a(t1), .b(t3), .x(t8), .y(t9));

comp b5 (.a(t7), .b(t8), .x(t10), .y(t11));
comp b6 (.a(t6), .b(t4), .x(x), .y(t12));

comp b7 (.a(t10), .b(t5), .x(t13), .y(t14));

comp b8 (.a(t13), .b(t12), .x(y), .y(t15));
comp b9 (.a(t9), .b(t14), .x(t16), .y(k));

comp b10 (.a(t11), .b(t15), .x(z), .y(t17));

comp b11 (.a(t16), .b(t17), .x(m), .y(n));


endmodule
```



Rysunek 4: Schemat algorytmu z *sn.v*.

3.2.3 Test

Plik *test.do* (3) zawiera kod stanowiący algorytm testujący działanie naszego algorytmu sortowania (*"sn.v"*).
Plik zawiera taką samą składnię jak podany w projekcie. Dodane zostały 4 wektory do sprawdzenia, a długość wektorów zostałą wydłużona z 4 do 6 elementów.

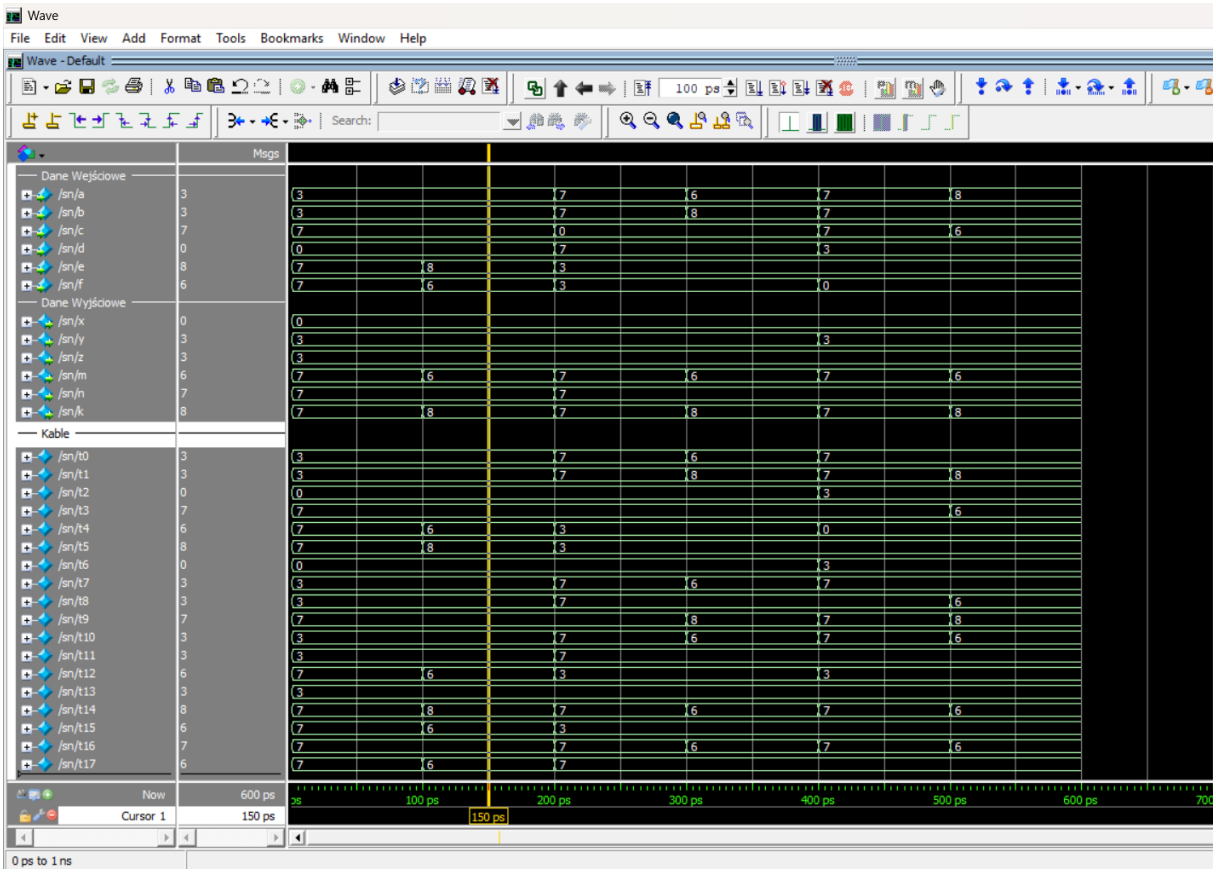
Listing 3: Skrypt testujący w ModelSim

```
restart -force -nowave
add wave -radix unsigned *
force a 10#3 0
force b 10#3 0
force c 10#7 0
force d 10#0 0
force e 10#7 0
force f 10#7 0
run
force a 10#3 0
force b 10#3 0
force c 10#7 0
force d 10#0 0
force e 10#8 0
force f 10#6 0
run
force a 10#7 0
force b 10#7 0
force c 10#0 0
force d 10#7 0
force e 10#3 0
force f 10#3 0
```

```
run
force a 10#6 0
force b 10#8 0
force c 10#0 0
force d 10#7 0
force e 10#3 0
force f 10#3 0
run
force a 10#0 0
force b 10#3 0
force c 10#3 0
force d 10#7 0
force e 10#7 0
force f 10#7 0
run
force a 10#0 0
force b 10#3 0
force c 10#3 0
force d 10#6 0
force e 10#7 0
force f 10#8 0
run
```

3.2.4 Proces Testowania

Aby przetestować nasze wektory (1) została odpalona symulacja w programie ModelSim. Wywołany został plik *test.do* dla programu *sn.v*.
Oto wyniki otrzymane w symulacji 5:



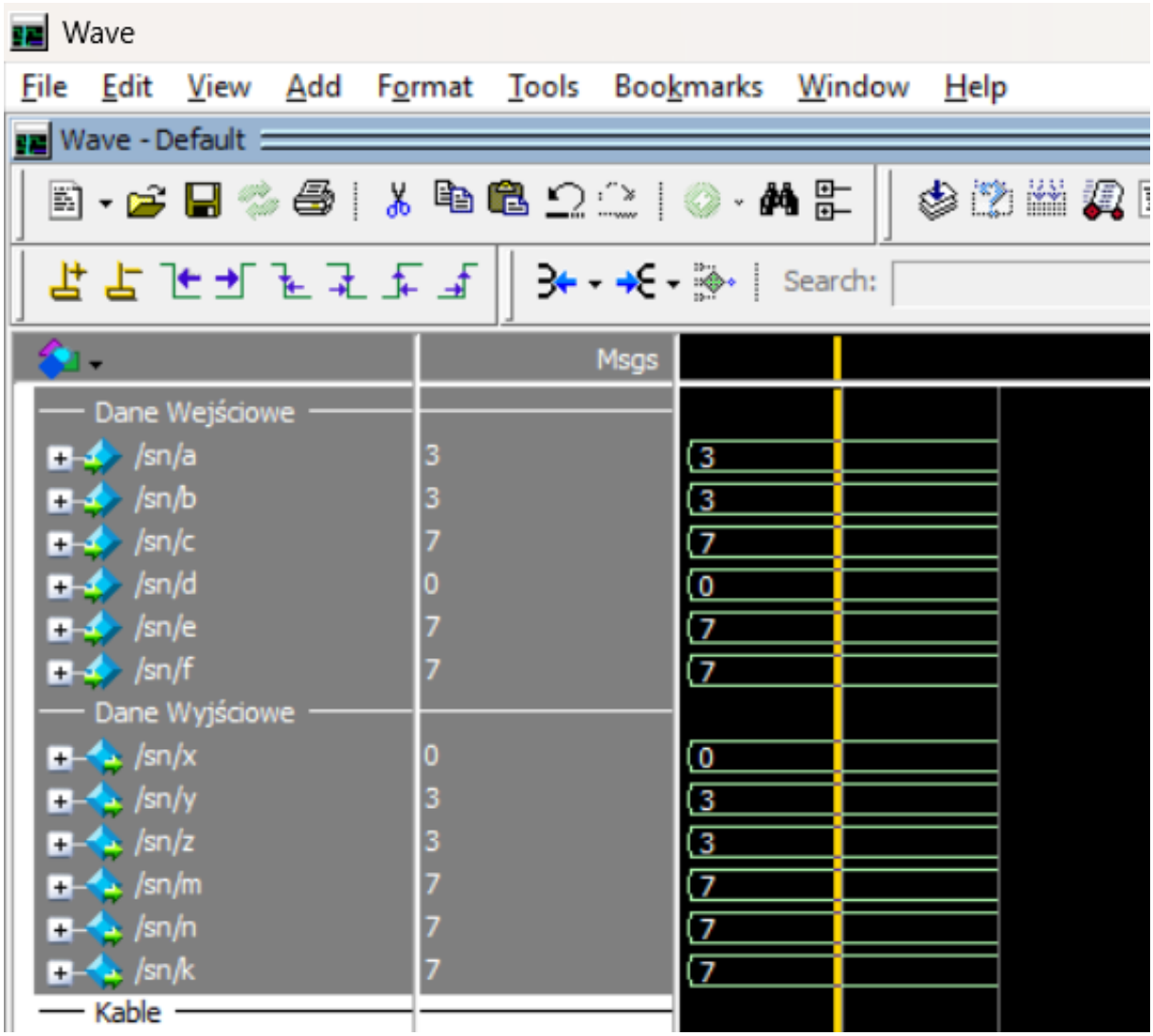
Rysunek 5: Wynik symulacji skryptu test.do dla ModelSim w Quartus.

Na rysunku 5 widać wartości wywoływane oraz wartości zwracane przez program. Pierwsze 6 wierszy to reprezentacja danych wejściowych od a do f. Kolejne 6 wierszy (nie licząc wiersza separującego) to reprezentacja danych wejściowych, czyli danych po przejściu przez nasz algorytm sortujących.
UWAGA! Nie w każdej kolumnie danych występują dane w każdym wierszu. Jeżeli w danej kolumnie w danym wierszu nie występuje dana, oznacza to, że w tym miejscu arguemnt przyjmuję taką samą wartość jak przy ostatniej zmianie.
Po analizie końcowego wyniku można stwierdzić, że algorytm posortował elementy każdego wektora prawidłowo.

3.2.5 Wave - Szczegółowa Analiza wyników

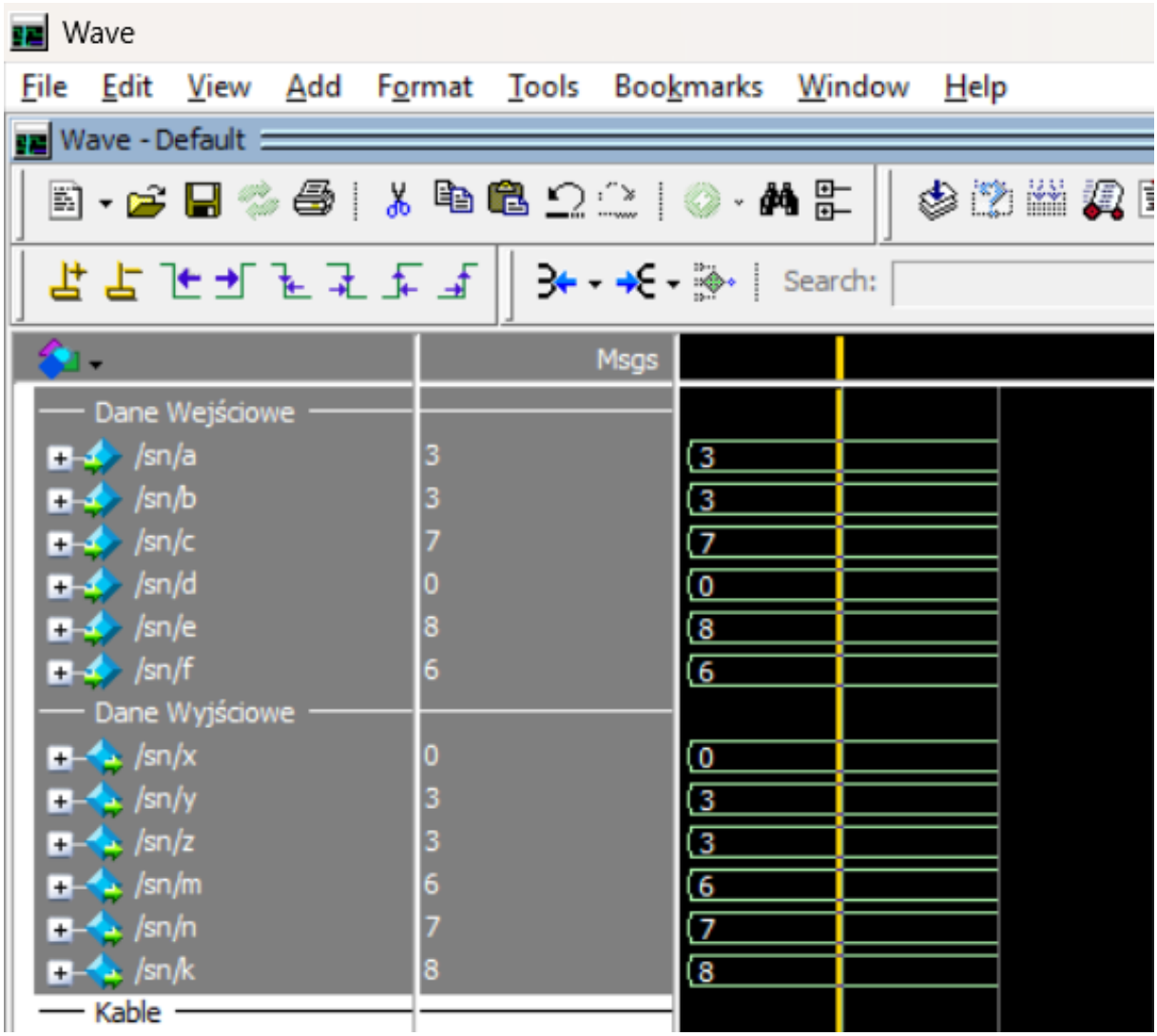
W tej części znajduje się szczegółowa analiza wyników sortowania przez nasz algorytm w Wave programu ModelSim. W przypadku zrozumienia wyników już w poprzednim punkcie można pominąć tą część. Link do spisów: 5.

- Wektor 1:



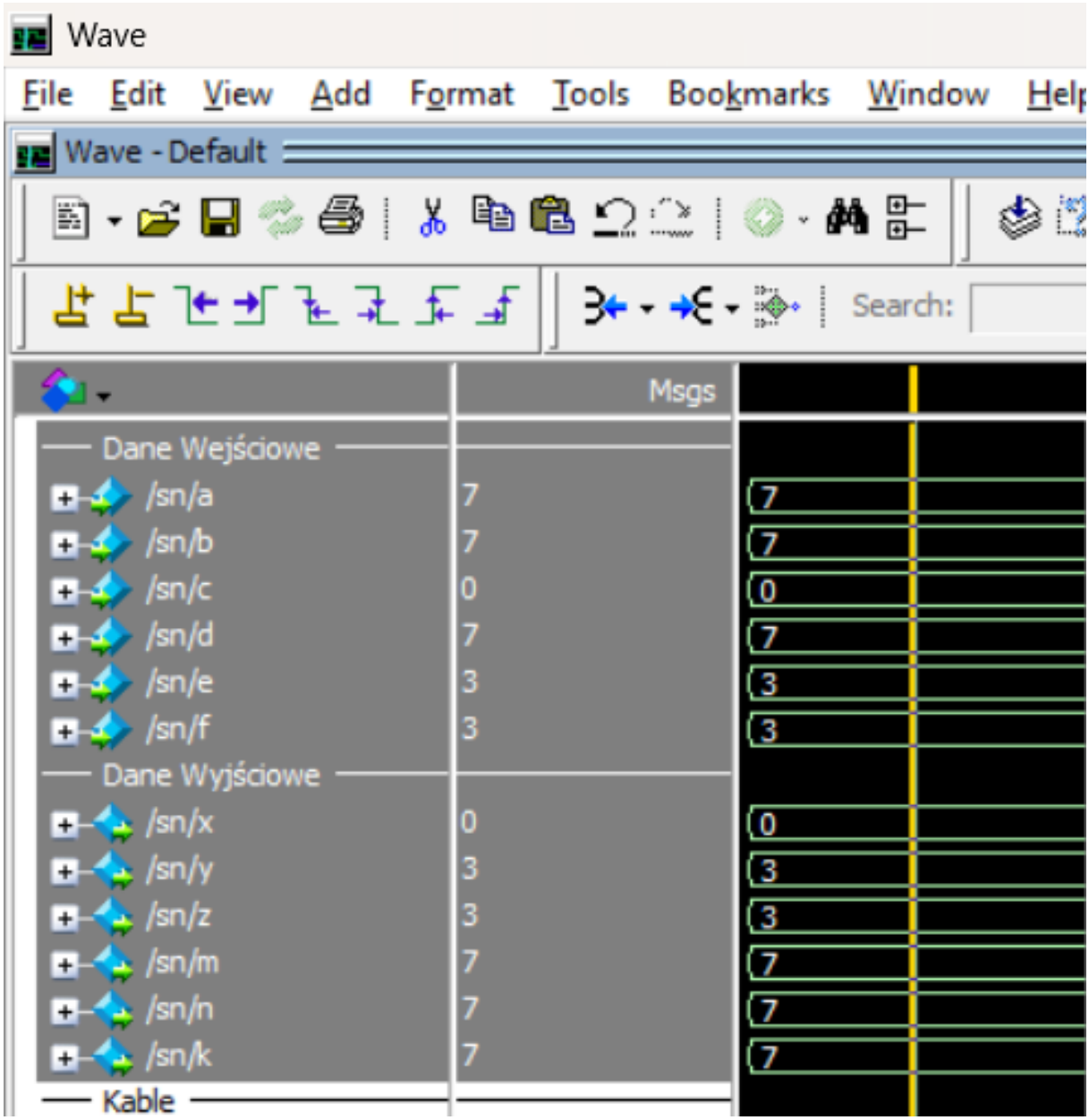
Rysunek 6: Wynik symulacji wektora *337077* dla ModelSim w Quartus.

- Wektor 2:



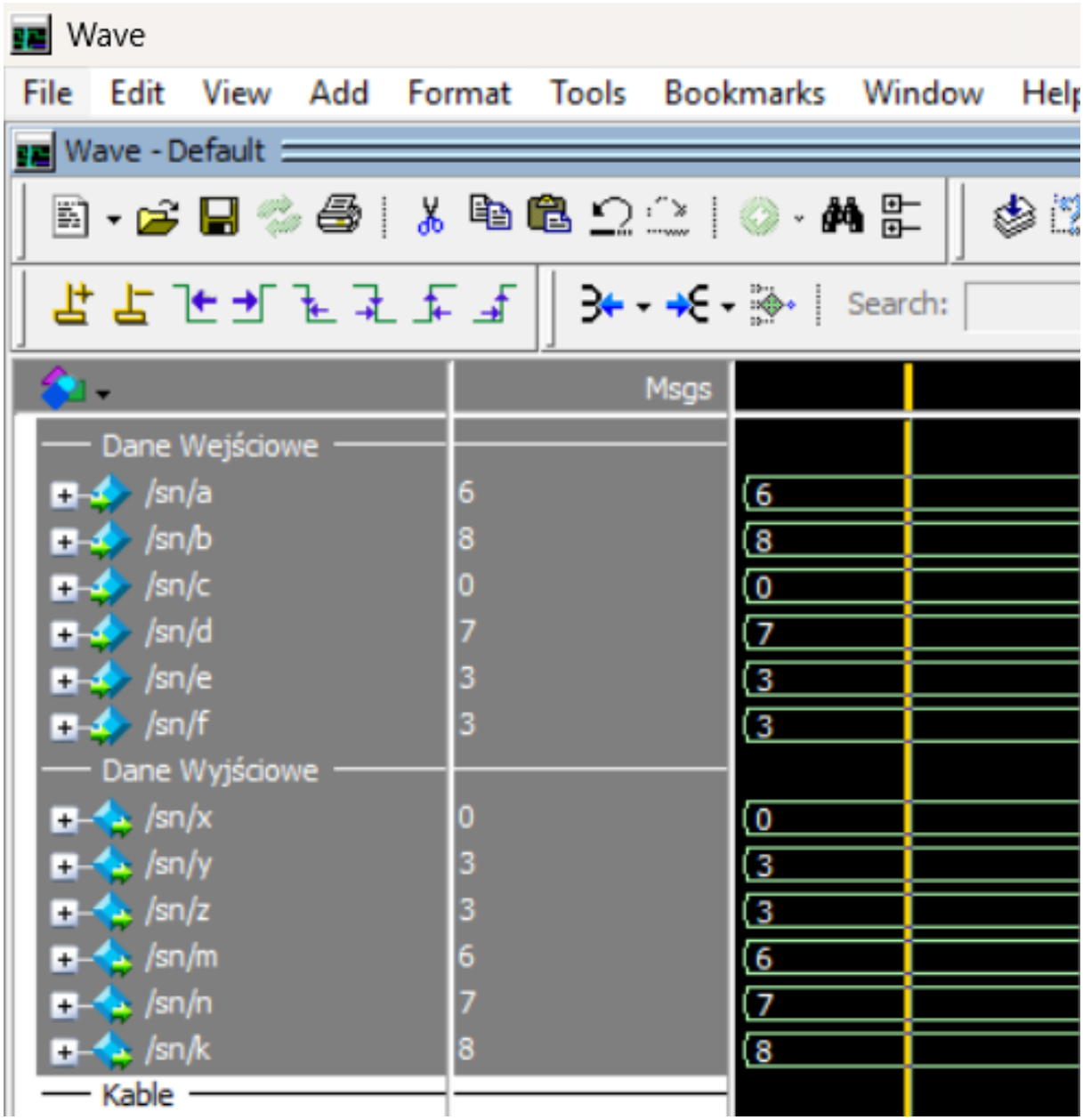
Rysunek 7: Wynik symulacji wektora *337086* dla ModelSim w Quartus.

- Wektor 3:



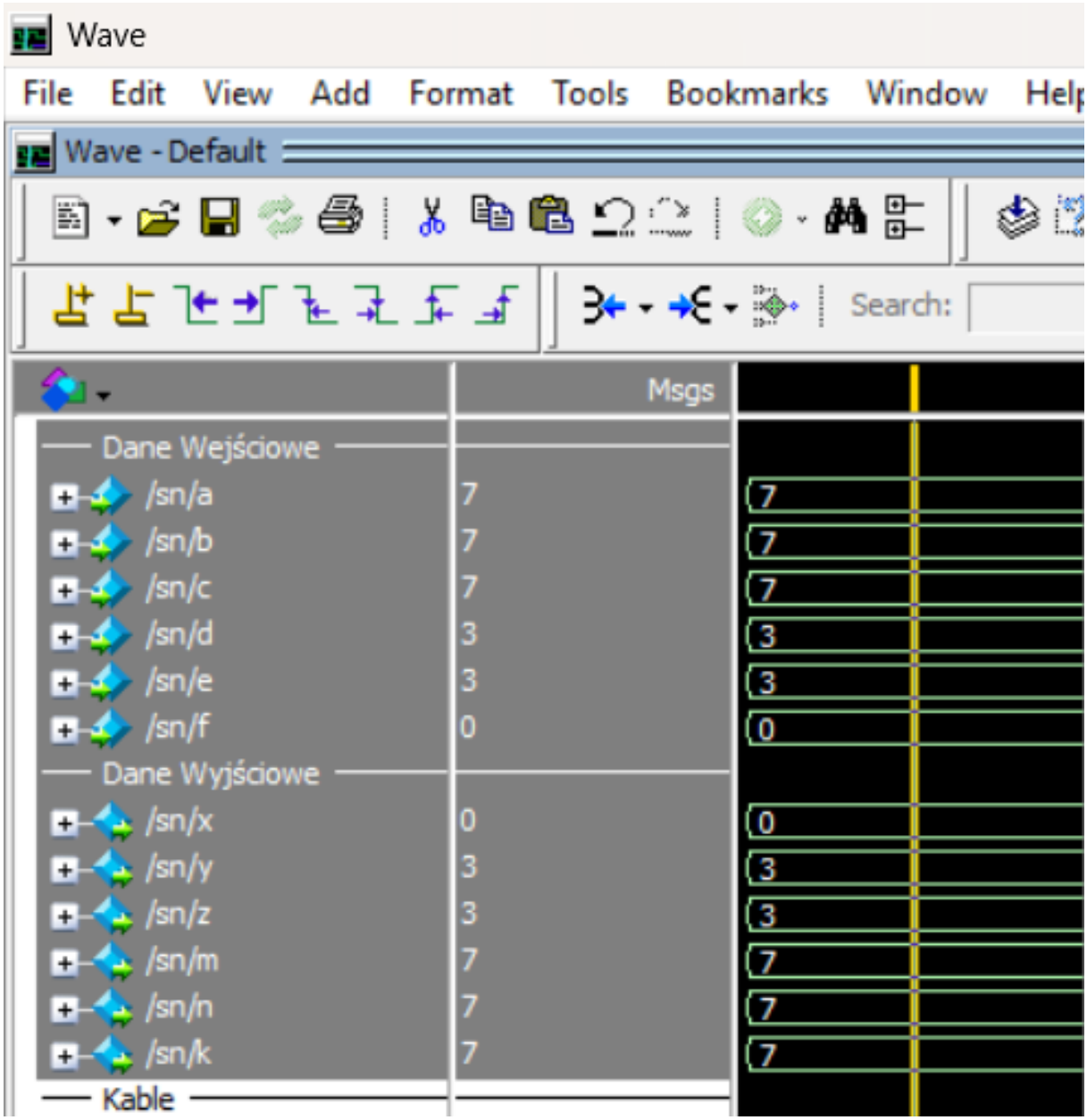
Rysunek 8: Wynik symulacji wektora *770733* dla ModelSim w Quartus.

- Wektor 4:



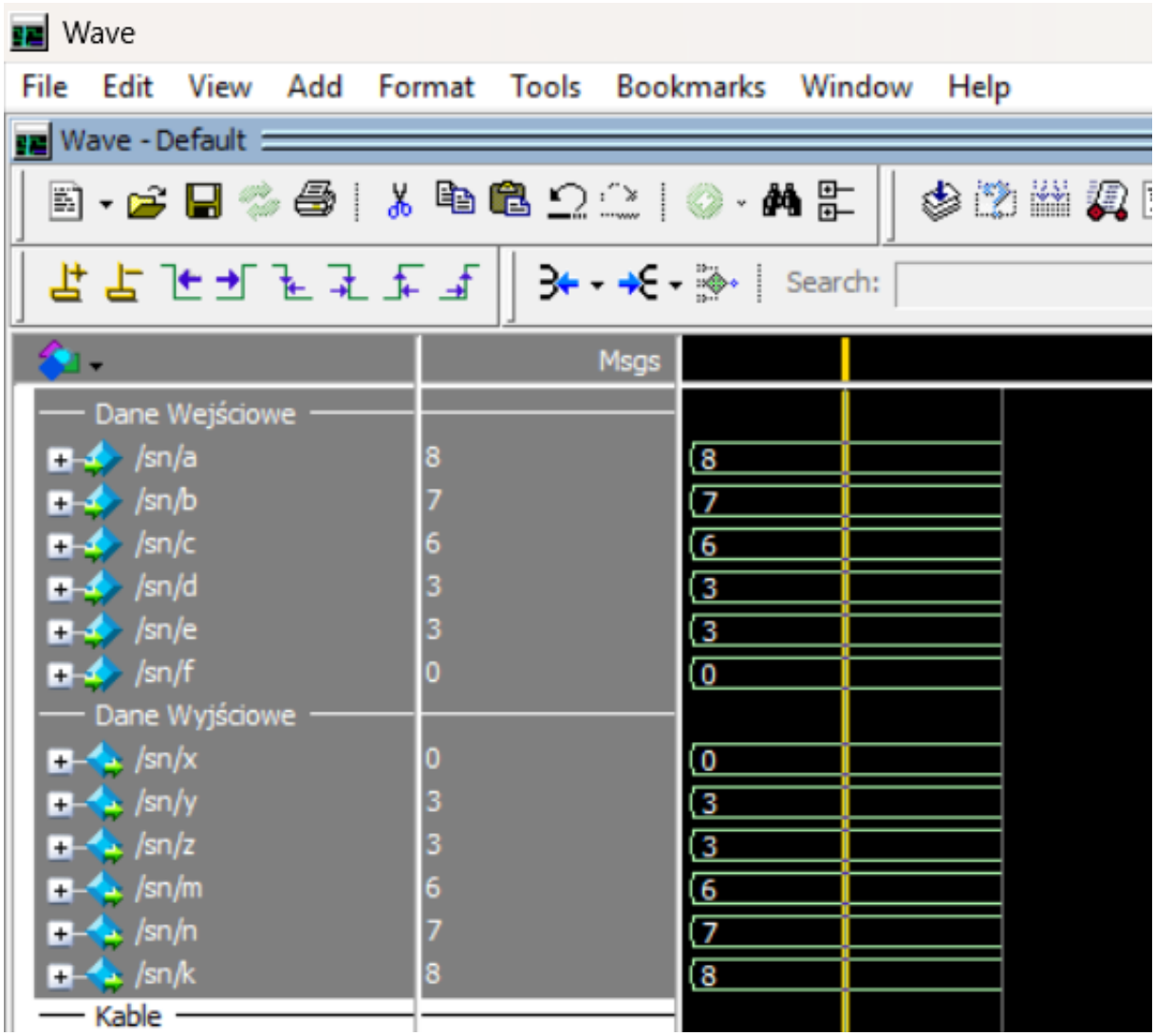
Rysunek 9: Wynik symulacji wektora *680733* dla ModelSim w Quartus.

- Wektor 5:



Rysunek 10: Wynik symulacji wektora *777330* dla ModelSim w Quartus.

- Wektor 6:



Rysunek 11: Wynik symulacji wektora *876330* dla ModelSim w Quartus.

Jak widać dla każdego z wektorów otrzymujemy posortowany wynik co potwierdza poprawne działanie naszego algorytmu na rysunku 5.

4 Zawartość

W folderze zip - *POTEC-25L-proj.2-PP-TZ* - znajdują się następujące pliki:

- *comp.v* - plik komparatora do quartus
- *sn.v* - plik programu realizującego sortowanie Hibbarda do quartus
- *test.do* - plik skryptu testującego ModelSim do quartus
- *POTEC_PLEWA_ZADARA_VEC.txt* - plik z wektorami testującymi do Logisim

- *POTEC_PLEWA_ZADARA2.circ* - projekt Logisim
- *Testowanie_quartus_film* - film z symulacja testowania działania algorytmu wektorami w ModelSimie
- *POTEC-25L-proj.2-PP-TZ* - sprawozdanie z projektu

5 Wnioski

Zrealizowana implementacja sieci sortującej według algorytmu Hibbarda okazała się poprawna i skuteczna. W obu środowiskach – *Logisim-evolution* oraz *Intel Quartus Prime Lite z ModelSim* – uzyskaliśmy zgodne wyniki, które potwierdzają poprawność działania naszej sieci.
Podczas pracy nad projektem zdobyliśmy praktyczne doświadczenie zarówno w tworzeniu układów logicznych z wykorzystaniem komparatorów, jak i w implementacji algorytmów sortujących w języku Verilog. Udało nam się również przećwiczyć testowanie algorytmów przy użyciu ModelSima.
Analiza wyników symulacji dla różnych zestawów danych wejściowych(wektorów) wykazała, że sieć niezawodnie sortuje liczby zgodnie z założeniem – od najmniejszej do największej.
Projekt wykazał, że zastosowanie algorytmów sortujących w formie sprzętowej (sieciowej) jest możliwe i efektywne. Algorytm Hibbarda jest skutecznym algorytmem. W porównaniu do pozostałych algorytmów wykorzystywanych przez inne grupy w tym projekcie, jest on szybkim algorytmem wykorzystującym małą ilość komparatorów(12 komparatorów dla 6 wejść).

Spis rysunków

1 Komparator wykorzystany w sieci sortującej, mniejsza liczba wypływa, większa tonie. 3
2 Sieć algorytmu Hibbard'a dla 6 cyfr. 3
3 Wynik testowania na danych z pliku .txt. 4
4 Schemat algorytmu z *sn.v*. 6
5 Wynik symulacji skryptu test.do dla ModelSim w Quartus. 8
6 Wynik symulacji wektora *337077* dla ModelSim w Quartus. 9
7 Wynik symulacji wektora *337086* dla ModelSim w Quartus. 10
8 Wynik symulacji wektora *770733* dla ModelSim w Quartus. 11
9 Wynik symulacji wektora *680733* dla ModelSim w Quartus. 12

10 Wynik symulacji wektora *777330* dla ModelSim w Quartus. 13

11 Wynik symulacji wektora *876330* dla ModelSim w Quartus. 14

Spis tabel

1 Zawartość pliku .txt. 4

Listings

1 Moduł *comp* w Verilog, plik *comp.v* 4

comp.v 4

2 Moduł *sn* w Verilog, plik *sn.v* 5

sn.v 5

3 Skrypt testujący w ModelSim 6

test.do 6