Jan Czechowski

Kinga Konieczna

Projekt 2 - Sieć sortująca

13 maja 2025

Spis treści

1. Cel projektu

Projekt ma na celu **zaprojektowanie i zaimplementowanie cyfrowej sieci sortującej**, umożliwiającej uporządkowanie zbioru liczb naturalnych w porządku rosnącym. Realizacja zostanie przeprowadzona w dwóch środowiskach symulacyjno-projektowych:

— **Logisim-evolution** – umożliwiającym wizualne modelowanie obwodów cyfrowych,
— **Intel Quartus Prime Lite** (wersja 20.1.1) – środowisku dedykowanym projektowaniu układów cyfrowych w języku HDL.

Celem jest odwzorowanie wybranej struktury sieci sortującej o określonej liczbie wejść, zgodnie z jej nazwanym wariantem. W celu weryfikacji poprawności działania zaprojektowanego układu będziemy testować układy poprzez podanie 6 różnych wektorów testowych.

Dla potrzeb projektu przyjmujemy konwencję, w której mniejsze liczby „unoszą się" ku górze wyjść, natomiast większe „toną" w dół – czyli dane są sortowane od najmniejszej na górze do największej na dole.

2. Wyznaczenie wskaźnika

Indeksy:

— Jan Czechowski – 337066
— Kinga Konieczna – 337072

Najmłodsze cyfry indeksów to odpowiednio 6 i 2.
Sumujemy je:

$$6 + 2 = 8$$

Poddajemy operacji modulu 8:

$$8 \equiv_8 0$$

Dodajemy 1:

$$X = 0 + 1$$

$$\mathbf{X = 1}$$

Otrzymany wynik, czyli nasz wskaźnik do zbioru, to **1**.
Oznacza to, że będziemy realizować sieć sortującą **algorytmem *Balanced***.

3. Wyznaczenie wektorów testujących

Indeksy:

— Jan Czechowski – 337066
— Kinga Konieczna – 337072

Zatem **wektory** to:
1 wektor - 2 wektor: 3,3,7,0,6,6 - 3,3,7,0,7,2
3 wektor - 4 wektor: 6,6,0,7,3,3 - 2,7,0,7,3,3
5 wektor - 6 wektor: 7,6,6,3,3,0 - 7,7,3,3,2,0

4. Metoda działania algorytmu *Balanced*

Algorytm *Balanced* to deterministyczna sieć sortująca, która porządkuje ciąg liczb przy użyciu stałej sekwencji porównań (komparatorów), niezależnie od danych wejściowych. Jest szczególnie efektywny dla małych rozmiarów danych (np. $N = 6$, jak w przykładzie).

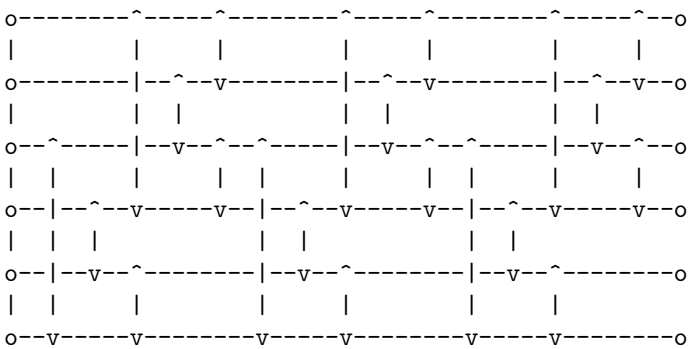
Kluczowe cechy:

— **Stała liczba komparatorów**: Sieć zawiera z góry określoną liczbę komparatorów, które wykonują się w ustalonej kolejności.
— **Równoległe wykonywanie operacji**: Komparatory są zgrupowane w operacje równoległe, co umożliwia ich jednoczesne wykonanie.
— **Efektywność dla małych** N: Dla $N = 6$ algorytm *Balanced* jest zoptymalizowany pod kątem minimalnej liczby komparatorów i głębokości.

Aby gwarantować pełne posortowanie dowolnego ciągu 6-elementowego, niezależnie od początkowego ułożenia, należy zaimplementować sieć, w której będą porównywane w kolejnych krokach algorytmu dane indeksy elementów:

Krok 1: $[2, 5]$, $[3, 4]$
Krok 2: $[0, 3]$, $[1, 2]$, $[4, 5]$
Krok 3: $[0, 1]$, $[2, 3]$
Krok 4: $[2, 5]$, $[3, 4]$
Krok 5: $[0, 3]$, $[1, 2]$, $[4, 5]$
Krok 6: $[0, 1]$, $[2, 3]$
Krok 7: $[2, 5]$, $[3, 4]$
Krok 8: $[0, 3]$, $[1, 2]$, $[4, 5]$
Krok 9: $[0, 1]$, $[2, 3]$

Poniżej znajduje się graf, przedstawiający sieć o rozmiarze 6 z użyciem tego algorytmu:

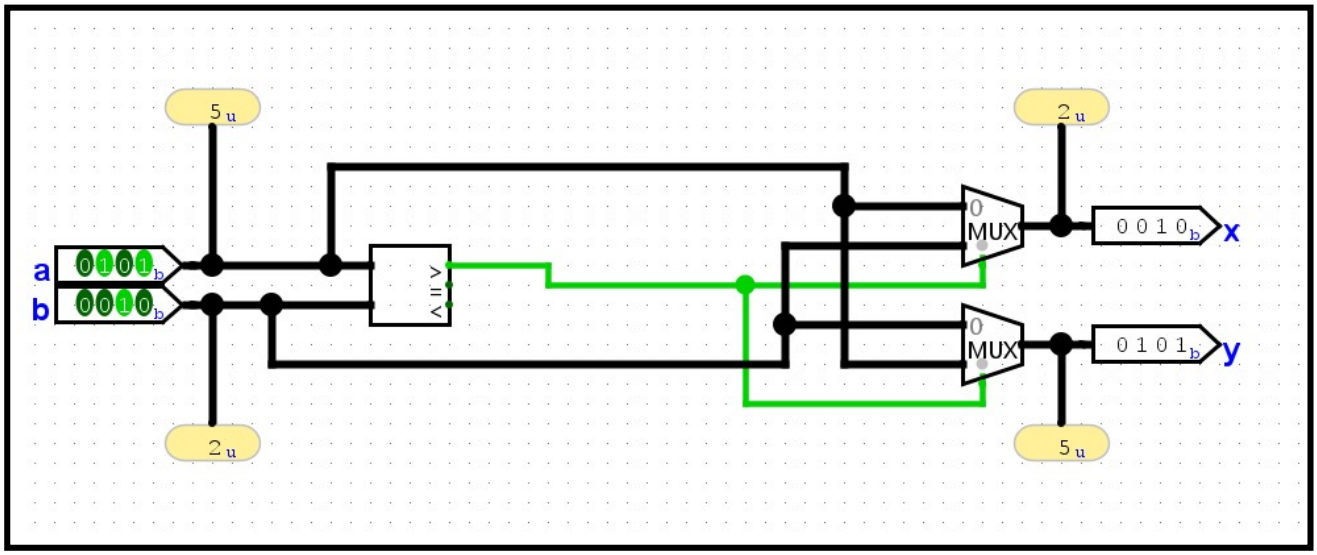


5. Układ w programie Logisim

5.1. Budowa bloku komparatora

Zaprojektowany blok komparatora *comp* przedstawiony na **Rys. 1.** został zrealizowany z wykorzystaniem zestawu podstawowych modułów logicznych:

— **Blok danych wejściowych** – dwie liczby 4-bitowe na wejściu zapisane w systemie binarnym,
— **Moduł porównujący (komparator)** – porównuje dwie dane, a następnie zwraca wynik. Aby układ działał poprawnie, należy w konfiguracji modułu komparatora ustawić typ danych na „Unsigned" (bez znaku). Pozostawienie domyślnego ustawienia „2's Complement" sprawi, że 4-bitowe wartości zaczynające się od „1" będą traktowane jako liczby ujemne, co zakłóciłoby proces sortowania,
— **Moduł przełączający (multiplekser)** – realizuje przełączanie ścieżek sygnałowych na podstawie wyników porównań, umożliwiając dynamiczne sterowanie przepływem danych w obrębie sieci,
— **Blok danych wyjściowych** – dwie liczby 4-bitowe na wyjściu zapisane w systemie binarnym, uporządkowanie rosnąco.

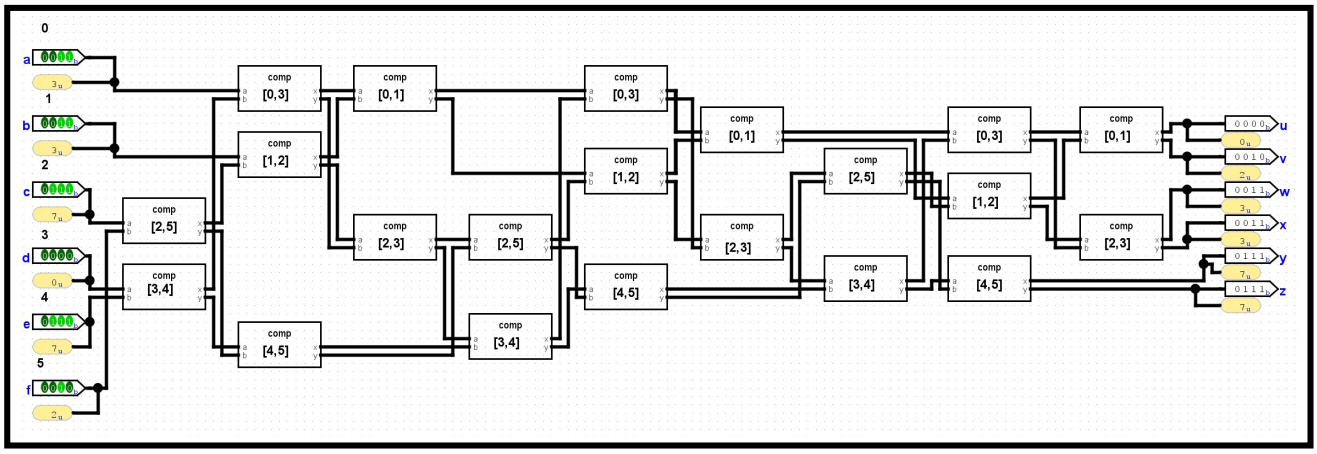


Rys. 1: Budowa komparatora w programie Logisim.

5.2. Budowa układu głównego sieci sortującej

Zaprojektowany układ główny sieci sortującej przedstawiony na **Rys. 2.** został zrealizowany z wykorzystaniem danych modułów logicznych:

— **Blok komparatora *comp*** – odpowiada za analizę dwóch liczb równocześnie. Wyznacza spośród nich wartość większą i mniejszą, co umożliwia poprawne przekierowanie danych do dalszych etapów sortowania.
— **Blok wejściowy** – przyjmuje sześć liczb naturalnych, z których każda jest reprezentowana w formacie czterobitowym. Wartości te stanowią dane wejściowe sieci sortującej.
— **Blok wyjściowy** – odpowiada za prezentację końcowego rezultatu działania układu, czyli uporządkowanego zbioru liczb. Dane te są posortowane rosnąco, zgodnie z konwencją: od najmniejszej liczby (na górze) do największej (na dole).



Rys. 2: Układ sieci sortującej *Balanced* w programie Logisim.

5.3. Testowanie przy użyciu wektorów

W celu umożliwienia automatycznego testowania poprawności działania zaprojektowanej sieci sortującej, należy przygotować plik tekstowy z rozszerzeniem `.txt`, zawierający zestaw wektorów testowych. Plik powinien uwzględniać zarówno dane wejściowe, jak i oczekiwane dane wyjściowe, zapisane w jednolitym formacie.

Struktura pliku:

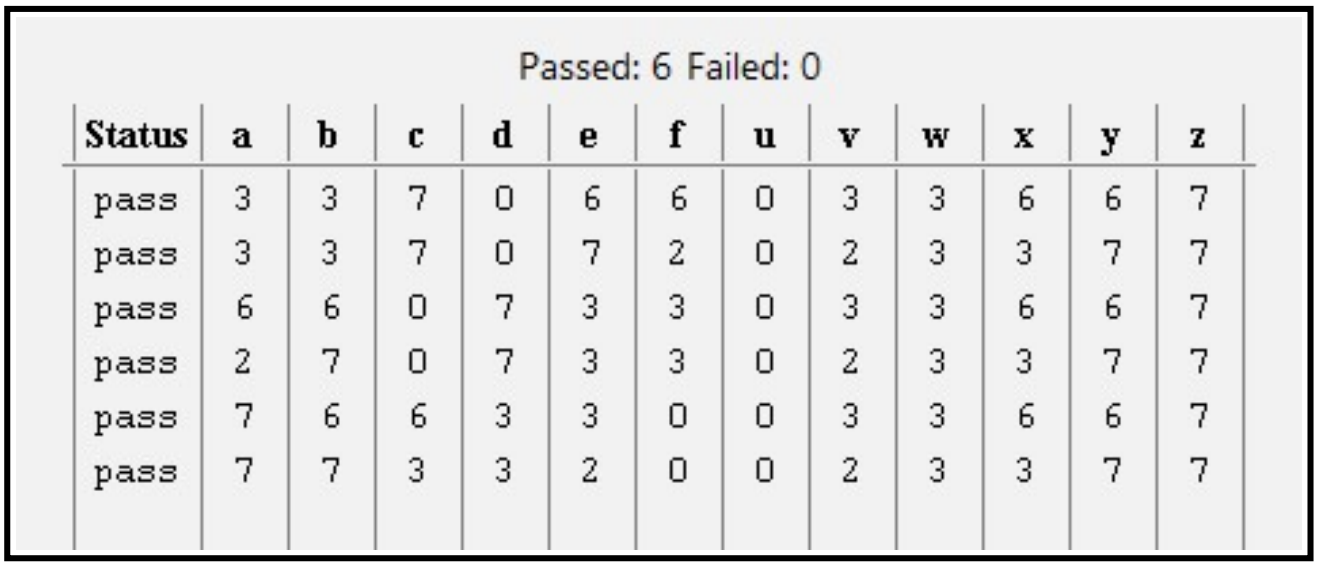
— **Nagłówek wejściowy:** dla każdego sygnału wejściowego należy podać jego nazwę oraz długość słowa w bitach, ujętą w nawiasy kwadratowe, np. `a[4]`. Poszczególne sygnały oddziela się spacją.
— **Nagłówek wyjściowy:** ma identyczną postać jak nagłówek wejściowy, przy czym odnosi się do sygnałów wyjściowych.
— **Wiersze testowe:** każda linia odpowiada jednemu przypadkowi testowemu. Najpierw podaje się wartości na wejściach, a następnie — po odpowiedniej liczbie kolumn — oczekiwane wyniki na wyjściach. Wszystkie wartości są liczbami dziesiętnymi i są rozdzielane spacjami.

W **Tab. 1.** przedstawiliśmy zestaw naszych danych wejściowych dla testów sieci sortującej w programie Logisim Evolution.

Tab. 1: Zestaw danych wejściowych dla testów sieci sortującej.

a[4]	b[4]	c[4]	d[4]	e[4]	f[4]	u[4]	v[4]	w[4]	x[4]	y[4]	z[4]
3	3	7	0	6	6	0	3	3	6	6	7
3	3	7	0	7	2	0	2	3	3	7	7
6	6	0	7	3	3	0	3	3	6	6	7
2	7	0	7	3	3	0	2	3	3	7	7
7	6	6	3	3	0	0	3	3	6	6	7
7	7	3	3	2	0	0	2	3	3	7	7

Na **Rys. 3.** poniżej, przedstawiliśmy wyniki testów wektorów przy użyciu narzędzia *Simulate:Test Vector/Load Vector* na przykładzie powyższych danych z pliku `.txt`.



Rys. 3: Wyniki testów przy użyciu wektorów w programie Logisim.

Jak widać, zrealizowany układ zwraca posortowane rosnąco wartości dla podanych wektorów - zwrócone wartości przedstawiliśmy w **Tab. 2.** poniżej.

Tab. 2: Wektory wejściowe i ich odpowiedniki posortowane rosnąco.

Wektor wejściowy	**Wektor posortowany**
3, 3, 7, 0, 6, 6	0, 3, 3, 6, 6, 7
3, 3, 7, 0, 7, 2	0, 2, 3, 3, 7, 7
6, 6, 0, 7, 3, 3	0, 3, 3, 6, 6, 7
2, 7, 0, 7, 3, 3	0, 2, 3, 3, 7, 7
7, 6, 6, 3, 3, 0	0, 3, 3, 6, 6, 7
7, 7, 3, 3, 2, 0	0, 2, 3, 3, 7, 7

5.4. Symulacja układu

Link do filmiku z nagraną symulacją z programu Logisim: `https://youtu.be/XUwBYGDioRU`.

Przykładowe dane wejściowe użyte w filmiku z symulacją to nasze numery indeksu (337072 i 337066) oraz data założenia WEiTI, czyli 1 października 1951 r. (011051).

6. Układ w programie Intel Quartus Prime

6.1. Komparator

Tak jak w programie Logisim, na początku należy stworzyć moduł komparatora. Implementacja komparatora *comp* została przedstawiona na **Wydruku 1.** poniżej.

Wydruk 1: Moduł `comp` w Verilog, plik `.v`

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

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

endmodule
```

6.2. Moduł główny sieci sortującej

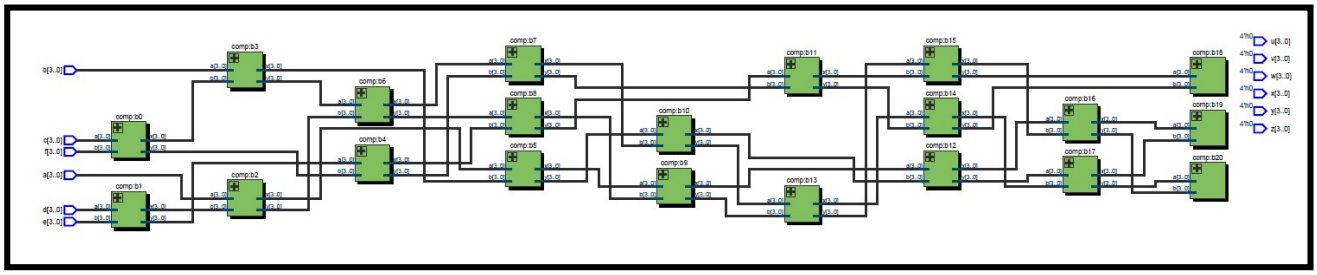
Sieć sortującą algorytmem *Balanced* zrealizowaliśmy, tworząc interfejs modułu (wejścia i wyjścia), a także połączenia logiczne bloków komparatora. Moduł ten przedstawiliśmy na **Wydruku 2.** poniżej.

Wydruk 2: Moduł `sn` w Verilog, plik `.v`

```
module sn
(
        input [3:0] a,b,c,d,e,f,
        output [3:0] u,v,w,x,y,z
);

wire [3:0] t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,t13,
           t14,t15,t16,t17,t18,t19,t20,t21,t22,t23,t24,
           t25,t26,t27,t28,t29,t30,t31,t32,t33,t34,t35;

comp b0 (.a(c), .b(f), .x(t0), .y(t1));
comp b1 (.a(d), .b(e), .x(t2), .y(t3));
comp b2 (.a(a), .b(t2), .x(t4), .y(t5));
comp b3 (.a(b), .b(t0), .x(t6), .y(t7));
comp b4 (.a(t3), .b(t1), .x(t8), .y(t9));
comp b5 (.a(t4), .b(t6), .x(t10), .y(t11));
comp b6 (.a(t7), .b(t5), .x(t12), .y(t13));
comp b7 (.a(t12), .b(t9), .x(t14), .y(t15));
comp b8 (.a(t13), .b(t8), .x(t16), .y(t17));
comp b9 (.a(t10), .b(t16), .x(t18), .y(t19));
comp b10 (.a(t11), .b(t14), .x(t20), .y(t21));
comp b11 (.a(t17), .b(t15), .x(t22), .y(t23));
comp b12 (.a(t18), .b(t20), .x(t24), .y(t25));
comp b13 (.a(t21), .b(t19), .x(t26), .y(t27));
comp b14 (.a(t26), .b(t23), .x(t28), .y(t29));
comp b15 (.a(t27), .b(t22), .x(t30), .y(t31));
comp b16 (.a(t24), .b(t30), .x(t32), .y(t33));
comp b17 (.a(t25), .b(t28), .x(t34), .y(t35));
comp b18 (.a(t31), .b(t29), .x(y), .y(z));
comp b19 (.a(t32), .b(t34), .x(u), .y(v));
comp b20 (.a(t35), .b(t33), .x(w), .y(x));


endmodule
```

6.3. Schemat sieci sortującej wygenerowany automatycznie

W celu obejrzenia schematu sieci sortującej automatycznie wygenerowanej przez program Quartus, należy wybrać z listy na górze: *Tools/Netlist Viewers/RTL Viewer*. Układ naszej sieci sortującej przedstawiliśmy na **Rys. 4**.



Rys. 4: Schemat sieci sortującej wygenerowany przez program Intel Quartus Prime.

6.4. Testowanie układu

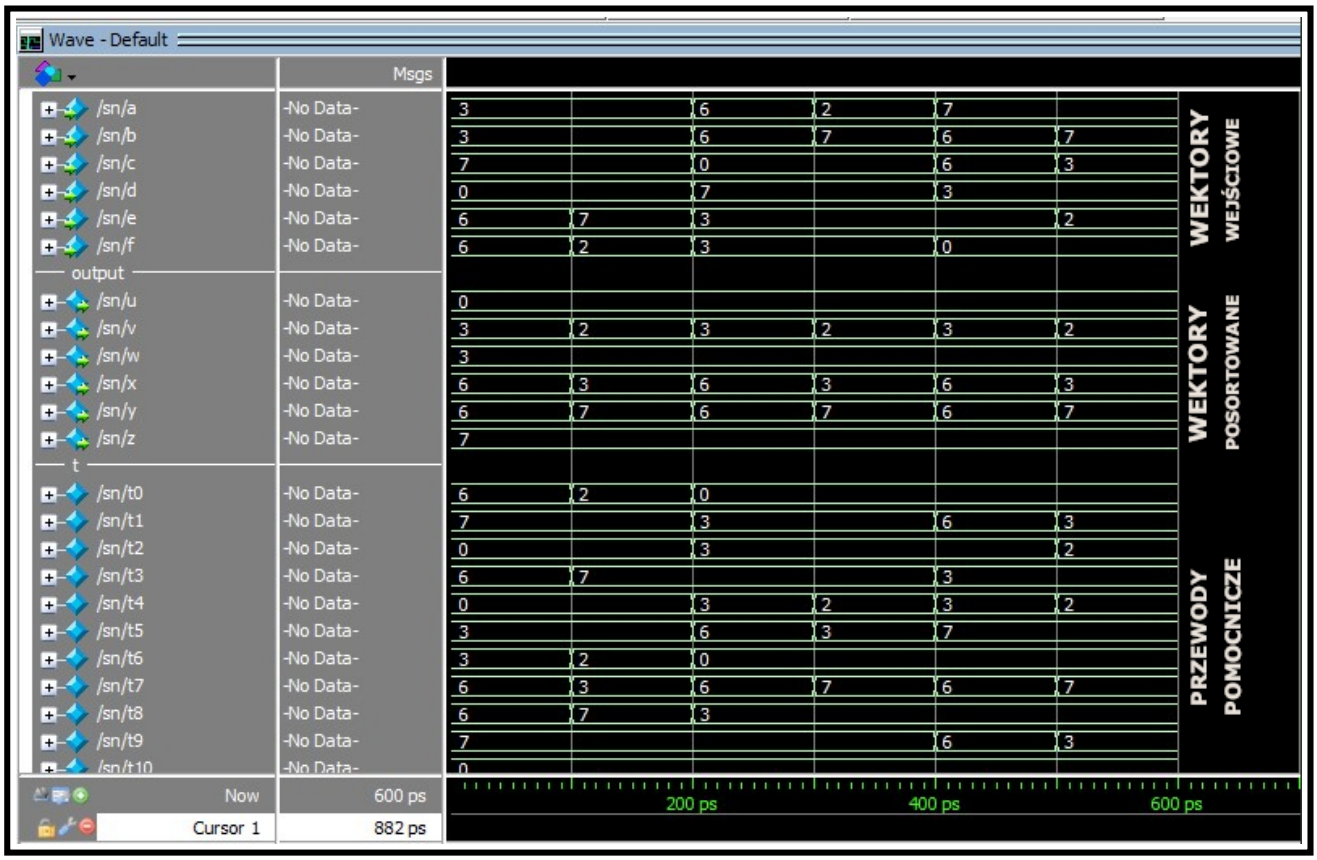
Do testowania układu posłużyliśmy się plikiem `test.do` przedstawionym na **Wydruku 3.**, w którym znajduje się 6 wektorów:

Wydruk 3: Plik `test.do` służący do weryfikacji poprawności modułów `.v`.

```
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#6 0
force f 10#6 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#7 0
force f 10#2 0
run

force a 10#6 0
force b 10#6 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#2 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#7 0
force b 10#6 0
force c 10#6 0
force d 10#3 0
force e 10#3 0
force f 10#0 0
run

force a 10#7 0
force b 10#7 0
force c 10#3 0
force d 10#3 0
force e 10#2 0
force f 10#0 0
run
```

Wyniki testów przeprowadzonych w programie ModelSim zostały przedstawiony na **Rys. 5**.



Rys. 5: Wynik testów przeprowadzonych dla wektorów w programie ModelSim.

Widzimy na przebiegach czasowych jak zamieniane są wektory wejściowe na wektory posortowane:

Wektor 1: $(3,\ 3,\ 7,\ 0,\ 6,\ 6) \rightarrow (0,\ 3,\ 3,\ 6,\ 6,\ 7)$
Wektor 2: $(3,\ 3,\ 7,\ 0,\ 7,\ 2) \rightarrow (0,\ 2,\ 3,\ 3,\ 7,\ 7)$
Wektor 3: $(6,\ 6,\ 0,\ 7,\ 3,\ 3) \rightarrow (0,\ 3,\ 3,\ 6,\ 6,\ 7)$
Wektor 4: $(2,\ 7,\ 0,\ 7,\ 3,\ 3) \rightarrow (0,\ 2,\ 3,\ 3,\ 7,\ 7)$
Wektor 5: $(7,\ 6,\ 6,\ 3,\ 3,\ 0) \rightarrow (0,\ 3,\ 3,\ 6,\ 6,\ 7)$
Wektor 6: $(7,\ 7,\ 3,\ 3,\ 2,\ 0) \rightarrow (0,\ 2,\ 3,\ 3,\ 7,\ 7)$

6.5. Symulacja układu

Link do filmiku z nagranymi testami ze wskazanymi punktami charakterystycznymi, skąd widać, że uzyskano poprawny wynik: `https://youtu.be/6Qcq6xIbeVM`.

7. Podsumowanie i wnioski

Zaprojektowany układ został poprawnie zaimplementowany zarówno w środowisku Logisim-evolution, jak i Intel Quartus Prime oraz ModelSim, spełniając wszystkie założenia i cele ćwiczenia laboratoryjnego. W trakcie testów nie stwierdzono błędów w działaniu układu – wynik działania był zgodny z oczekiwanym w obu przypadkach. Liczby zostały posortowane rosnąco w sposób prawidłowy. Implementacja sieci sortującej oraz testowanie ich na zadanych wektorach w obu środowiskach umożliwiła pełną weryfikację poprawności działania układu. Wszystkie operacje wykonywane były w założonych przedziałach czasowych, bez zauważalnych opóźnień ani żadnych błędów logicznych. Zastosowanie algorytmu *Balanced* w sieci sortującej zapewniło optymalną równowagę pomiędzy czasem wykonania operacji a zużyciem zasobów, co przełożyło się na jego wysoką wydajność w testowanych środowiskach.

Spis tabel

Spis rysunków

Spis wydruków