Politechnika
Wrocławska

Wydział Elektroniki

Organizacja i Architektura Komputerów

Implementacja fizyczna układów cyfrowych

Autorzy:
Maja Bojarska
Paweł Sajewicz

Prowadzący projekt:
Dr inż. Piotr Patronik

Wrocław 11 czerwca 2020

Spis treści

1 Wstęp

1.1 Cele projektu

Cel ogólny: Synteza logiczna i fizyczna sumatora prefiksowego z wykorzystaniem narzędzi Yosys/Qflow.

Cele szczegółowe:

- Analiza literatury w zakresie narzędzia Yosys/Qflow.
- Wybór architektury 6-bitowego sumatora prefiksowego i jej zapis w strukturalnym języku Verilog.
- Implementacja testu wyczerpującego.
- Synteza logiczna układu z wykorzystaniem Yosys.
- Synteza fizyczna układu z wykorzystaniem Qflow.

1.2 Użyte technologie

- Ubuntu 20.04 [1] - system operacyjny GNU/Linux.
- Verilog 2005 [2] - język opisu sprzętu.
- Icarus Verilog 10.3 [3] - symulator języka Verilog.
- Pytest [4] - framework do tworzenia testów.
- Yosys 0.9 [5] - narzędzie do syntezy Verilog.
- Qflow 1.3 [6] - narzędzie do syntezy układów cyfrowych.

2 Podstawy

2.1 Sumatory prefiksowe

W celu skonstruowania sumatora prefiksowego najpierw musimy poznać zależności wytwarzania przez nie sum oraz propagacji i generacji przeniesień. Prezentują je wzory [7]:

$s_i = h_i \oplus c_i = h_i \oplus (G_{i-1:0} + P_{i-1:0}c_0)$,

$c_i = G_{i-1:0} + P_{i-1:0}c_0$,

$h_i = x_i \oplus y_i$,

gdzie s_i to bit sumy na i-tej pozycji, c_i - przeniesienia, h_i - pół sumy, $G_{i-1:0}$ - generacji przeniesienia, $P_{i-1:0}$ - propagacji przeniesienia, x_i - pierwszego składnika dodawania, y_i drugiego składnika, a c_0 to przeniesienie z poprzedniej pozycji.

Węzeł sieci GP realizuje funkcję

$$(G_{HL}, P_{HL}) = (G_H, P_H) \bullet (G_L, P_L) = (G_H + P_H G_L, _H P_L). \tag{1}$$

Zasada konstrukcji struktury GP:

integracja funkcji G_H i P_H oraz G_L i P_L obejmujących sąsiadujące bloki H i L:

- bloki H i L powinny być styczne,
- bloki H i L nie mogą być rozdzielone,
- bloki H i L mogą mieć część wspólną – funkcje G_{HL} i P_{HL} są nadmiarowe,
- regularne struktury dla $n = 2^k$ wejść (pozycji),
- w innych przypadkach przyjąć $k = int(1 + log_2 n)$ i usunąć zbędne gałęzie (sieć integrującą 2^{k-1} pozycji połączyć siecią integrującą pozostałe wejścia).

Przekształcenie prefiksowe Ladnera-Fischera (Sklansky'ego) [8]:

$P_{i:i} = x_i \oplus y_i, G_{i:i} = x_i y_i (i = 0, 1, \ldots, n-1)$ $G_{0:0}$

Poziom 1 $(i = 0, 1, \ldots, 2^{-1}n - 1)$ $G_{1:0}$
$(G_{2i+1:2i}, P_{2i+1:2i}) = (G_{2i+1:2i+1} + P_{2i+1:2i+1} G_{2i:2i}, P_{2i+1:2i+1} P_{2i:2i})$

Poziom 2 $(i = 0, 1, \ldots, 2^{-2}n - 1; s = 2, 3)$ $G_{3:0}, G_{2:0}$
$(G_{4i+s:4i}, P_{4i+s:4i}) = (G_{4i+s:4i+2} + P_{4i+s:4i+2} G_{4i+1:4i}, P_{4i+s:4i+2} P_{4i+1:4i})$

Poziom 3 $(i = 0, 1, \ldots, 2^{-3}n - 1; s = 4, 5, 6, 7)$ $G_{7:0}, \ldots, G_{4:0}$

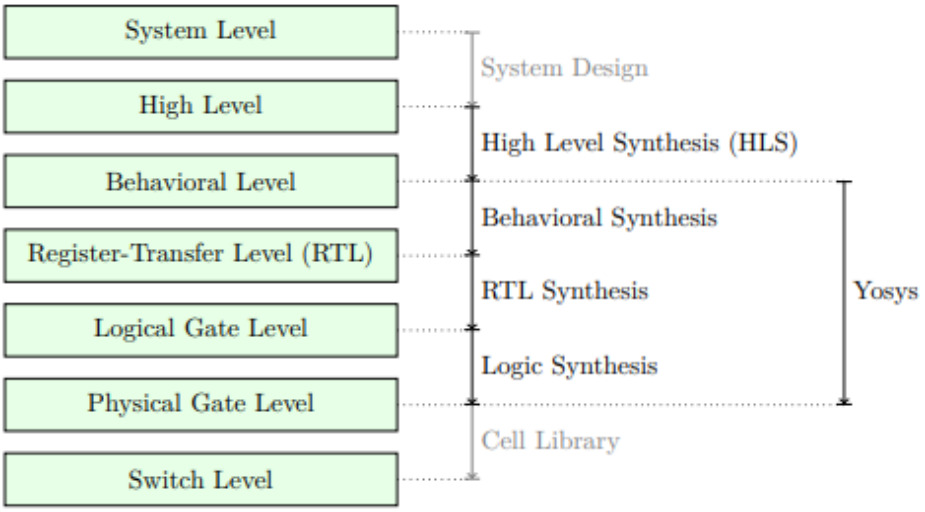
$$(G_{8i+s,8i}, P_{8i+s,8i}) = (G_{8i+s,8i+4} + P_{8i+s,8i+4}G_{8i+3,8i}, P_{8i+s,8i+4}P_{8i+3,8i})$$

Poziom 4 $(i = 0, 1, \ldots, 2^{-4}n - 1; s = 8, 9, \ldots, 15)$ $G_{15:0}, \ldots, G_{8:0}$
$(G_{16i+s,16i}, P_{16i+s,16i}) = (G_{16i+s,16i+8} + P_{16i+s,16i+8}G_{16i+7,16i}, P_{16i+s,16i+8}P_{16i+7,16i})$
...

2.2 Yosys

Yosys [5] to oprogramowanie do syntezy Verilog HDL, które obsługuje ogromną większość możliwych do syntezy funkcji Verilog. Synteza HDL służy do tłumaczenia kodu języka opisu sprzętu Verilog HDL na układy cyfrowe. Yosys został zbudowany jako rozszerzalna platforma, dzięki czemu można go łatwo używać jako podstawy dla niestandardowych przepływów syntezy oraz jako środowisko do wdrażania i badań nad nowymi algorytmami syntezy. Yosys posiada szerokie wsparcie dla Verilog HDL i jest w stanie syntezować złożone projekty Verilog. [9]

Synteza jest automatyczną konwersją reprezentacji wysokiego poziomu obwodu na funkcjonalnie równoważną reprezentację niskiego poziomu obwodu. Rysunek 1 przedstawia różne poziomy abstrakcji i ich związek z różnymi rodzajami syntezy. Jak widzimy oprogramowanie Yosys odpowiada za syntezę behawioralną, RTL i logiczną.



Rysunek 1: Różne poziomy abstrakcji i syntezy. [9]

2.3 Qflow

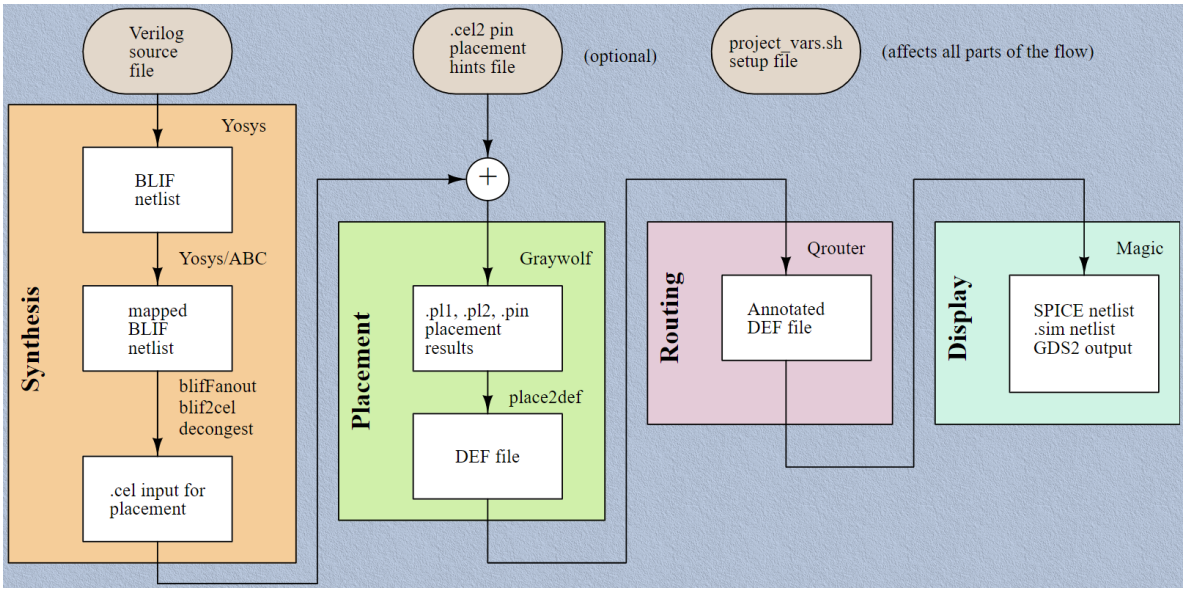
Qflow to kompletny łańcuch narzędzi do syntezy układów cyfrowych, czyli zestaw narzędzi i metod używanych do przekształcenia projektu układu napisanego w wysoko poziomowym języku behawioralnym, takim jak verilog lub VHDL, w układ fizyczny.

Taki obwód może być kodem konfiguracji dla układu FPGA, takiego jak układ Xilinx lub Altera, który stałby się częścią ukształtowanego układu scalonego. [6]

Qflow korzysta z szeregu narzędzi, każde jest odpowiedzialne za inną funkcję:

- Yosys - parsowanie, synteza, optymalizacja i weryfikacja Verilog.
- Graywolf - rozmieszczenie komórek i pinów.
- Grouter - szczegółowy routing.
- Magic - przeglądanie, ekstrakcja, sprawdzanie DRC, generowanie GDS.
- Netgen - LVS (Layout vs. Schematic), weryfikuje czy otrzymany układ scalony odpowiada oryginalnemu schematowi obwodu w projekcie.

Proces syntezy obrazuje rysunek 2.



Rysunek 2: Proces syntezy z użyciem Qflow. [10]

3 Rozwiązanie

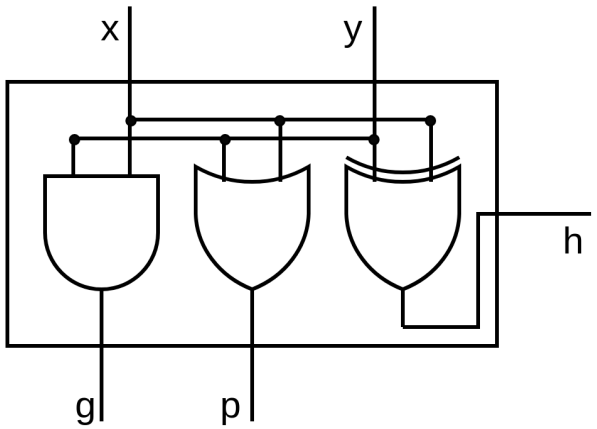
3.1 Architektura układu

Przed rozpoczęciem opisu układu językiem Verilog, każdy z modułów został zaprojektowany w postaci układu, składającego się z bramek logicznych i/lub lub modułów.

3.1.1 Blok PG

Blok PG realizuje funkcje logiczne generacji oraz propagacji, opisane funkcjami ze wzoru nr 2.

$$\begin{cases} g = x * y \\ p = x + y \\ h = x \oplus y \end{cases} \tag{2}$$

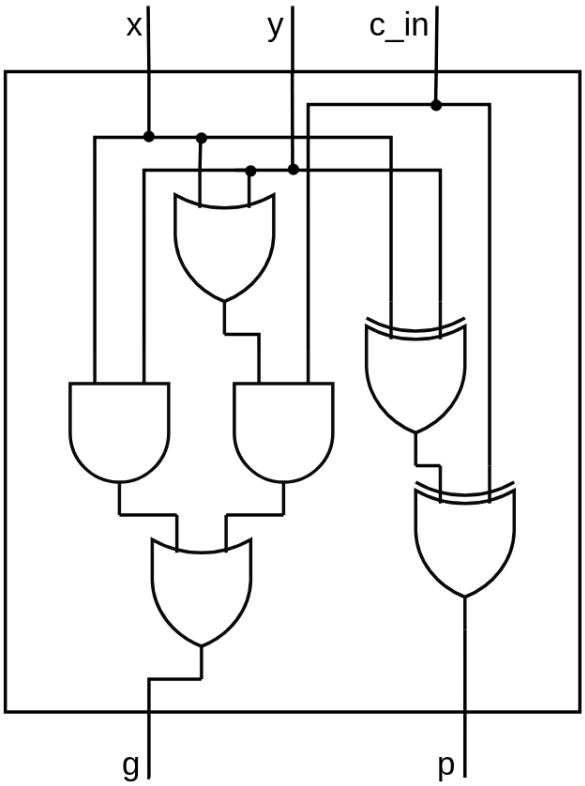


Rysunek 3: Schemat bloku PG

3.1.2 Blok PG_IN

Blok PG_IN jest szczególnym przypadkiem bloku PG. Pozwala on na wczesne włączenie korekty wejściowej (c_in) do sumatora. Funkcje logiczne opisujące wyjścia tego układu zostały opisane we wzorze nr 3.

$$\begin{cases} g = (x * y) + (c_{in} * (x + y)) \\ p = x \oplus y \oplus c_{in} \end{cases} \tag{3}$$

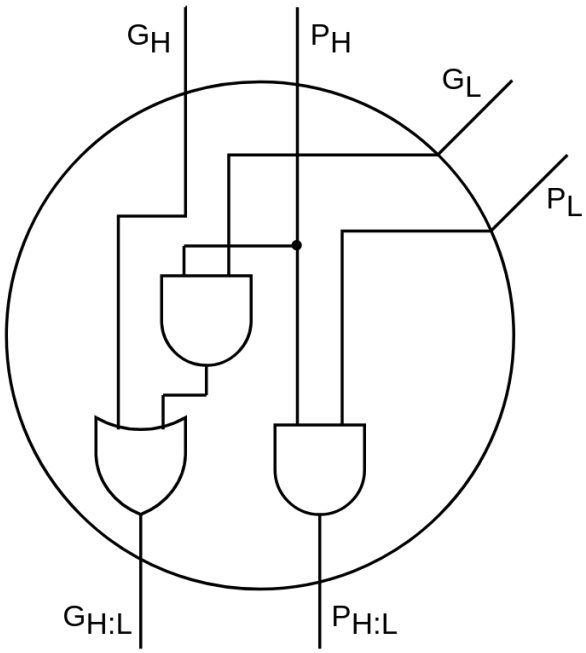


Rysunek 4: Schemat bloku PG_IN

3.1.3 Węzeł sieci grafu prefiksowego

Układ ten realizuje funkcję wektorową, opisaną wzorem nr 1. Funkcje opisujące poszczególne wyjścia, zostały opisane na wzorze nr 4.

$$\begin{cases} G_{H:L} = G_H + (P_H * G_L) \\ P_{H:L} = P_H * P_L \end{cases} \tag{4}$$

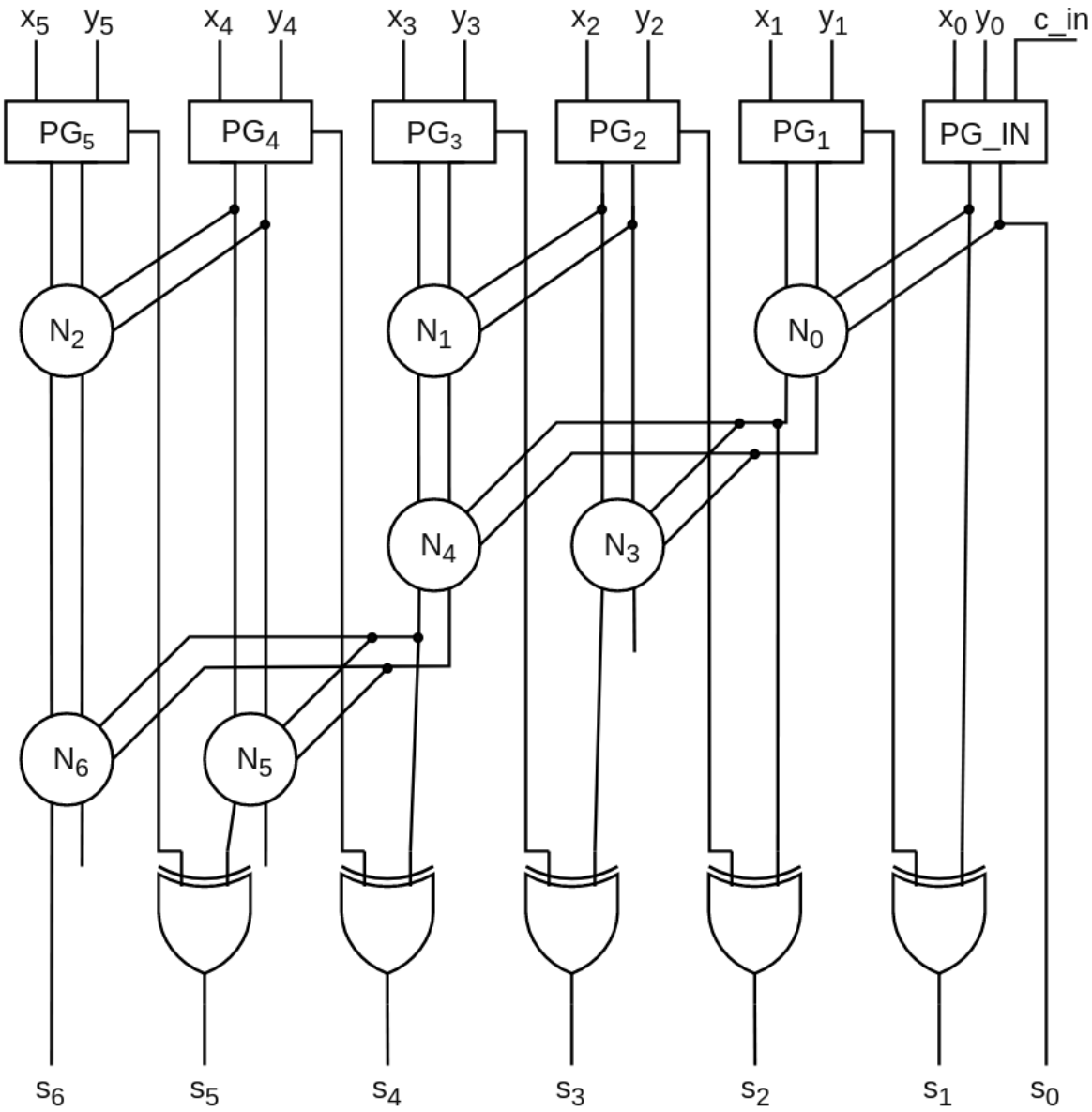


Rysunek 5: Schemat węzła sieci grafu prefiksowego.

3.1.4 Sumator prefiksowy

Do zaprojektowania układu została wykorzystana architektura Ladnera-Fischera (Sklansky'ego) [8]. Sygnały wejściowe oraz wyjściowe, zostały opisane wzorem nr 5.

$$\begin{cases} X = \{x_5, x_4, ..., x_0\} \\ Y = \{y_5, y_4, ..., y_0\} \\ S = \{s_6, s_5, ..., s_0\} = X + Y + c_{in} \end{cases} \tag{5}$$



Rysunek 6: Schemat układu 6-bitowego sumatora prefiksowego.

3.2 Implementacja układu za pomocą języka Verilog

Układy z sekcji 3.1.1, 3.1.2, 3.1.3, zostały zrealizowane jako niezależne moduły Verilog. Dopiero moduł *prefix_ adder* łączy je w 6-bitowy układ sumatora prefiksowego.

Kod bloków PG i węzłów, przedstawiony na listingach 1, 2, 3, całkowicie pokrywa się z zaprezentowanymi wcześniej schematami. Moduły zawierają identyczne porty co na rysunkach 3, 4, 5, odpowiednio zdefiniowane w kodzie jako wejście lub wyjście (*input*, *output*). Odpowiednie operatory określają bramki logiczne, np. "&" dla bramki logicznej AND.

```
module pg (
    input x, y,
    output gen, prop, half_sum
);

assign gen = x & y;
assign prop = x | y;
assign half_sum = x ^ y;

endmodule
```

Listing 1: Moduł PG

```
module pg_in (
    input x, y, c_in,
    output gen, prop
);

assign gen = (x & y) | (c_in & (x | y));
assign prop = x ^ y ^ c_in;

endmodule
```

Listing 2: Moduł PG_IN

```
module prefix_node (
    input gen_high, prop_high, gen_low, prop_low,
    output gen_out, prop_out
);

assign gen_out = gen_high | (prop_high & gen_low);
assign prop_out = prop_high & prop_low;

endmodule
```

Listing 3: Moduł węzła

Kod sumatora jest bardziej złożony. Został przedstawiony na listingu 4. Moduł zawiera dwa wektory sygnałów wejściowych, X i Y dla dodawanych liczb, sygnał przeniesienia z niższej pozycji *c_in* oraz wektor sygnałów wyjściowych dla otrzymanej sumy S.

W liniach 17-23 zostały zadeklarowane wektory typu *wire*, pełniące funkcje połączeń przewodzących sygnały pomiędzy elementami aktywnymi (bramkami).

Używanie zaimportowanych modułów przypomina korzystanie z funkcji w językach programowania. Składnia wygląda następująco:

$$module\ name(.port(sygnal), ...),$$

gdzie *module* to nazwa zaimportowanego modułu, *name* to unikalna nazwa instancji, *port* to nazwa portu w zaimportowanym module, a *sygnal* to sygnał z obecnego modułu (tak jak naszym obecnym modułem jest sumator). Przykład użycia zewnętrznego modułu widać na przykład w linijce 25 listingu 4. Moduły użyte w liniach 25-39 odpowiadają za sygnały wchodzące i wychodzące do bloków PG. widać to szczególnie dobrze, gdy porównuje się kod bezpośrednio ze schematem sumatora na rysunku 6.

Moduły zaczynające się w lini 42 to węzły grafu prefiksowego. Moduł *prefix_node_x* odpowiada węzłowi N_x z rysunku 6, gdzie $x \in [0, 6] \ \wedge \ x \in \mathbb{N}$.

Za przykład niech posłuży *prefix_node_3*. Węzeł ten przyjmuje wyjścia z bloku PG na pozycji 2 oraz wyjścia z węzła 0, a następnie wyprowadza swoje wyjścia na *node_out_g[3]* i *node_out_p[3]*.

Na koniec wyznaczane są bity wektora wyjściowego S. Są to proste operacje XOR z bitów pół-sumy i sygnałów generacji przeniesienia z niższej pozycji (może to być sygnał wyjściowy z konkretnego węzła lub bloku). Wyjątkiem jest najmłodszy bit sumy, będący propagacją z bloku PG_IN.

```
1  `include "pg.v"
2  `include "pg_in.v"
3  `include "prefix_node.v"
4
5  module prefix_adder (
6      X, Y, c_in, S
7  );
8
9  input [5:0] X;
10 input [5:0] Y;
11 input c_in;
12
13 // Sum
14 output [6:0] S;
15
16 // PG block outputs, p - propagation, g - generation, h - half sum.
17 wire [5:0] pg_out_p;
18 wire [5:0] pg_out_g;
19 wire [5:0] pg_out_h;
```

```
20
21 // Prefix graph nodes outputs, p - propagation, g - generation.
22 wire [6:0] node_out_p;
23 wire [6:0] node_out_g;
24
25 pg_in pg_in_block(
26     .x(X[0]),
27     .y(Y[0]),
28     .c_in(c_in),
29     .gen(pg_out_g[0]),
30     .prop(pg_out_p[0])
31 );
32
33 pg pg_blocks[5:1] (
34     .x(X[5:1]),
35     .y(Y[5:1]),
36     .gen(pg_out_g[5:1]),
37     .prop(pg_out_p[5:1]),
38     .half_sum(pg_out_h[5:1])
39 );
40
41 // Instantiate prefix graph nodes.
42 prefix_node prefix_node_0(
43     .gen_high(pg_out_g[1]),
44     .prop_high(pg_out_p[1]),
45     .gen_low(pg_out_g[0]),
46     .prop_low(pg_out_p[0]),
47     .gen_out(node_out_g[0]),
48     .prop_out(node_out_p[0])
49 );
50 prefix_node prefix_node_1(
51     .gen_high(pg_out_g[3]),
52     .prop_high(pg_out_p[3]),
53     .gen_low(pg_out_g[2]),
54     .prop_low(pg_out_p[2]),
55     .gen_out(node_out_g[1]),
56     .prop_out(node_out_p[1])
57 );
58 prefix_node prefix_node_2(
59     .gen_high(pg_out_g[5]),
60     .prop_high(pg_out_p[5]),
61     .gen_low(pg_out_g[4]),
62     .prop_low(pg_out_p[4]),
63     .gen_out(node_out_g[2]),
64     .prop_out(node_out_p[2])
65 );
66 prefix_node prefix_node_3(
67     .gen_high(pg_out_g[2]),
68     .prop_high(pg_out_p[2]),
69     .gen_low(node_out_g[0]),
70     .prop_low(node_out_p[0]),
71     .gen_out(node_out_g[3]),
```

```
72      .prop_out(node_out_p[3])
73  );
74  prefix_node prefix_node_4(
75      .gen_high(node_out_g[1]),
76      .prop_high(node_out_p[1]),
77      .gen_low(node_out_g[0]),
78      .prop_low(node_out_p[0]),
79      .gen_out(node_out_g[4]),
80      .prop_out(node_out_p[4])
81  );
82  prefix_node prefix_node_5(
83      .gen_high(pg_out_g[4]),
84      .prop_high(pg_out_p[4]),
85      .gen_low(node_out_g[4]),
86      .prop_low(node_out_p[4]),
87      .gen_out(node_out_g[5]),
88      .prop_out(node_out_p[5])
89  );
90  prefix_node prefix_node_6(
91      .gen_high(node_out_g[2]),
92      .prop_high(node_out_p[2]),
93      .gen_low(node_out_g[4]),
94      .prop_low(node_out_p[4]),
95      .gen_out(node_out_g[6]),
96      .prop_out(node_out_p[6])
97  );
98
99  // Assign S output values.
100 assign S[0] = pg_out_p[0];
101 assign S[1] = pg_out_h[1] ^ pg_out_g[0];
102 assign S[2] = pg_out_h[2] ^ node_out_g[0];
103 assign S[3] = pg_out_h[3] ^ node_out_g[3];
104 assign S[4] = pg_out_h[4] ^ node_out_g[4];
105 assign S[5] = pg_out_h[5] ^ node_out_g[5];
106 assign S[6] = node_out_g[6];
107
108 endmodule
```

Listing 4: Moduł sumatora prefiksowego

3.3 Testy jednostkowe

Testy jednostkowe modułów Verilog zostały zrealizowane za pomocą symulatora Icarus Verilog [3] oraz frameworku pytest [4]. Moduły zostały przetestowane dla wszystkich możliwych wektorów sygnałów wejściowych.

W celu zachowania przejrzystości dokumentacji, został omówiony jedynie proces testowania modułu sumatora prefiksowego. Testowanie pozostałych modułów odbywa się analogicznie.

Tabela 1: Liczba przypadków testowych przypadających na moduł Verilog.

Moduł Verilog	**Liczba przypadków testowych**
pg	$2^2 = 4$
pg_in	$2^3 = 8$
prefix_node	$2^4 = 16$
prefix_adder	$2^{13} = 8192$

3.3.1 Testbenche Verilog

Symulacja działania modułów odbyła się poprzez utworzenie instacji testowanej jednostki z poziomu przynależącego do niej testbencha, a następnie sekwencyjne podanie wszystkich mozliwych wektorów sygnałów wejściowych. Przy każdym przebiegu pętli zadającej sygnały wejściowe, stan wejść oraz wynikających z nich wyjść, był zapisywany jako ciąg znaków, zgodny ze składnią formatu JSON.

```
1  `include "prefix_adder.v"
2
3  module prefix_adder_tb;
4
5  reg [5:0] X;
6  reg [5:0] Y;
7  reg c_in;
8
9  wire [6:0] S;
10
11 parameter TEST_SIGNAL_COUNT = 2 ** (13);
12
13 initial begin
14     $monitor (
15         "{\"time\": \"%t\", \"X\": \"%6b\", \"Y\": \"%6b\", \"c_in\":\"%b
       \", \"S\": \"%6b\"}",
16         $time, X, Y, c_in, S
17     );
18
19     for(int i=0; i<TEST_SIGNAL_COUNT; i=i+1) begin
20         {X, Y, c_in} = i;
```

```
        #1;
    end

    #1 $finish;
end

// UUT
prefix_adder U0 (
    .X(X),
    .Y(Y),
    .c_in(c_in),
    .S(S)
);

endmodule
```

Listing 5: Testbench dla modułu prefix_adder

Testbenche były syntezowane i symulowane zgodnie ze standardem SystemVerilog 2012. Wybór tej wersji wynika z dodania wsparcia dla pętli for, która umożliwiła uproszczenie kodu i ograniczyła możliwości popełnienia błędów w jego pisaniu.

```
iverilog -Wall -g2012 -o ../test/pg.out tb_prefix_adder.v
```

Listing 6: Przykładowe wywołanie symulacji testbencha za pomocą programu Icarus Verilog.

```
{"time": "0", "X": "000000", "Y": "000000", "c_in":"0", "S": "0000000"}
{"time": "1", "X": "000000", "Y": "000000", "c_in":"1", "S": "0000001"}
{"time": "2", "X": "000000", "Y": "000001", "c_in":"0", "S": "0000001"}
{"time": "3", "X": "000000", "Y": "000001", "c_in":"1", "S": "0000010"}
{"time": "4", "X": "000000", "Y": "000010", "c_in":"0", "S": "0000010"}
{"time": "5", "X": "000000", "Y": "000010", "c_in":"1", "S": "0000011"}
{"time": "6", "X": "000000", "Y": "000011", "c_in":"0", "S": "0000011"}
{"time": "7", "X": "000000", "Y": "000011", "c_in":"1", "S": "0000100"}
{"time": "8", "X": "000000", "Y": "000100", "c_in":"0", "S": "0000100"}
```

Listing 7: Początkowy fragment treści wynikowej wywołania testbencha dla modułu prefix_adder

3.3.2 Testy jednostkowe pytest

Wynik wywołania testbencha (listing 7) był parsowany, a następnie były określane wartości oczekiwane, dla dostarczonych wartości wejściowych.

```
@pytest.mark.parametrize("entry", data_prefix_adder)
def test_prefix_adder(entry):
    out_sum = entry["S"]
    arg_x = entry["X"]
    c_in = entry["c_in"]
    arg_y = entry["Y"]

    assert out_sum == arg_x + arg_y + c_in, "Invalid sum: {}".format(
    entry)
```

Listing 8: Test jednostkowy pytest, dla modułu prefix_adder

Test z listingu 8, dla poprawnie działającego modułu PG, daje wynik przedstawiony na listingu 9.

```
collected 8220 items / 28 deselected / 8192 selected

test/test_tb_outputs.py::test_prefix_adder[entry0] PASSED          [  0%]
test/test_tb_outputs.py::test_prefix_adder[entry1] PASSED          [  0%]
test/test_tb_outputs.py::test_prefix_adder[entry2] PASSED          [  0%]
test/test_tb_outputs.py::test_prefix_adder[entry3] PASSED          [  0%]
(8186 pominietych linii)
test/test_tb_outputs.py::test_prefix_adder[entry8190] PASSED       [ 99%]
test/test_tb_outputs.py::test_prefix_adder[entry8191] PASSED       [100%]

================== 8192 passed, 28 deselected in 45.25s ==================
```

Listing 9: Wynik wywołania testów pytest dla modułu prefix_adder.

3.4 Synteza za pomocą narzędzi Yosys i Qflow

Do syntezy układu zostały wykorzystane programy wymienione w tabeli 2.

Tabela 2: Wersje programów użytych do syntezy układu.

Narzędzie	**Wersja**
Qflow	1.3r17
Yosys	0.9-1build2
Graywolf	0.1.6-3
Qrouter	1.4.71.T
Magic	8.2.157
NetGen	1.5.133
Icarus Verilog	10.3

Wybrana technologia wykonania układu to "osu018". Jej wybór wynika z łatwej dostępności wymaganych bibliotek (są dostarczone wraz z narzędziem qflow). Określa ona właściwości fizyczne układu wynikowego, takie jak szybkość, wymagana powierzchnia, zużycie energii.

W procesie syntezy zostały wykonane sekwencyjnie następujące kroki:

1. synthesize - synteza modułu Verilog za pomocą programu yosys,
2. place - rozmieszczenie komórek i pinów,
3. buffer - zmniejszenie obciążalności wyjść układu,
4. route - trasowanie ścieżek,
5. sta - statyczna analiza czasowa,
6. migrate - rozplanowanie przestrzenne elementów, określenie list połączeń,
7. drc - sprawdzenie zgodności układu z regułami projektowymi,
8. lvs - sprawdzenie zgodności układu ze schematem,
9. gdsii - utworzenie masek w formacie GDSII, opisujących powierzchnię warstw układu [12],
10. clean - usunięcie plików tymczasowych w katalogach wyjściowych.

Zadanie syntezy zostało zautomatyzowane za pomocą narzędzia GNU Make [11]. Dzięki temu, sekwencję kroków syntezy można wywołać poleceniem "make qflow_custom" z powłoki systemowej.

```
QFLOW_TECH = 'osu018'
QFLOW_TARGET_MODULE = 'prefix_adder'

qflow_custom: qflow_dirs
  qflow synthesize --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow place --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow buffer --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow route --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow sta --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow migrate --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow drc --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow lvs --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow gdsii --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)
  qflow clean --tech $(QFLOW_TECH) $(QFLOW_TARGET_MODULE)

qflow_dirs:
  mkdir -p synthesis layout log reports
```

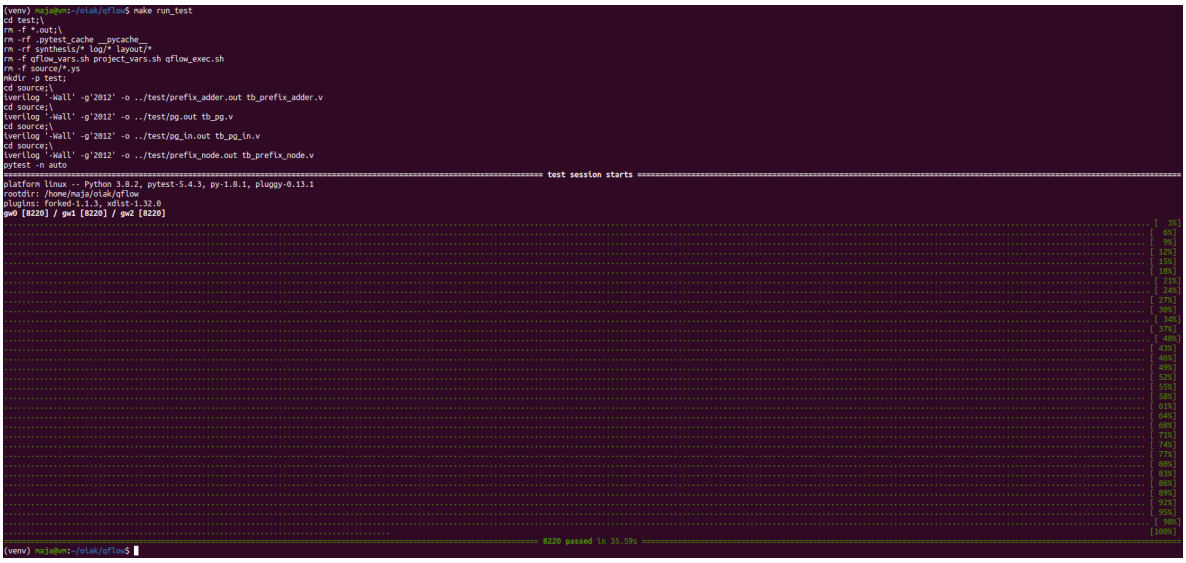
Listing 10: Fragment pliku Makefile, realizujący syntezę modułu prefix_adder.

4 Wyniki i dyskusja

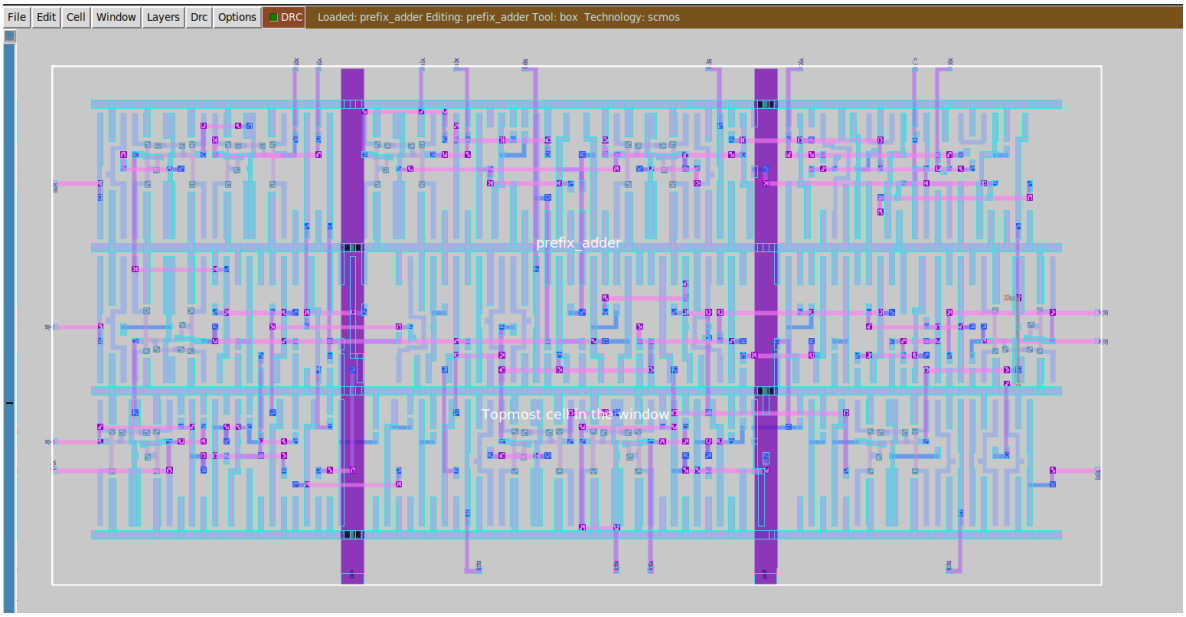
4.1 Testy i synteza

Zaprojektowany sumator działa poprawnie. Moduły napisane w języku Verilog pomyślnie przeszły kompilację oraz zwracały właściwe wyniki podczas testowania w symulatorze. Przeprowadzony później test wyczerpujący, również zakończył się całkowitym powodzeniem. Jego wynik można zobaczyć na rysunku 7.

Udało się również przeprowadzić pełną syntezę, zarówno logiczną jak i fizyczną. Powstały w ten sposób układ prezentuje rysunek 8.



Rysunek 7: Wynik testu dla sumatora



Rysunek 8: Topografia układu implementującego moduł prefix_adder.

4.2 Statyczna analiza czasowa

Statyczna analiza czasowa została przeprowadzona za pomocą narzędzia qflow. Wywołanie odpowiedniego polecenia oraz fragment wyniku został umieszczony na listingu 11.

```
$ qflow sta prefix_adder
(czesc tresci pominieta dla zachowania czytelnosci)
Top 0 maximum delay paths:
Computed maximum clock frequency (zero margin) = inf MHz
------------------------------------------------

Number of paths analyzed:  0

Top 0 minimum delay paths:
Design meets minimum hold timing.
------------------------------------------------

Number of paths analyzed:  7

Top 7 maximum delay paths:
Path input pin Y[0] to output pin S[5] delay 1041.84 ps
      0.0 ps                      Y[0]:                 ->  AOI21X1_1/A
     93.5 ps                     _14_:  AOI21X1_1/Y ->   NOR2X1_7/B
    212.2 ps          pg_in_block_gen:   NOR2X1_7/Y ->  NAND2X1_7/A
    301.7 ps                     _17_:  NAND2X1_7/Y ->  NAND2X1_8/B
    438.0 ps  prefix_node_0_gen_out:  NAND2X1_8/Y -> NAND2X1_15/A
    534.6 ps                     _25_: NAND2X1_15/Y -> NAND2X1_16/B
    672.6 ps  prefix_node_4_gen_out: NAND2X1_16/Y -> NAND2X1_17/A
    769.3 ps                     _27_: NAND2X1_17/Y -> NAND2X1_18/B
    864.7 ps  prefix_node_5_gen_out: NAND2X1_18/Y ->   XOR2X1_5/A
    961.0 ps                   _0__5_:   XOR2X1_5/Y ->    BUFX2_6/A
   1041.8 ps                     S[5]:    BUFX2_6/Y -> S[5]
```

Listing 11: Wywołanie oraz fragment wyniku statycznej analizy czasowej.

Na podstawie listingu 11, można określić minimalny czas wymagany do uzyskania poprawnego wyniku na wyjściach układu, od momentu zadania sygnałów wejściowych. Ścieżką układu wymagającą najwięcej czasu jest ścieżka $Y[0] \rightarrow S[5]$. Czas propagacji sygnału przez całą jej długość, wynosi 1041.84ps, zatem takie jest opóźnienie sygnału dla całego układu sumatora.

5 Wnioski

Udało się zrealizować wszystkie cele projektu. Wybrano architekturę sumatora prefiksowego i zapisano ją w języku Verilog. Zaimplementowano test wyczerpujący, z wykorzystaniem frameworku pytest, który zakończył się powodzeniem. Przeprowadzono

również syntezę logiczną i fizyczną z pomocą narzędzi Yosys i Qflow. Cały proces został zautomatyzowany za pomocą narzędzia GNU Make, co znacząco ułatwia jego powtórne przeprowadzenie.

Literatura

[1] System operacyjny Ubuntu
https://ubuntu.com/

[2] IEEE Standard for Verilog Hardware Description Language
http://staff.ustc.edu.cn/~songch/download/IEEE.1364-2005.pdf

[3] Icarus Verilog
http://iverilog.icarus.com/

[4] pytest
https://docs.pytest.org/en/stable/

[5] Yosys Open Synthesis Suite
http://www.clifford.at/yosys/

[6] Qflow 1.3: An Open-Source Digital Synthesis Flow
http://opencircuitdesign.com/qflow/

[7] Janusz Biernat, AK1-7-09 Szybkie sumatory.doc, 4 listopada 2009
https://www.lucc.pl/inf/architektura_komputerow_1/2009_-_7_szybkie_sumatory.pdf

[8] Janusz Biernat, 10-06-Szybkie sumatory.doc ,2 października 2006
https://www.lucc.pl/inf/architektura_komputerow_1/2006_-_szybkie_sumatory.pdf

[9] Clifford Wolf, "Yosys Manual", str 14
http://www.clifford.at/yosys/files/yosys_manual.pdf ,

[10] Qflow 1.4 Digital Synthesis Flow Reference Page
http://opencircuitdesign.com/qflow/reference.html

[11] GNU Make https://www.gnu.org/software/make/

[12] Computer Aids for VLSI Design, Steven M. Rubin, Appendix C: GDS II Format
https://www.rulabinsky.com/cavd/text/chapc.html