

*Politechnika Wrocławskaw
Wydział Elektroniki
Informatyka*

ARCHITEKTURA KOMPUTERÓW 2 - PROJEKT

IMPLEMENTACJA FIZYCZNA UKŁADÓW CYFROWYCH

Skład grupy projektowej:

Alicja Myśliwiec 248867
Daria Milczarska 248894

Termin zajęć:

Czwartek 17.05 - 18.45 TP

Prowadzący:

Dr inż. Piotr Patronik

Wrocław, 15.06.2020 r.

SPIS TREŚCI

1. WSTĘP - ANALIZA LITERATURY.....	4
1.1. VERILOG.....	4
1.2. YOSYS.....	5
1.3. QFLOW	5
2. SUMATOR.....	6
3. SUMATOR PREFIKSOWY	7
3.1. SCHEMAT DODAWANIA X+Y.....	8
3.2. WYTWARZANIE I PROPAGACJA PRZENIESIEŃ W DODAWANIU.....	9
3.3. SCHEMAT BLOKU GP	10
3.4. SIEĆ PREFIKSOWA GP	10
4. RODZAJE SUMATORÓW.....	11
4.1. SUMATOR KOGGE - STONE.....	11
4.2. SUMATOR BRENT - KUNG	11
4.3. SUMATOR LADNER - FISHER	12
4.4. SUMATOR HAN - CARLSON	13
4.5. SUMATOR KNOWLES	13
4.6. SUMATOR SKLANSKY	14
4.7. PORÓWNANIE SUMATORÓW	14
5. 6-BITOWY SUMATOR.....	15
5.1. MODUŁ GP	15
5.2. MODUŁ BLACK	16
5.3. MODUŁ GRAY	16
5.4. MODUŁ SKLANSKY_0	16
5.5. MODUŁ SKLANSKY_1	17
5.6. MODUŁ SKLANSKY_2	17
5.7. MODUŁ SKLANSKY_3	17
5.8. MODUŁ SKLANSKY_4	18
5.9. MODUŁ SKLANSKY_ALL.....	18
6. TEST WYCZERPUJĄCY.....	19
7. SYNTEZA LOGICZNA UKŁADU - YOSYS	20
7.1. MODUŁ GP	20
7.2. MODUŁ GRAY	20
7.3. MODUŁ BLACK	20
7.4. MODUŁ SKLANSKY_0.....	21
7.5. MODUŁ SKLANSKY_1.....	22
7.6. MODUŁ SKLANSKY_2.....	23
7.7. MODUŁ SKLANSKY_3.....	23

7.8. MODUŁ SKLANSKY_4.....	24
7.9. MODUŁ SKLANSKY_ALL	24
7.10. STATYSTYKI	24
8. SYNTEZA FIZYCZNA UKŁADU - QFLOW	25
8.1. MODEL FIZYCZNY SUMATORA	25
8.2. OPÓŹNIENIE NA ŚCIEŻKACH - 3 ŚCIEŻKI Z MAKSYMALNYM OPÓŹNIENIEM	25
8.3. OPÓŹNIENIE NA ŚCIEŻKACH - 3 ŚCIEŻKI Z MINIMALNYM OPÓŹNIENIEM.....	26
8.4. STATYSTYKI	27
9. WNIOSKI.....	28
10. ŹRÓDŁA I BIBLIOGRAFIA	29
11. SPIS RYSUNKÓW.....	30
12. SPIS TABEL.....	30

1. Wstęp - analiza literatury

1.1. Verilog

Verilog (znormalizowany jako IEEE 1364) jest językiem opisu sprzętu (HDL) używanym do projektowania, modelowania oraz symulacji układów cyfrowych i systemów elektronicznych, zwłaszcza typu SIC i FPGA. Jest to najczęściej stosowane narzędzie w projektowaniu i weryfikacji układów cyfrowych na poziomie abstrakcji. Stosowany jest również w analizie obwodów analogowych i obwodów sygnałów mieszanych, a także w projektowaniu obwodów genetycznych. Standard Verilog (IEEE 1364 - 2005) został włączony do SystemVerilog standardu w 2009 roku, tworząc tym samym Standard IEEE 1800 - 2009. Od tego czasu Verilog jest oficjalnie częścią języka SystemVerilog. Obecnie jest to Standrad IEEE 1800 - 2017.

Verilog został stworzony około roku 1984 przez Phila Moorby'ego w firmie Gateway Design Automation. W roku 1985 ukazała się pierwsza wersja handlowa Veriloga. W roku 1986 powstał symulator używający tego języka – Verilog-XL. Rok 1988 przyniósł pierwsze narzędzie do syntezы sprzętu na podstawie opisu w Verilogu, wyprodukowane przez firmę Synopsys. W roku 1989 Gateway Design Automation została przejęta przez firmę Cadence, która zdecydowała uczynić Verilog otwartym standardem w roku 1990. W celu sprawowania nadzoru nad rozwojem języka powołano Open Verilog International(OVI). Ta organizacja doprowadziła do powstania pierwszego oficjalnego standardu języka – IEEEStd 1364-1995 – w roku 1995. Open Verilog International i VHDL International połączyły się w roku 2000, tworząc nową organizację – Accellera. Na podstawie życzeń użytkowników i sugestii producentów oprogramowania standard został zrewidowany w roku 2001 (IEEE Std 1364-2001). W roku 2002 Accellera rozpoczęła proces gruntownych ulepszeń w Verilogu. Ze względu na znaczną ilość zmian i ulepszeń, zdecydowano się utworzyć nowy projekt – SystemVerilog – przy zachowaniu dużego poziomu zgodności z oryginalnym Verilogiem. Pod koniec roku 2005 zaaprobowano nową wersję standardu Veriloga IEEE Std 1364-2005 i pierwszą wersję standardu SystemVeriloga IEEE Std 1800-2005.

Początkowo Verilog był nieco niejednorodnym językiem opisu sprzętu. Posiadał doskonałe środki opisu na niskim poziomie, bardzo dobrą obsługę poziomów pośrednich i akceptowlą obsługą wyższych poziomów. Późniejsze zmiany uczyniły język bardziej jednorodnym, ulepszając składnię i poprawiając obsługę wyższych poziomów opisu. Języki używane do opisów sprzętu, takie jak Verilog, są podobne do programowania w tradycyjnych, ponieważ obejmują sposoby opisywania propagacji czasu i moc sygnałów. Istnieją dwa rodzaje operatorów przypisania: przypisanie blokujące (=) oraz przypisanie nieblokujące (<=). Przypisanie nieblokujące umożliwia projektantom opisywanie aktualizacji automatów bez konieczności deklarowania i używania tymczasowych zmiennych pamięci. Pojęcia te są częścią semantyki języka Verilog, zatem projektanci mogą w niedługim czasie stworzyć opisy dużych obwodów w stosunkowo zwartej i zwięzłej formie. Verilog umożliwia programistom używanie graficznego oprogramowania do przechwytywania schematów i specjalnych programów do dokumentowania i symulacji układów cyfrowych. Podobnie jak w języku C, Verilog rozróżnia małe i wielkie litery oraz ma podstawowy preprocesor. Słowa kluczowe (if/else, for, while, case itp.) są równoważne, a operatory pierwszeństwa są zgodne z C. Różnice składniowe obejmują wymagane szerokości bitów dla deklaracji zmiennych, rozgraniczenie bloków proceduralnych (Verilog używa begin/end zamiast nawiasów klamrowych {}) i inne niewielkie różnice. Verilog wymaga, aby zmiennie miały określony rozmiar, natomiast w C wielkości te przyjmuje się na podstawie typu zmiennej na przykład typ całkowity int może wynosić 8 bitów. Podstawową jednostką opisu hierarchii projektu w Verilogu jest moduł. Moduły komunikują się z innymi modułami poprzez zestaw deklarowanych portówwejściowych, wyjściowych i dwukierunkowych. Moduły niższego poziomu mogą być łatwo używane wewnątrz modułów wyższego poziomu przez podanie nazwy modułu, etykiety, listy wartości parametrów modułu (opcjonalnie) oraz listy połączeń z innymi modułami. Opis funkcjonalny modułów jest możliwy dzięki wspólniejącym procesom typu always i initial. Wewnątrz

procesów można używać instrukcji znanych z innych, tradycyjnych języków (instrukcje przypisania, wywołania funkcji, pętle for, while, repeat itp.). Instrukcje sekwencyjne umieszczone są w bloku begin/end i wykonywane w sekwencji, jednak same bloki wykonywane są jednocześnie, co czyni Verilog językiem przepływu danych. Koncepcja Verilogu obejmuje zarówno wartości sygnału (4 stany: „1, 0, floating, undefined”), jak i siły sygnału (silny, słaby itp.). System ten umożliwia abstrakcyjne modelowanie współdzielonych linii sygnałowych, w których wiele źródeł napędza wspólną sieć. Podzbiór instrukcji w języku Verilog można syntezować. Moduły Verilog zgodne są ze stylem kodowania syntezowalnego, znanym jako RTL (register-transfer level), mogą być fizycznie zrealizowane za pomocą oprogramowania do syntezy. Oprogramowanie do syntezy algorytmicznie przekształca (abstrakcyjne) źródło Verilog w logicznie równoważny opis składający się tylko z elementarnych operacji logicznych (AND, OR, NOT, flip-flops itp.), które są dostępne w określonej technologii FPGAlub VLSI. Układ cyfrowy musi komunikować się z otoczeniem, pobierać informacje do wewnętrz i wyprowadzać wyniki. Interfejs jest częścią układu odpowiedzialną za komunikację, a wewnętrz układu, czyli ciało, odpowiada za przetwarzanie informacji pobranych z otoczenia. Verilog pozwala specyfikować zarówno interfejs jak i ciało układu.

1.2. Yosys

Yosys to struktura narzędzi do syntezy RTL. Obecnie ma szerokie wsparcie Verilog-2005 i zapewnia podstawowy zestaw algorytmów syntezy dla różnych domen aplikacji. Yosys można dostosować do wykonywania dowolnego zadania syntezy, łącząc istniejące przebiegi (algorytmy) za pomocą skryptów syntezy i dodając dodatkowe przejścia w razie potrzeby, rozszerzając bazę kodu yosys C++. Yosys jest wolnym oprogramowaniem licencjonowanym na licencji ISC (licencja zgodna z GPL, która jest podobna do licencji MIT lub 2-klauzulowej licencji BSD). Yosys jako dane wejściowe przyjmuje opis projektu behawioralnego i generuje RTL, logiczną bramę lub opis poziomu bramki fizycznej projektu jako dane wyjściowe. Istnieje szeroki zakres poleceń (przebiegów syntezy) Yosys, za pomocą którego można wykonywać szeroki zakres zadań syntezy w dziedzinie behawioralnej, rtl i syntezy logicznej. Yosys został zaprojektowany tak, aby był rozszerzalny i dlatego stanowi dobrą podstawę do wdrożenia niestandardowych narzędzi syntezy do specjalistycznych zadań. Yosys można przekierować, a dodanie obsługi dodatkowych celów nie jest bardzo trudne. W tej chwili Yosys jest dostarczany z obsługą syntezy ASIC, FPGA iCE40, FPGA Xilinx 7-Series, urządzenia Silego GreenPAK4 i FPGA Gowinsemi GW1N / GW2A. We wszystkich tych przypadkach Yosys wykonuje tylko syntezę. Aby uzyskać pełny przepływ ASIC jest potrzebny Qflow. Dodatkowe pliki wejściowe dla Yosys zawierają mały skrypt syntezy i dodatkowy plik Verilog opisujący, w jaki sposób Yosys powinien mapować konstrukty RTL do biblioteki. Oprócz prostej syntezy HDL Yosys może być wykorzystywany do szerokiego zakresu zaawansowanych analiz i transformacji obwodów. Może wyodrębniać FSM i wykonywać różne operacje wyodrębnionych FSM, takich jak przekodowanie i przeniesienie dodatkowej funkcji z logikisieci do FSM. Można to wykorzystać do sterowania kontrolą logiki ogólnej komórki FSM, np. TR-FSM. Yosys obsługuje również mapowanie technologii, wykrywając izomorfizm obwodów, umożliwiając realizację bogatszej funkcji logicznej niż wykorzystywane przez komórki RTL.

1.3. Qflow

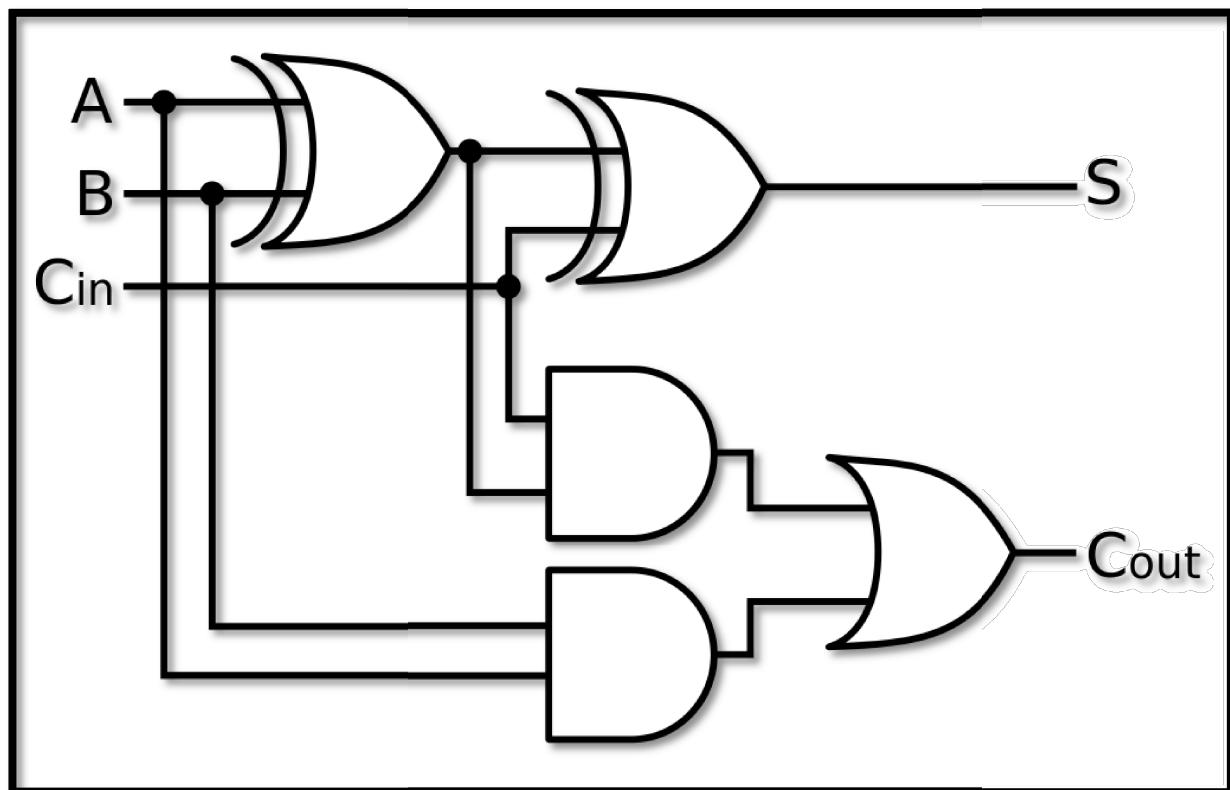
Cyfrowy przepływ syntezy to zestaw narzędzi i metod używanych do przekształcenia układu obwodu napisanego w języku behawioralnym wysokiego poziomu, takim jak Verilog lub VHDL, w obwód fizyczny, który może być kodem konfiguracji dla układu FPGA, takiego jak układ Xilinx, Altera lub układ w określonej technologii procesu wytwarzania, który mógłby stać się częścią ukształtowanego układu scalonego.

Qflow to kompletny łańcuch narzędzi do syntezy obwodów cyfrowych, poczynając od źródła Verilog, a kończąc na fizycznym układzie dla konkretnego procesu produkcji docelowej. W świecie elektroniki

komercyjnej synteza cyfrowa z docelowym zastosowaniem układu czipowego jest zwykle pakowana w duże systemy oprogramowania EDA, takie jak Cadence lub Synopsys. Nie należy zakładać, że łańcuch narzędzi Qflow można wykorzystać do stworzenia mikroprocesorów wielogigerowych nowej generacji. Ale łańcuch narzędzi Qflow doskonale obsługuje cyfrowe podsystemy potrzebne wielu układom, w tym komunikację między hostem a urządzeniem (na przykład SPI i I2C), przetwarzanie sygnałów (filtry cyfrowe, modulatory sigma-delta), jednostki arytmetyczne itd. Wczesne wersje przepływu cyfrowego Qflow zostały wykorzystane do stworzenia układów cyfrowych używanych w wysokowydajnych komercyjnych układach scalonych. Qflow to rozwijający się zestaw narzędzi, który wykorzystuje najlepsze dostępne komponenty.

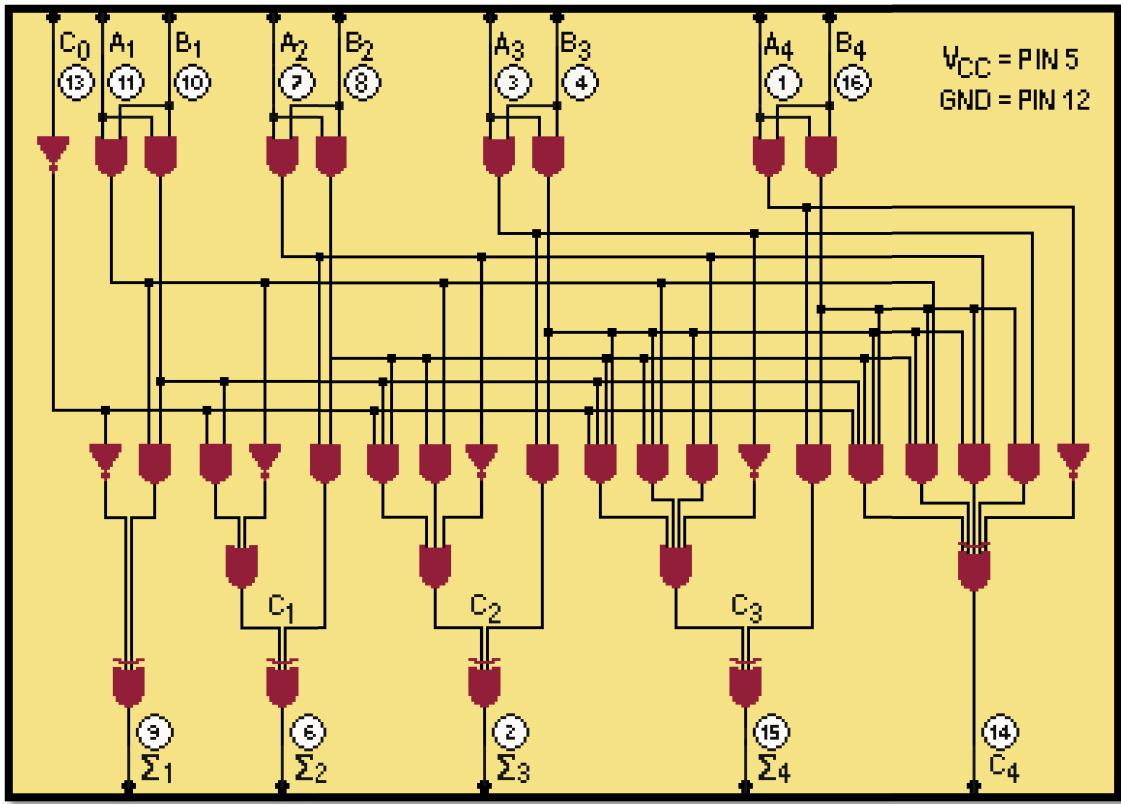
2. Sumator

Sumator to cyfrowy układ kombinacyjny, który wykonuje operacje dodawania dwóch (lub więcej) liczb dwójkowych. Sumatory można podzielić na szeregowe, które podczas każdej operacji dodają dwa bity składników oraz bit przeniesienia oraz równoległe, które są wielopozycyjne i dodają do siebie jednocześnie bity ze wszystkich pozycji, a przeniesienie realizowane jest w zależności od sposobu połączenia sumatorów jednobitowych. Z kolei sumatory równoległe obejmują dwie grupy: z przeniesieniami szeregowymi i z przeniesieniami równoległymi. Sumator z przeniesieniami równoległymi jest szybszy niż sumator z przeniesieniami szeregowymi. Pełny sumator jednobitowy wygląda tak:



Rysunek 1: Pełny sumator jednobitowy

Dodawanie dwóch wielobitowych liczb dwójkowych może być realizowane szeregowo lub równolegle. Stąd wynika podział sumatorów na szeregowe i równoległe. W realizacji szeregowej kolejne pary bitów są sumowane wraz z odpowiednim przeniesieniem w szeregu cykli, dopóki dodawanie całego słowa nie zostanie zakończone. Przy dodawaniu równoległym, poszczególne pary bitów są sumowane za pomocą osobnych sumatorów, a przeniesienie z każdej pozycji jest kierowane do sumatora pozycji następnej. Proces dodawania przebiega szybciej w sumatorach równoległych. Poniżej przedstawiono schemat 4-bitowego sumatora w połączeniu kaskadowym wejść i wyjścia przeniesienia:

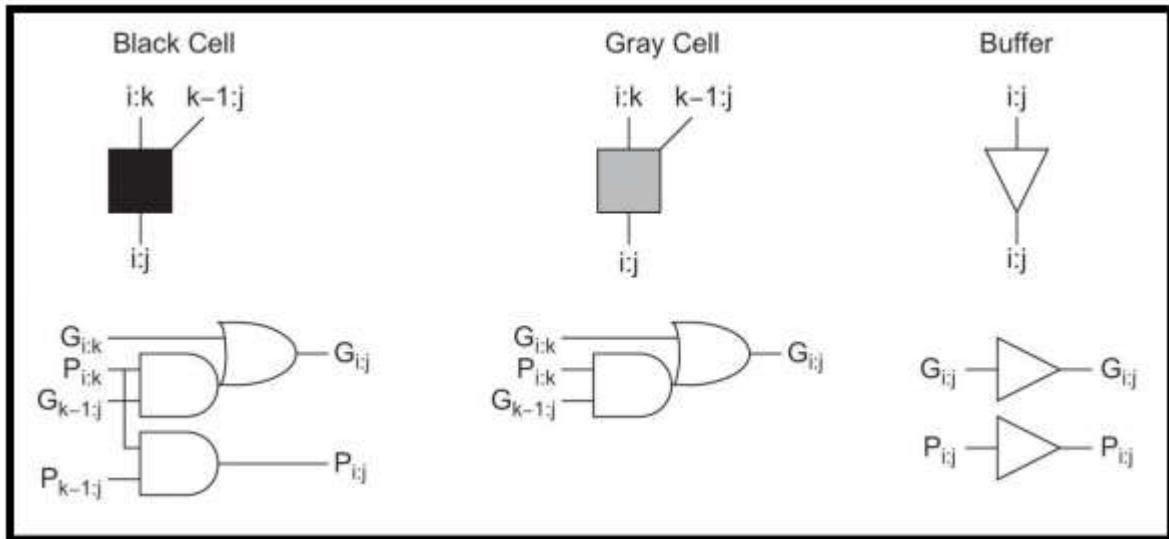


Rysunek 2: Schemat 4-bitowego sumatora

3. Sumator prefiksowy

Sumatory prefiksowe wykonują równoległe dodawanie i cyfrowe przetwarzanie sygnałów, co jest ważne w aspekcie mikroprocesorów. Wykorzystanie takiego układu poprawia złożoność logiczną i opóźnienie czasowe, zatem takie sumatory są potrzebnymi elementami w szybkich obwodach arytmetycznych. Sumatory drzewiaste generują przeniesienie równoległe, zatem są w stanie liczyć szybko, ale ze zwiększoną powierzchnią i mocą. Zmniejsza się całkowita liczba poziomów logicznych poprzez szacowanie przeniesień równoległych. Jest to zaleta sumatorów prefiksowych. W porównaniu ze ścieżką przenoszenia innych sumatorów, są bardziej niezawodne pod względem prędkości, ponieważ złożoność opóźnienia jest rzędu $O(\log(2N))$.

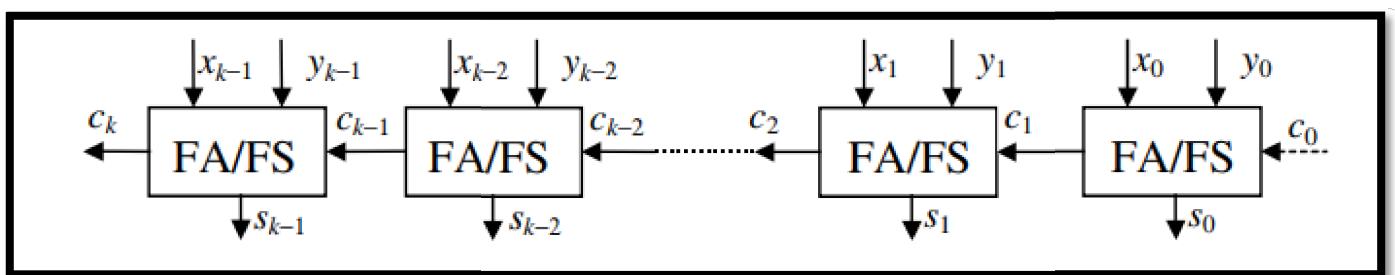
Obliczenie prefiksu ma następujące kroki: Obliczenie generacji oraz propagacji przeniesienia za pomocą bitów wejściowych w odniesieniu do każdej pary bitów w A i B. Sygnały generacji i propagacji są określone wzorami, a odpowiednie obliczenie prefiksu pozwala wyznaczyć wszystkie sygnały przeniesienia. Po obliczeniu przeniesień, dzielone one są na mniejsze fragmenty. W równoległych sumatorach ważny jest blok operacyjny, który składa się z trzech części: czarna komórka, szara komórka, bufor. Czarne komórki obliczają zarówno generowanie jak i propagowanie bitów. Podobnie jest w obliczeniach przeniesień dokonanych, które są używane na etapie przetwarzania końcowego do obliczenia sumy. Bufory służą do zrównoważenia efektu ładowania. Ostatni etap składa się z prostego obwodu sumatora do wygenerowania bitu końcowego.



Rysunek 3: Schematy elementów sumatora prefiksowego

3.1. Schemat dodawania X+Y

$$s_i = x_i \oplus y_i \oplus c_i \\ c_{i+1} = x_i y_i + (x_i \oplus y_i) c_i = c_{i+1} = x_i y_i + (x_i + y_i) c_i$$



Rysunek 4: Schemat dodawania

Obliczenie sumy na pozycji i -tej wymaga przeniesienia z pozycji $i-1$. Czas wytwarzania sumy jest stały od chwili ustalenia przeniesienia. Gwarantowany czas wykonania dodawania zależy od najdłuższego czasu przesłania zmiany przeniesienia z pozycji najwyższej, natomiast czas sekwencyjnego dodawania n-pozycyjnego jest iloczynem - nT . Przyspieszenie czasu dodawania można zrealizować poprzez skrócenie czasu propagacji przeniesienia, składanie sum tymczasowych lub składanie sum redundantnych. Sumatory prefiksowe skracają czas propagacji przeniesienia poprzez wywarzanie przeniesień równoległych.

3.2. Wytwarzanie i propagacja przeniesień w dodawaniu

Funkcja przeniesienia może mieć jedną z równoważnych form:

$$c_{i+1} = x_i y_i + (x_i \oplus y_i) c_i = c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

Składowymi wyrażenia są:

- Funkcja wytwarzania (generowania) przeniesienia, określająca warunki, przy których przeniesienie wyjściowe $c_{i+1} = 1$, niezależnie od c_i :

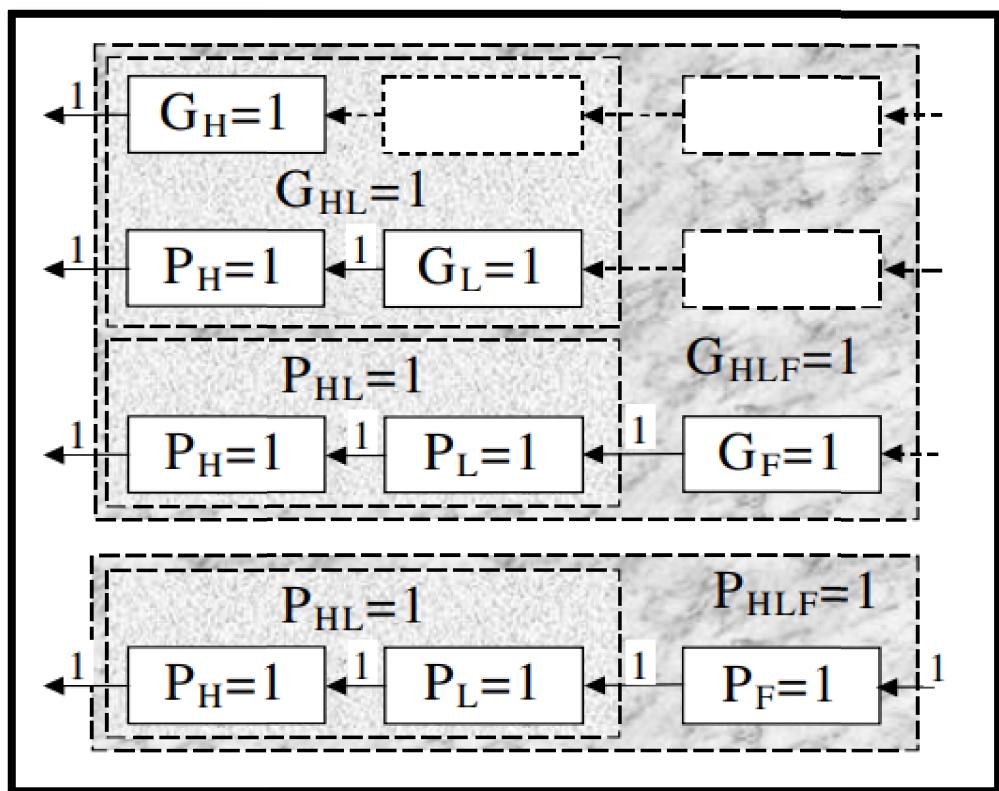
$$g_i = \bar{x}_i y_i$$

- Funkcja półsumy, która także określa warunki przekazywania (propagacji) przeniesienia ($x_i \neq y_i \Rightarrow c_{i+1} = c_i$)

$$h_i = \bar{x}_i \oplus y_i$$

W wyrażeniach na przeniesienia może ją zastąpić (nadmiarowa) funkcja przekazywania przeniesienia (\bar{p}_i – funkcja wygaszania)

$$p_i = \bar{x}_i + y_i$$



Rysunek 5: Schemat propagacji i generowania przeniesień

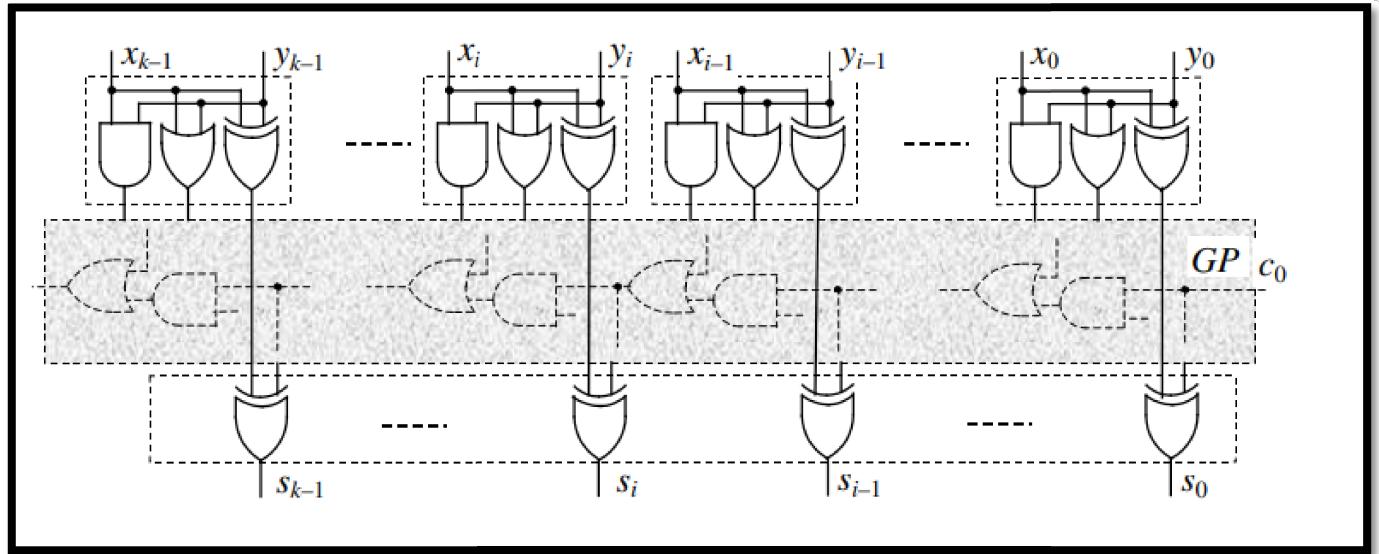
$$\begin{aligned} c_{out} &= G_H + P_H G_L + P_H P_L c_{in/L} = G_{HL} + P_{HLC} c_{in/L} \\ c_{out} &= G_{HL} + P_{HL} G_F + P_{HLC} P_F c_{in/F} = G_{HLF} + P_{HLC} c_{in/F} \end{aligned}$$

3.3. Schemat bloku GP

Schemat bloku GP przedstawia wytwarzanie wartości wszystkich przeniesień $c_i = G_{i-1:0} + P_{i-1:0}c_0$

$$s_i = h_i \oplus c_i$$

Jeżeli $c_i = 0$ to $c_i = G_{i-1:0}$. W tym wypadku $s_i = h_i \oplus c_i = s_i = h_i \oplus G_{i-1:0}$, w przeciwnym razie występuje problem silnie rozgałęzionego sygnału przeniesienia wejściowego c_0 . W celu uniknięcia rozgałęziania sygnału można dołączyć blok wejściowy CSA, redukujący sygnał, lub potraktować c_0 jako funkcję generowania przeniesienia z pozycji -1.



Rysunek 6: Blok GP

3.4. Sieć prefiksowa GP

Węzeł sieci GP realizuje funkcję:

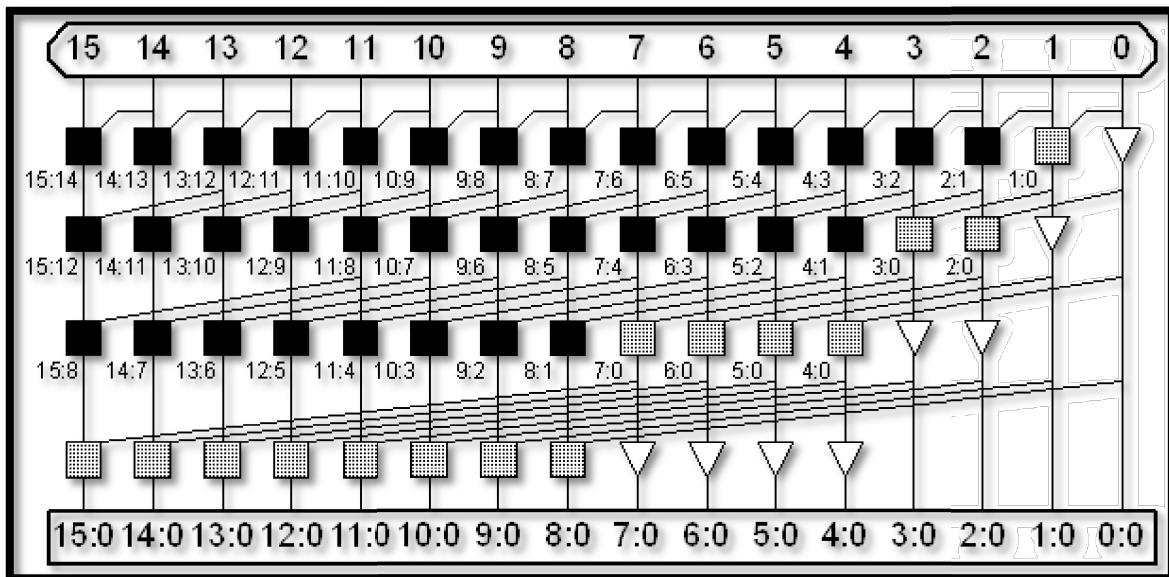
$$(G_{HL}, P_{HL}) = (G_H, P_H) \bullet (G_L, P_L) = (G_H + P_H G_L, P_H P_L)$$

Bloki H i L powinny być styczne, nie mogą być rozdzielone, mogą mieć część wspólną, struktura powinna być regularna dla $n = 2^k$ wejść, a w innych sytuacjach należy usunąć zbędne krawędzie.

4. Rodzaje sumatorów

4.1. Sumator Kogge - Stone

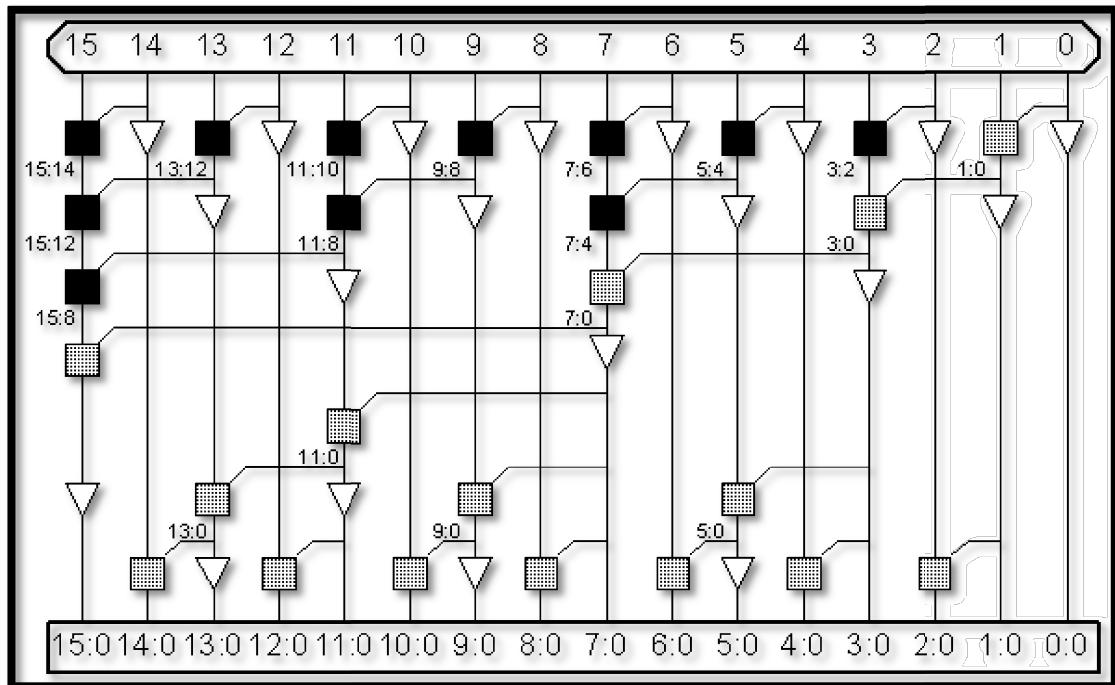
Każda kolumna produkuje zarówno sygnały propagacji jak i generacji. Generowane są sygnały, które są na końcu wprowadzane do bramki XOR razem z początkowo wyprodukowanymi sygnałami generacji i propagacji, aby powstała suma. Zaletą tego sumatora jest fakt, że generuje bit przeniesienia w średnim opóźnieniu $O(\log(2N))$, które daje najlepszą wydajność w obwodach zaimplementowanych w VLSI. Zmniejszana jest w dużym stopniu ścieżka krytyczna, dzięki czemu zwiększa się wydajność w implementacji sumatorów wyższych rzędów, takich jak 32-bitowe, 64-bitowe czy 128-bitowe.



Rysunek 7: Sumator Kogge - Stone

4.2. Sumator Brent - Kung

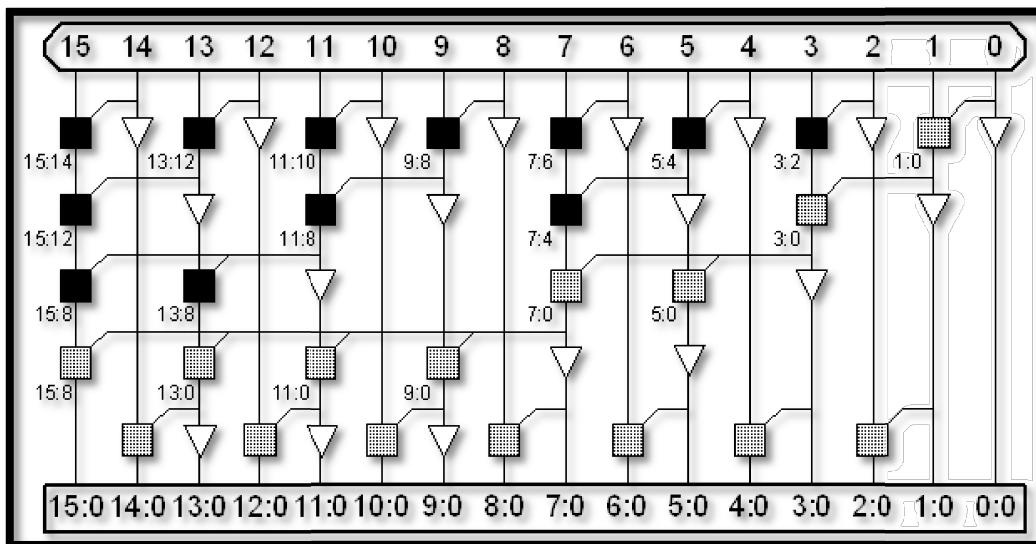
Sumator Brent-Kung oblicza najpierw nieparzyste bity prefiksu i wtedy (jak na rysunku) obliczone przenosi c₁, c₃, c₇, c₁₅. Obliczane są bity prefiksów dla dwóch grup wejściowych. Działają one jak dane wejściowe dla kolejnych grup komórek w celu obliczenia prefiksów dla grup 4-bitowych, które z kolei obliczają prefiksy dla grup 8-bitowych i tak dalej. Te prefiksy używane są do obliczania przeniesień dla każdego bitu na następnym poziomie. Wygenerowane przenesienia dołączane są do bramki XOR z grupą bitów propagacji z następnego etapu do obliczenia sumy bitów. Zaletą tego sumatora jest to, że zajmuje mniej miejsca, co sprawia, że zaimplementowany sumator jest mniej załoczony w porównaniu do innych sumatorów prefiksowych takich jak Sumator Kogge-Stone. Ten sumator ma maksymalną głębokość logiczną, co zwiększa opóźnienie oraz minimalną liczbę węzłów, co zmniejsza obszar. To daje również zmniejszenie opóźnienia, gdy zaimplementowane będą sumatory wyższych rzędów bez uszczerbku na wydajności sumatora.



Rysunek 8: Sumator Brent - Kung

4.3.Sumator Ladner - Fisher

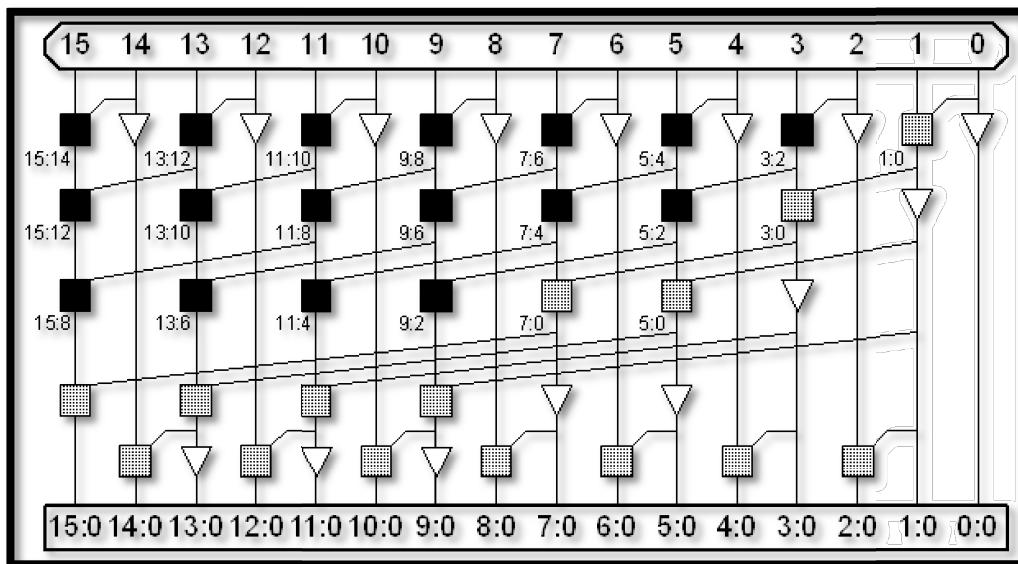
Ta struktura reprezentuje pośrednio drzewo Brent-Kung i Sklansky. W początkowej fazie sumator oblicza prefiksy dla nieparzystych numerów bitów i wykorzystuje kolejny dodatkowy etap do propagacji nieparzystych numerów bitów do pozycji parzystych. Ten sumator ma minimalną głębokość logiczną.



Rysunek 9: Sumator Ladner - Fisher

4.4.Sumator Han - Carlson

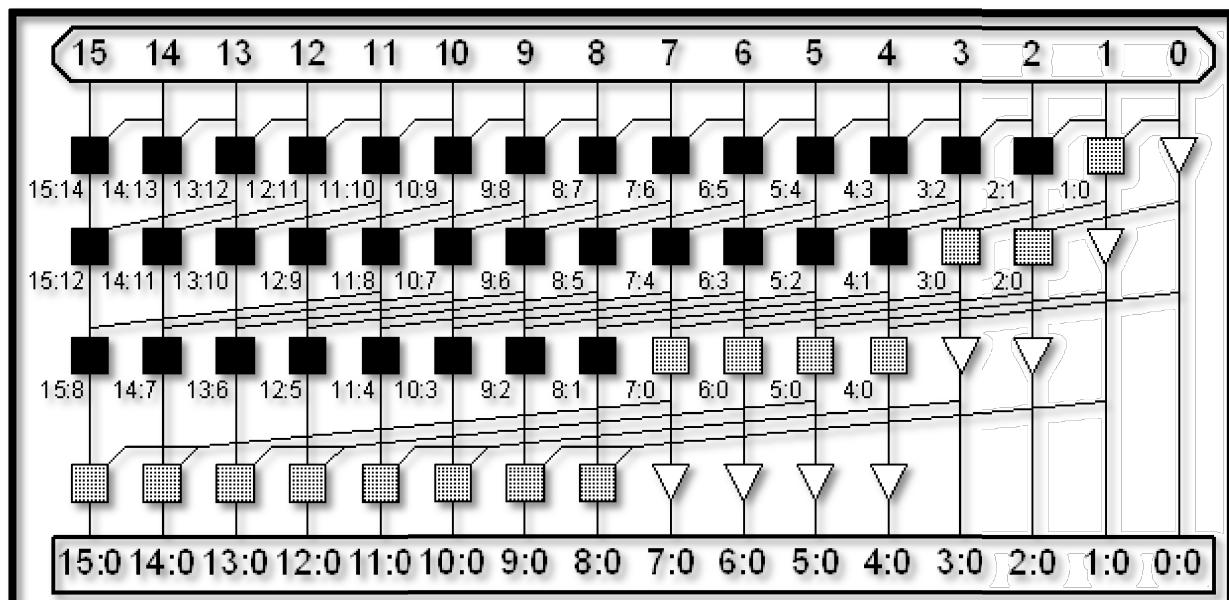
Sumator Han-Carlson można traktować jako reprezentującą hybrydową strukturę drzewa Brent-Kung i Kogge-Stone. Może być również reprezentowany jako rozproszona wersja Sumatora Kogge-Stone. Przeniesienie generacji HC obejmuje pięć etapów. Centralne trzy etapy przypominają strukturę drzewa Kogge-Stone. Ta struktura jest inna niż drzewo KS, ponieważ wykonuje scalanie operacji na parzystych bitach (tak jak na rysunku) przez obliczanie c₂, c₄, c₆, c₈, c₁₀, c₁₂, c₁₄ i generowanie, propagowanie operacji na bitach nieparzystych. Na końcu te grupy rozpowszechniają sygnały połączone z poprzednimi nieparzystymi bitami przeniesienia i produkują finalne bity przeniesienia. Ten sumator wykorzystuje mniej czarnych komórek i ma krótszą złożoność niż sumator KS. Złożoność może być zmniejszona kosztem etapu dodatkowego dla ścieki łączenia.



Rysunek 10: Sumator Han - Carlson

4.5.Sumator Knowles

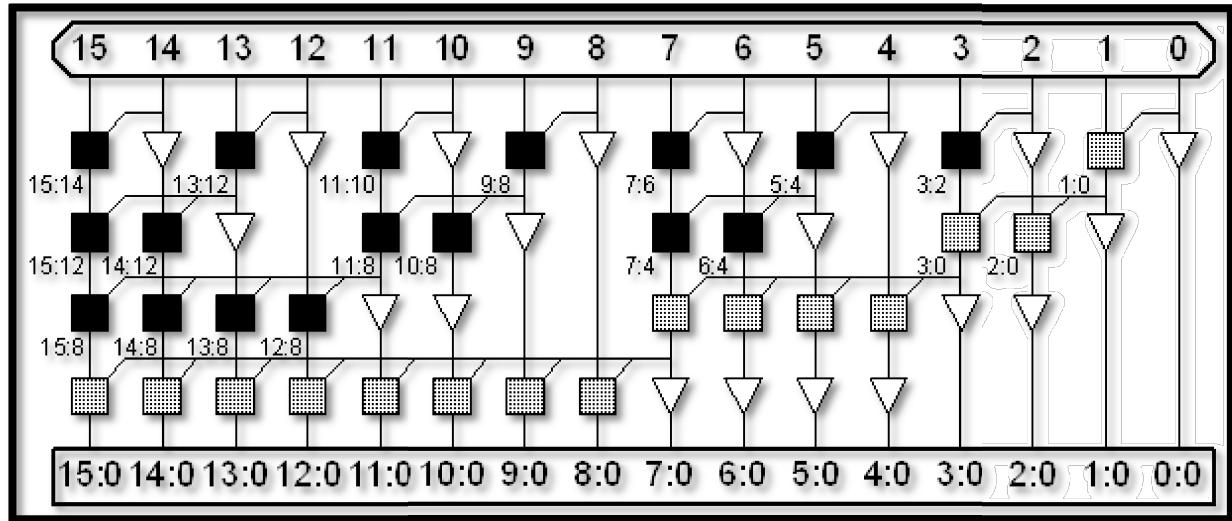
Drzewo Knowles to struktura, która jest połączeniem sieci drzew Kogge-Stone i Sumatora Sklansky'ego.



Rysunek 11: Sumator Knowles

4.6.Sumator Sklansky

Ten sumator ma prostą strukturę drzewa prefiksów. Reprezentuje strukturę dzielenia i pokonywania. Oblicza prefiksy rekurencyjnie dla grup 2-bitowych, 4-bitowych, 8-bitowych, a następnie 16-bitowych i tak dalej poprzez dodanie dwóch mniejszych sumatorów na każdym poziomie. Dzięki koncepcji dziej i zwycięzaj, opóźnienie wynosi etapów $\log_2 n$ poprzez obliczanie pośrednich bitów prefiku wraz z prefiksami z dużych grup. Zaletą tego sumatora jest prosta i regularna architektura.



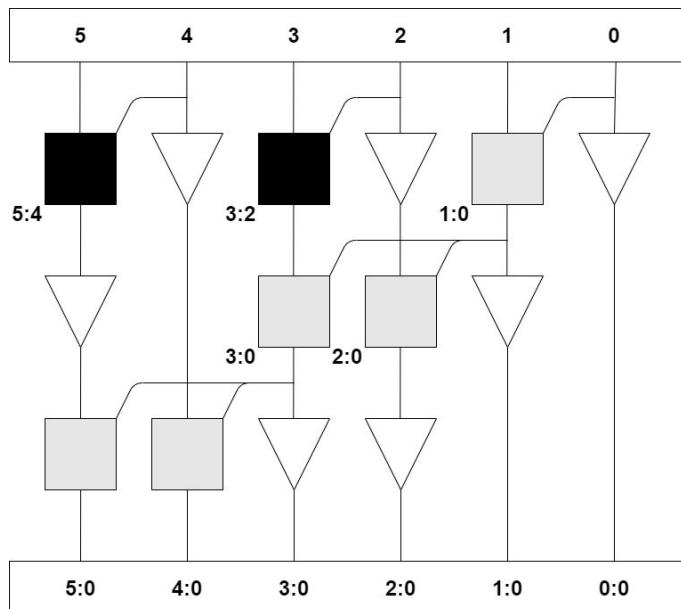
Rysunek 12: Sumator Sklansky

4.7.Porównanie sumatorów

Tabela 1: Parametry sieci GP jako elementy sumatora prefiksowego

Typ struktury	Liczba ogniw GP	Liczba poziomów	Obciążenie	Przełączenia
RCA	$\frac{2}{3} n$	$n - 1$	2	$\frac{n}{2}$
Sklansky	$\frac{1}{2} n \log_2 n$	$\log_2 n$	$\frac{n}{2}$	$\frac{1}{4} n \log_2 n$
Brent-Kung	$2n - n \log_2 n - 2$	$2 \log_2 n - 2$	$\log_2 n + 1$	$\sim \frac{3}{8} n \log_2 n$
Kogge-Stone	$n \log_2 n - n + 1$	$\log_2 n$	2	$\frac{1}{2} n \log_2 n$
Han-Carlson	$\frac{1}{2} n \log_2 n$	$\log_2 n + 1$	2	$\frac{1}{4} n \log_2 n$

5. 6-bitowy sumator



Rysunek 13: 6-bitowy sumator Sklansky'ego

Po analizie literatury zdecydowałyśmy się na zaprojektowanie sumatora typu Sklansky. Nasz wybór podyktowany był przede wszystkim prostą i regularną architekturą oraz rekurencyjnym obliczaniem prefiksów. Sumator składa się z trzech modułów pomocniczych (gp, black i gray), pięciu modułów reprezentujących każdy z poziomów logicznych zbudowanego przez nas sumatora oraz modułu głównego reprezentującego cały sumator.

5.1. Moduł gp

Wylicza propagacje i generacje dla dwóch sygnałów jednabitowych według wzorów:

$$G_i = a_i \wedge b_i$$

$$P_i = a_i \oplus b_i$$

```
//moduł zwracający generacje i propagacje dwóch sygnałów
module gp (
    input x, y, //sygnały wejściowe
    output g, p // generacja i propagacja
);

assign g = x & y;
assign p = x ^ y;

endmodule
```

Rysunek 14: Moduł gp

5.2.Moduł black

Wylicza generacje i propagacje dla bloku (i:k) według wzorów:

$$G_{i:j} = G_{i:k} \vee (P_{i:k} \wedge G_{k-1:j})$$
$$P_{i:j} = P_{i:k} \wedge P_{k-1:j}$$

```
//moduł komórki czarnej
module black (
    input g, p, g_pop, p_pop,
    output o_g, o_p
);

assign o_g = g|(p & g_pop);
assign o_p = p & p_pop;

endmodule
```

Rysunek 15: Moduł Black

5.3.Moduł gray

Wylicza generacje dla bloku (i:k) według wzoru:

$$G_{i:j} = G_{i:k} \vee (P_{i:k} \wedge G_{k-1:j})$$

```
//moduł komórki szarej
module gray (
    input g, p, g_pop, //sygnały wejściowe
    output o_g
);

assign o_g = g|(p & g_pop);

endmodule
```

Rysunek 16: Moduł gray

5.4.Moduł Sklansky_0

Reprezentuje zerowy rzad w sumatorze Sklansky'ego, wylicza generacje i propagacje dla sygnałów jednabitowych.

```
//moduł przedstawiający zerowy rzad (generacji i propagacji) sumatora typu sklansky
module sklansky_0 (
    input wire [5:0]i_x,
    input wire [5:0]i_y,
    output wire [5:0]o_g,
    output wire [5:0]o_p
);

gp gp_0(i_x[0], i_y[0], o_g[0], o_p[0]);
gp gp_1(i_x[1], i_y[1], o_g[1], o_p[1]);
gp gp_2(i_x[2], i_y[2], o_g[2], o_p[2]);
gp gp_3(i_x[3], i_y[3], o_g[3], o_p[3]);
gp gp_4(i_x[4], i_y[4], o_g[4], o_p[4]);
gp gp_5(i_x[5], i_y[5], o_g[5], o_p[5]);

endmodule
```

Rysunek 17: Moduł Sklansky_0

5.5.Moduł Sklansky_1

Reprezentuje pierwszy rzad w sumatorze Sklansky'ego obliczający prefiksy dla grup 2-bitowych

```
//moduł przedstawiający pierwszy rzad (obliczający prefiksy dla grup 2 bitowych) sumatora typu sklansky
module sklansky_1 (
    input wire [5:0]i_g,
    input wire [5:0]i_p,
    output wire [5:0]o_g,
    output wire [3:0]o_p
);
assign o_p[0]=i_p[2];
assign o_p[2]=i_p[4];
assign o_g[0]=i_g[0];
assign o_g[2]=i_g[2];
assign o_g[4]=i_g[4];

gray gray_1(i_g[1], i_p[1], i_g[0], o_g[1]);
black black_3(i_g[3], i_p[3], i_g[2], i_p[2], o_g[3], o_p[1]);
black black_5(i_g[5], i_p[5], i_g[4], i_p[4], o_g[5], o_p[3]);

endmodule
```

Rysunek 18: Moduł Sklanksy_1

5.6.Moduł Sklansky_2

Reprezentuje drugi rzad w sumatorze Sklansky'ego obliczający prefiksy dla grup 4-bitowych.

```
//moduł przedstawiający drugi rzad (obliczający prefiksy dla grup 4 bitowych) sumatora typu sklansky
module sklansky_2 (
    input wire [5:0]i_g,
    input wire [3:0]i_p,
    output wire [5:0]o_g,
    output wire [1:0]o_p
);
assign o_p[0]=i_p[2];
assign o_p[1]=i_p[3];
assign o_g[0]=i_g[0];
assign o_g[1]=i_g[1];
assign o_g[4]=i_g[4];
assign o_g[5]=i_g[5];

gray gray_2(i_g[2], i_p[0], i_g[1], o_g[2]);
gray gray_3(i_g[3], i_p[1], i_g[1], o_g[3]);

endmodule
```

Rysunek 19: Moduł Sklanksy_2

5.7.Moduł Sklansky_3

Reprezentuje trzeci rzad w sumatorze Sklansky'ego obliczający prefiksy dla całego sumatora.

```
//moduł przedstawiający trzeci rzad (obliczający prefiksy dla grup 6 bitowych) sumatora typu sklansky
module sklansky_3 (
    input wire [5:0]i_g,
    input wire [1:0]i_p,
    output wire [5:0]o_g
);
assign o_g[0]=i_g[0];
assign o_g[1]=i_g[1];
assign o_g[2]=i_g[2];
assign o_g[3]=i_g[3];

gray gray_4(i_g[4], i_p[0], i_g[3], o_g[4]);
gray gray_5(i_g[5], i_p[1], i_g[3], o_g[5]);

endmodule
```

Rysunek 20: Moduł Sklanksy_3

5.8.Moduł Sklansky_4

Reprezentuje ostatni rząd w sumatorze Sklansky'ego zliczający generacje i propagacje.

```
//moduł przedstawiający ostatni rząd (zliczający wynik) sumatora typu sklansky
module sklansky_4 (
    input wire [5:0]i_g,
    input wire [5:0]i_p,
    output wire [5:0]o_s,
    output wire o_carry
);
    assign o_carry=i_g[5];
    assign o_s[0]=i_p[0];
    assign o_s[5:1]=i_p[5:1]^i_g[4:0];

endmodule
```

Rysunek 21: Moduł Sklanksy_4

5.9.Moduł Sklansky_all

Cały 6-bitowy sumator typu Sklansky'ego.

```
//moduł skupiający cały sumator 6-bitowy typu sklansky
module sklansky_all (
    input wire [5:0]i_x,
    input wire [5:0]i_y,
    output wire [5:0]o_s,
    output wire o_carry
);

    wire [5:0]g0; //generacja po wyjściu z modulu sklansky_0
    wire [5:0]p0; //propagacja po wyjściu z modulu sklansky_0
    wire [5:0]g1; //generacja po wyjściu z modulu sklansky_1
    wire [3:0]p1; //propagacja po wyjściu z modulu sklansky_1
    wire [5:0]g2; //generacja po wyjściu z modulu sklansky_2
    wire [1:0]p2; //propagacja po wyjściu z modulu sklansky_2
    wire [5:0]g3; //generacja po wyjściu z modulu sklansky_3

    sklansky_0 s0(i_x, i_y, g0, p0);
    sklansky_1 s1(g0, p0, g1, p1);
    sklansky_2 s2(g1, p1, g2, p2);
    sklansky_3 s3(g2, p2, g3);
    sklansky_4 s4(g3, p0, o_s, o_carry);

endmodule
```

Rysunek 22: Moduł Sklansky_all

6. Test wyczerpujący

Test sprawdzający poprawność implementacji sumatora 6-bitowego typu Sklansky polega na sprawdzeniu wszystkich możliwych kombinacji wejść do sumatora i porównanie wyjść z wynikiem dodawania tych dwóch sygnałów. Napisaliśmy program w języku Verilog, który w dwóch pętlach wprowadza do sumatora liczby od 0 do 63 i porównuje sygnały wyjściowe z sumatora z prawidłowym wynikiem. Jeśli wynik jest poprawny, zwiększamy zmienną przechowującą liczbę pozytywnie zrealizowanych testów o jeden, w przeciwnym wypadku robimy to samo ze zmienną przechowującą liczbę błędnych wyników. Na koniec wyświetlamy informacje o liczbie prawidłowych i błędnych wyników. Dodatkowo każde błędne działanie sumatora jest wyświetlane na ekranie.

```
module test_bench;

reg [5:0] a, b;      // 6-bit wejścia
wire [5:0] s;        // 6-bit suma na wyjściu
wire c6;            // przeniesienie - wyjście z sumatora
reg [6:0] sprawdzenie; // 6-bit wartość do sprawdzenia poprawności
integer i, j;        // zmienne do obsługi pętli
integer licz_poprawne; // licznik zawierający liczbę poprawnych wyników działania sumatora
integer licz_bledne; // licznik zawierający liczbę błędnych wyników działania sumatora

sklansky_all sklansky(a, b, s, c6); // zainicjowanie 6-bitowego sumatora typu sklansky

// rozpoczęcie sprawdzania
initial begin
    $display("Rozpoczynamy test...");
    // ustawienie liczników na 0
    licz_poprawne = 0; licz_bledne = 0;
    // przechodzimy przez wszystkie możliwe wartości pierwszej i drugiej liczby
    for (i = 0; i < 64; i = i + 1) begin
        a = i; //ustawienie pierwszego sygnału wejściowego
        for (j = 0; j < 64; j = j + 1) begin
            b = j; //ustawienie drugiego sygnału wejściowego
            sprawdzenie = a + b; //dodanie dwóch liczb w celu sprawdzenia poprawności
            #1; //odczekanie chwili by sumator zdążył dodać sygnały
            if ({c6, s} == sprawdzenie) begin //porównanie wyjścia sumatora z prawidłowym wynikiem
                licz_poprawne = licz_poprawne + 1;
            end else begin
                licz_bledne = licz_bledne + 1;
                // wyświetlenie wyniku pracy sumatora w przypadku błędного wyniku
                $display($time, " %b + %b = %b (%b)", a, b, {c6, s}, sprawdzenie);
            end
        end
    end
    // wyświetlenie liczby prawidłowych i błędnych dodaw
    $display("Prawidłowe = %d, błędne = %d", licz_poprawne, licz_bledne);
    $finish;
end
endmodule|
```

Rysunek 23: Test wyczerpujący

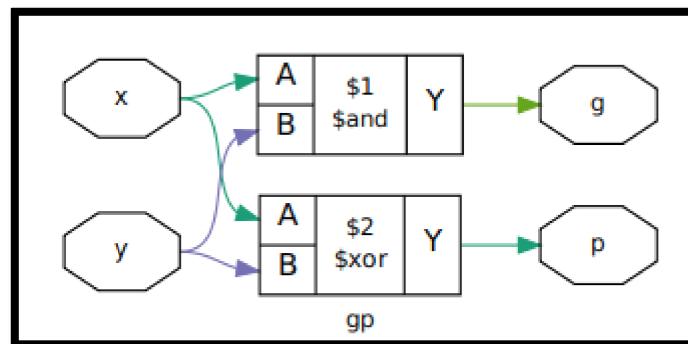
Poprawność działania sumatora została sprawdzona za pomocą symulatora „Icarus Verilog” i daje następujące rezultaty.

```
alicja@alicja:~/Pulpit$ iverilog -o test tes_bench.v sum.v
alicja@alicja:~/Pulpit$ vvp test
Rozpoczynamy test...
Prawidłowe =      4096, błędne =          0
```

Rysunek 24: Wyniki testu

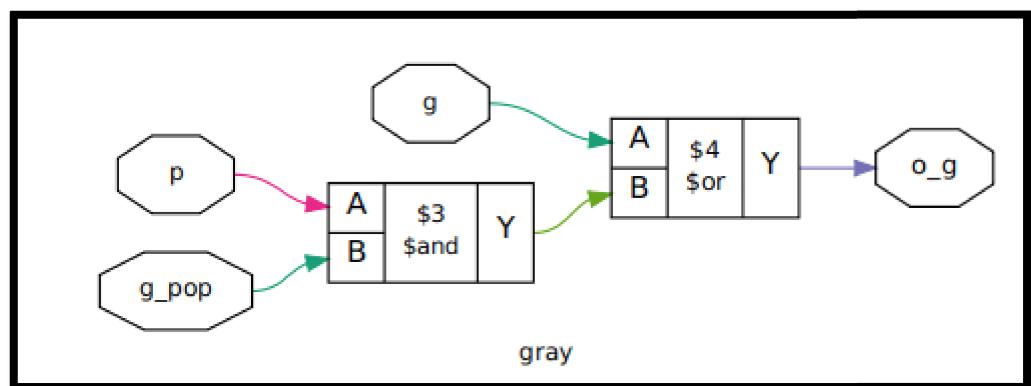
7. Synteza logiczna układu - Yosys

7.1.Moduł gp



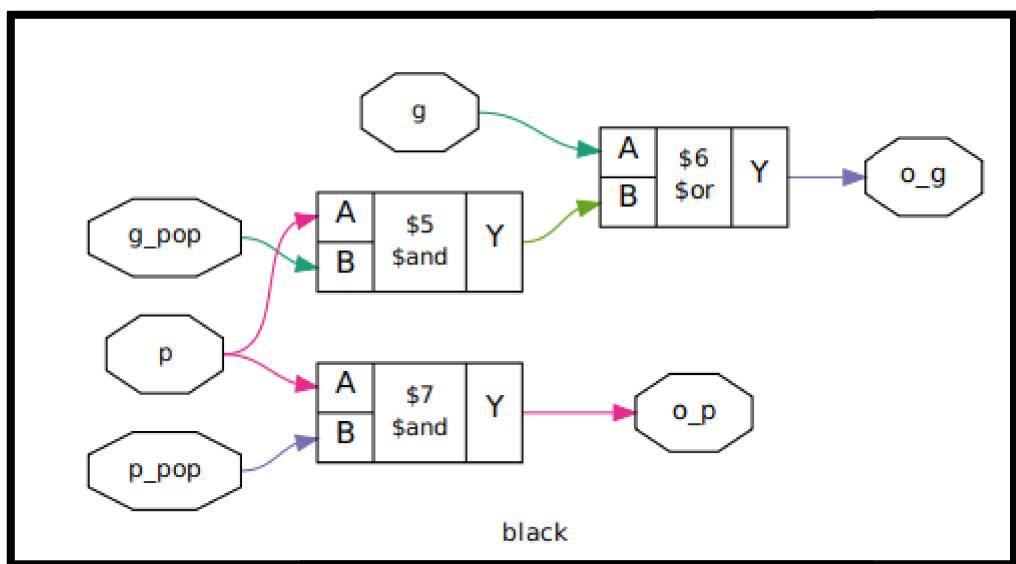
Rysunek 25: Moduł gp - synteza logiczna

7.2.Moduł gray



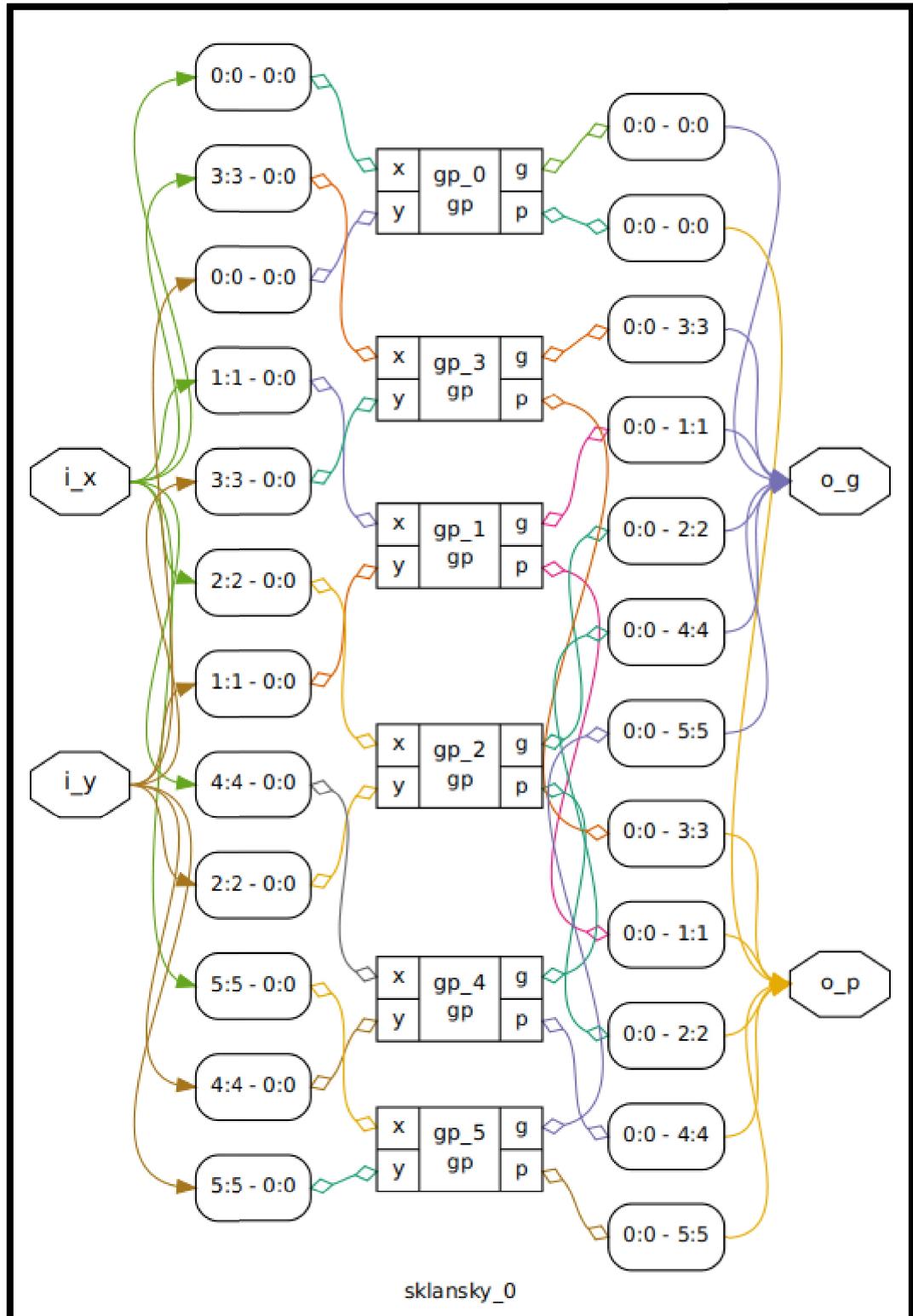
Rysunek 26: Moduł gray - synteza logiczna

7.3.Moduł black



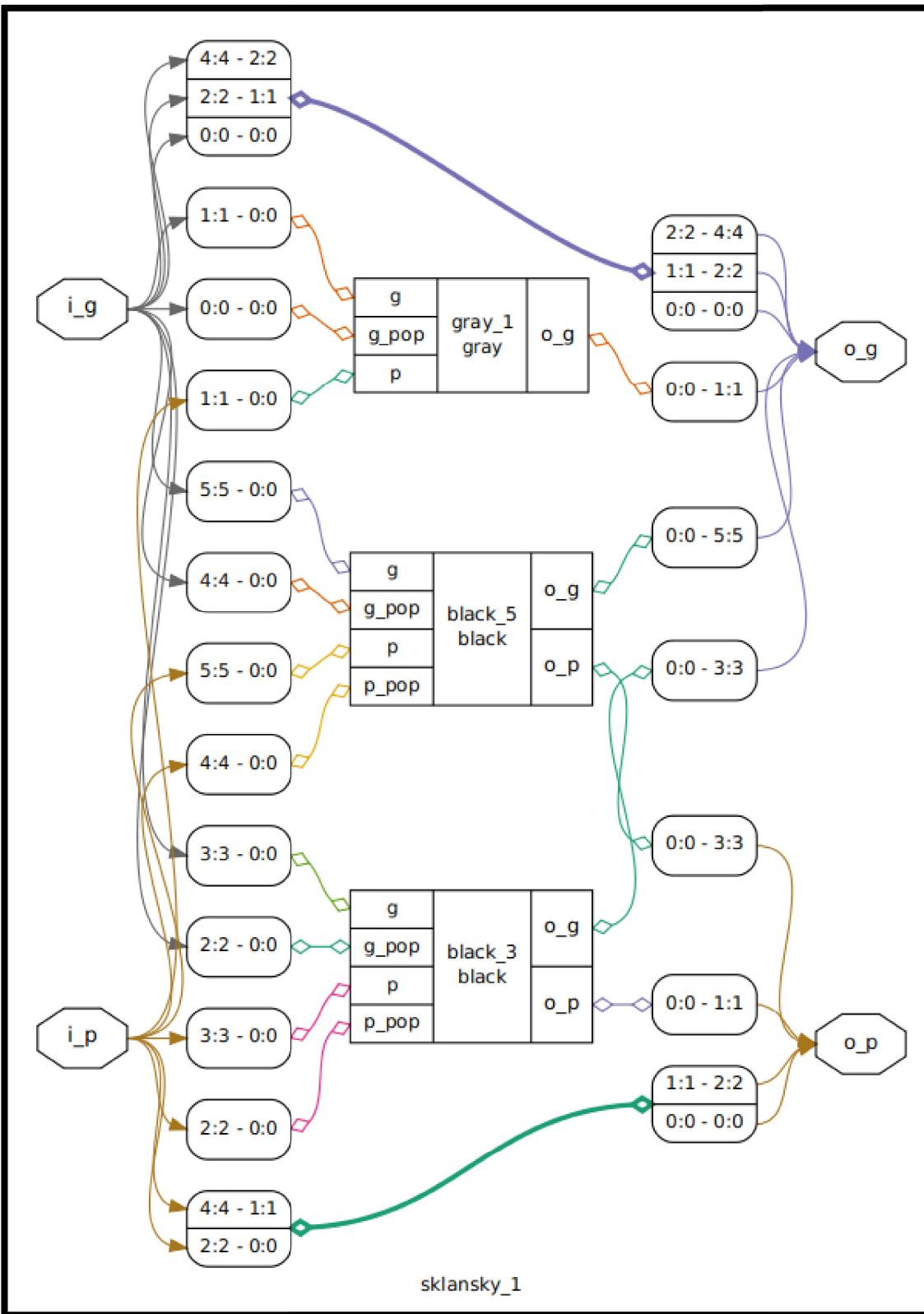
Rysunek 27: Moduł black - synteza logiczna

7.4. Moduł sklansky_0



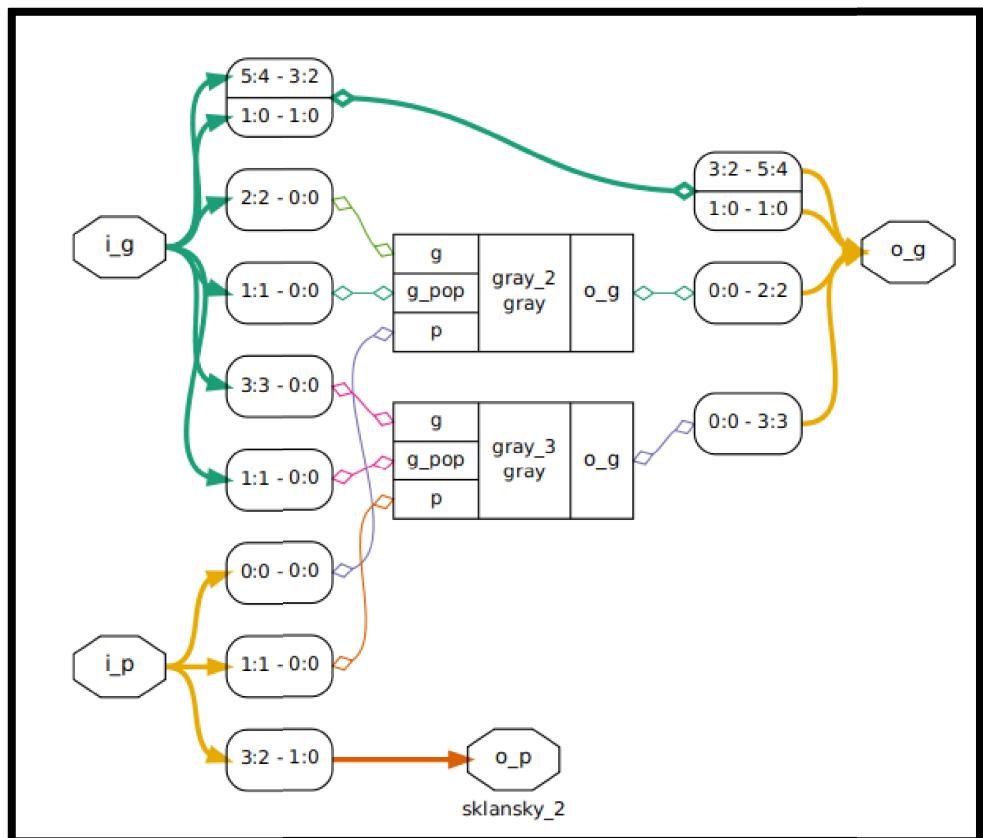
Rysunek 28: Moduł `sklansky_0` - synteza logiczna

7.5. Moduł sklansky_1



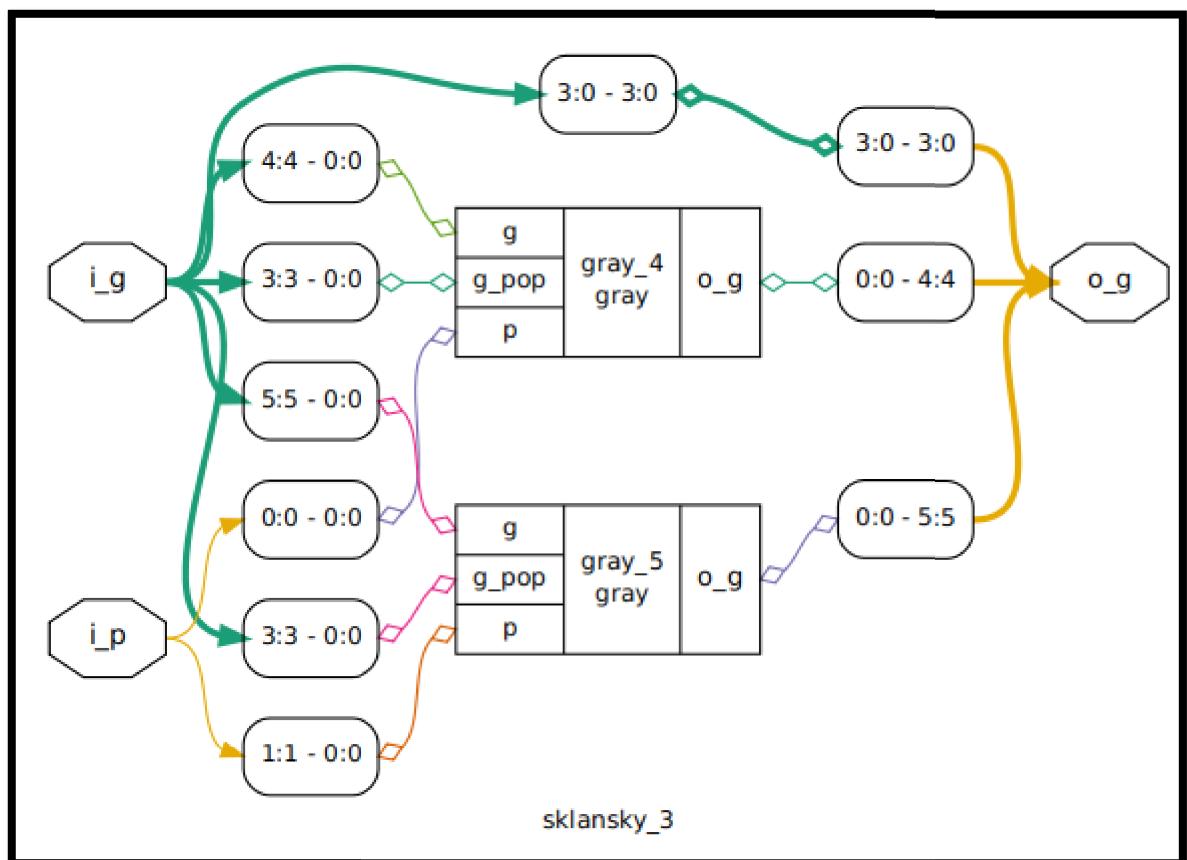
Rysunek 29: Moduł sklansky_1 - synteza logiczna

7.6.Moduł sklansky_2



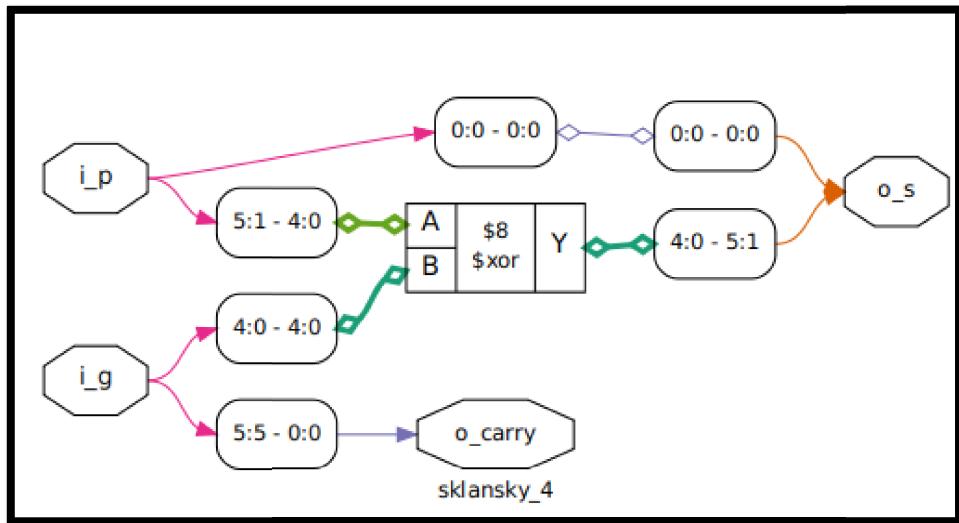
Rysunek 30: Moduł sklansky_2 - synteza logiczna

7.7.Moduł sklansky_3



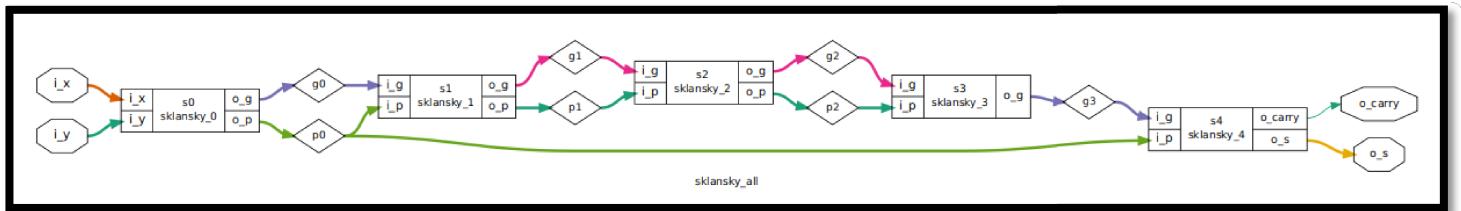
Rysunek 31: Moduł sklansky_3 - synteza logiczna

7.8.Moduł sklansky_4



Rysunek 32: Moduł sklansky_4 - synteza logiczna

7.9.Moduł sklansky_all



Rysunek 33: Moduł sklansky_all - synteza logiczna

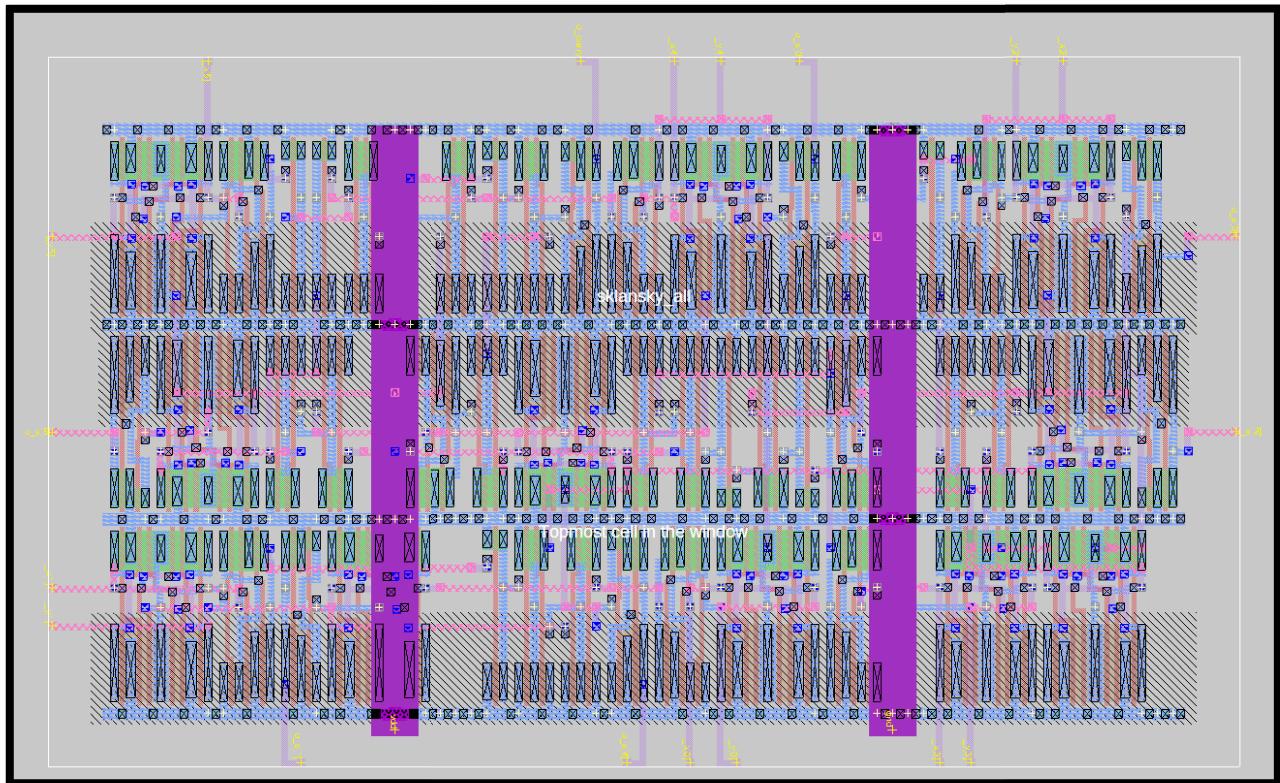
7.10. Statystyki

Tabela 2: Statystyki dla syntezy logicznej

Number of wires	40
Number of wire bits	60
Number of public wires	40
Number of public wire bits	60
Number of memories	0
Number of memory bits	0
Number of processes	0
Number of cells	47
AND2X2	8
BUFX2	7
INVX1	7
NAND2X1	14
XOR2X1	11

8. Synteza fizyczna układu - Qflow

8.1. Model fizyczny sumatora



Rysunek 34: Model fizyczny sumatora

8.2. Opóźnienie na ścieżkach - 3 ścieżki z maksymalnym opóźnieniem

- Path input pin $i_x[1]$ to output pin $o_s[5]$ delay 991.358 ps

0.0 ps	$i_x[1]:$	\rightarrow	$_26_B$
163.3 ps	$\backslash s0.gp_1.p:$	$_26_Y$	\rightarrow
250.8 ps	$_7:$	$_44_Y$	\rightarrow
387.5 ps	$\backslash s1.gray_1.o_g:$	$_45_Y$	\rightarrow
484.2 ps	$_11:$	$_50_Y$	\rightarrow
622.1 ps	$\backslash s2.gray_3.o_g:$	$_51_Y$	\rightarrow
718.9 ps	$_13:$	$_53_Y$	\rightarrow
814.2 ps	$\backslash s3.gray_4.o_g:$	$_54_Y$	\rightarrow
910.5 ps	$_1[5]:$	$_62_Y$	\rightarrow
991.4 ps	$o_s[5]:$	$_22_Y$	\rightarrow
			$o_s[5]$

- Path input pin i_x[1] to output pin o_s[4] delay 806.688 ps

0.0 ps	i_x[1]:	→	_26_/B
163.3 ps	\s0(gp_1.p: _26_/Y	→	_44_/B
250.8 ps	_7_: _44_/Y	→	_45_/B
387.5 ps	\s1.gray_1.o_g: _45_/Y	→	_50_/A
484.2 ps	_11_: _50_/Y	→	_51_/B
622.1 ps	\s2.gray_3.o_g: _51_/Y	→	_61_/A
725.8 ps	_1_[4]: _61_/Y	→	_21_/A
806.7 ps	o_s[4]: _21_/Y	→	o_s[4]

- Path input pin i_x[1] to output pin o_s[3] delay 756.632 ps

0.0 ps	i_x[1]:	→	_26_/B
163.3 ps	\s0(gp_1.p: _26_/Y	→	_44_/B
250.8 ps	_7_: _44_/Y	→	_45_/B
387.5 ps	\s1.gray_1.o_g: _45_/Y	→	_47_/A
484.2 ps	_9_: _47_/Y	→	_48_/B
579.5 ps	\s2.gray_2.o_g: _48_/Y	→	_60_/A
675.8 ps	_1_[3]: _60_/Y	→	_20_/A
756.6 ps	o_s[3]: _20_/Y	→	o_s[3]

8.3.Opóźnienie na ścieżkach - 3 ścieżki z minimalnym opóźnieniem

- Path input pin i_y[0] to output pin o_s[0] delay 142.2 ps

0.0 ps	i_y[0]:	→	_24_/A
73.6 ps	\s0(gp_0.p: _24_/Y	→	_17_/A
142.2 ps	o_s[0]: _17_/Y	→	o_s[0]

- Path input pin i_y[0] to output pin o_s[1] delay 258.905 ps

0.0 ps	i_y[0]:	→	_23_/A
103.4ps	\s0(gp_0.g:	→	_58_/A
190.3ps	_1_[1]:	→	_18_/A
258.9 ps	o_s[1]:	→	o_s[1]

- Path input pin i_y[4] to output pin o_s[4] delay 312.397 ps

0.0 ps	i_y[4]:	→	_32_/A
132.9ps	\s0(gp_4.p:	→	_61_/B
244.1ps	_1_[4]:	→	_21_/A
312.4 ps	o_s[4]:	→	o_s[4]

8.4.Statystyki

Tabela 3: Statystyki dla syntezy fizycznej

Total stdcells	47
Total cell width	1.49e+04
Total cell height	4.70e+04
Total cell area	1.49e+07
Total core area	1.49e+07
Average cell height	1.00e+03

9. Wnioski

W pierwszym etapie przyswoiłyśmy wiedzę na temat Veriloga, Yosys oraz Qflow. Dzięki takim programom możemy w prosty sposób zaprojektować układ cyfrowy oraz zasymulować jego działanie. Modelowanie systemów elektronicznych na poziomie abstrakcji jest istotne, gdyż sam model pozwala nam przetestować działanie układu jeszcze przed syntezą, zatem oszczędzamy czas na późniejszym wyszukiwaniu błędów. Yosys oraz Qflow udostępniają nam zestawy narzędzi, dzięki którym możemy stworzyć wybrany przez siebie układ cyfrowy. Istnieje wiele różnych rodzajów sumatorów, każdy z nich możemy oceniać według wybranych kryteriów m.in. liczba ogniw GP, obciążenie czy liczba poziomów oraz opóźnienie przeniesienia. Gdy mamy do czynienia z sumatorami, zastanawiamy się, w jaki sposób skrócić czas wykonywanych działań. Aby przyspieszyć dodawanie dwuargumentowe można wykorzystać sumatory, które składają sumy tymczasowe poprzez składanie sum warunkowych (COSA - conditional sum adder), funkcję przełączania sum częściowych (CSA - carry-select adder), składanie sum korygowanych (CIA - carry-increment adder) oraz obliczanie i korekcję sum tymczasowych (ELM). W przypadku, gdy skracamy czas propagacji przeniesień możemy wykorzystać antycypację przeniesień (CLA - carry look-ahead adder), skracanie ścieżki propagacji przeniesienia (CSKA - carry skip adder) lub wytwarzanie przeniesień równoległych (PPA - parallel prefix adder). Złożoność i szybkość sumatorów można opisać za pomocą charakterystyk AT. Porównanie sumatorów znajdziemy w tabelce poniżej:

Tabela 4: Charakterystyki AT

Rodzaj sumatora	A	T	Opóźnienie przeniesienia	Liczba poziomów
Pełny 1-bitowy FA	7	4	2	5 bramek
RCA	$7n$	$2n$	$2n$	nxF
CLA	$\sim 7n$	$4\log n$	$4\log n$	$\log n$ bloków
COSA	$3n\log n$	$2 + 2\log n$	$2\log n$	$2xRCA$, $\log n$ poziomów MPX
CSKA	$\sim 8n$	$4\sqrt{n}$	$4\sqrt{n}$	$nxF + 2\sqrt{n}$ MPX, $2\sqrt{n}$ bloków
CSLA	$\sim 14n$	$2\sqrt{2n}$	$2\sqrt{2n}$	$2xRCA$, $\sqrt{2n}$ bloków
PPA	$\sim 5n + 3n\log\sqrt{n}$	$3 + 2\log n$	$2\log n$	$\log n$ poziomów GP

Sumator prefiksowy ma bardziej złożoną strukturę, ale za to działa szybciej. Można zauważyc, że opóźnienie przeniesienia jest krótsze niż T. Rodzajów sumatorów PPA jest kilka, każdy działa na trochę innych zasadach. Przeanalizowałyśmy 6 typów, porównałyśmy je, a następnie wybrałyśmy architekturę typu Sklansky ze względu na rekurencyjne obliczanie prefiksów, ale również regularność. W związku z tym, że sumatory prefiksowe składają się z 3 części, czyli komórki czarnej, szarej oraz bufora, stworzyłyśmy model sumatora 6-bitowego, który składa się z 2 czarnych komórek, 5 szarych oraz 6 buforów, a następnie przystąpiłyśmy do tworzenia modelu i syntezy tego sumatora za pomocą kilku stworzonych modułów. Sumator 6-bitowy dodaje do siebie liczby z zakresu 0-63, zatem wyniki będą z zakresu 0-126. Aby sprawdzić poprawność działania sumatora, należy w pętli wykonać wszystkie działania dla każdej pary liczb. Testy wyszły poprawnie, a po syntezie fizycznej miałyśmy okazję sprawdzić szybkość działania sumatora poprzez analizę ścieżek z minimalnym i maksymalnym opóźnieniem. W dokumentacji

zawarłyśmy wstęp teoretyczny, który był efektem analizy literatury na temat programów i sumatorów oraz opis realizacji projektu, gdzie przedstawiłyśmy zaprojektowany i stworzony przez nas sumator. Testy wykazały, że działa on poprawnie. Zatem na końcu po całej syntezie zdecydowałyśmy się również porównać sumatory prefiksowe do tych innego rodzaju. Projekt pozwolił nam rozwinąć naszą wiedzę na temat sumatorów prefiksowych, pogłębił zakres wiadomości na ten temat, ale co najważniejsze, zaciekawił sposobem tworzenia układów cyfrowych i samą architekturą komputerów.

10. Źródła i bibliografia

- <http://www.clifford.at/yosys/>
- <https://github.com/YosysHQ/yosys>
- <http://www.clifford.at/intersynth/>
- http://www.clifford.at/yosys/files/yosys_manual.pdf
- https://en.wikipedia.org/wiki/Logic_synthesis
- <https://symbiosys.readthedocs.io/en/latest/quickstart.html>
- <https://pl.wikipedia.org/wiki/Verilog>
- <https://en.wikipedia.org/wiki/Verilog>
- http://staff.uz.zgora.pl/rwisniew/instrukcje/inne/verilog/verilog_kurs.pdf
- <http://opencircuitdesign.com/qflow/>
- https://inf.lucc.pl/architektura_komputerow_1/2009 - 7_szybkie_sumatory.pdf
- [https://pl.wikipedia.org/wiki/Sumator_\(uk%C5%82ad_logiczny\)](https://pl.wikipedia.org/wiki/Sumator_(uk%C5%82ad_logiczny))
- <http://zak.ict.pwr.wroc.pl/materials/architektura/wyklad%20AK1/AK1-7-18%20Szybkie%20sumatory.pdf>
- <http://zak.ict.pwr.wroc.pl/materials/architektura/wyklad%20AK1/AK1-8-18-Sumatory%20CSA%20i%20multyplikatory.pdf>
- <http://zak.ict.pwr.wroc.pl/materials/Arytmetyka%20komputerow/sumatory%20wieloargumentowe.pdf>
- https://inf.lucc.pl/architektura_komputerow_1/sumatory.pdf
- https://pl.qwe.wiki/wiki/Kogge%E2%80%93Stone_adder
- https://en.wikipedia.org/wiki/Kogge%E2%80%93Stone_adder
- https://users.encs.concordia.ca/~asim/COEN_6501/Lecture_Notes/Parallel%20prefix%20adders%20presentation.pdf
- <https://vdocuments.mx/a-new-parallel-prefix-adder-structure-with-efficient-of-a-32-bit-brent-kung.html>
- <https://www.slideshare.net/IOSR/a0610106-46027342>
- <https://imanagerpublications.com/index.php/home/articleHtml/3764/5>
- <https://www.ijrte.org/wp-content/uploads/papers/v7i5s4/E10920275S419.pdf>
- http://zto.ita.pwr.wroc.pl/~luban/uklady_kom/sum/sum.html
- [https://en.wikipedia.org/wiki/Adder_\(electronics\)](https://en.wikipedia.org/wiki/Adder_(electronics))
- https://en.wikipedia.org/wiki/Carry-lookahead_adder
- https://inf.lucc.pl/architektura_komputerow_1/2009 - 7_szybkie_sumatory.pdf?fbclid=IwAR3TMJsW6a1GrFZPqC6TYqVUysjVnwL7ehmx2BktRLfNEoCj8NdKaAOqII

11. Spis rysunków

RYSUNEK 1: PEŁNY SUMATOR JEDNOBITOWY.....	6
RYSUNEK 2: SCHEMAT 4-BITOWEGO SUMATORA.....	7
RYSUNEK 3: SCHEMATY ELEMENTÓW SUMATORA PREFIKSOWEGO	8
RYSUNEK 4: SCHEMAT DODAWANIA	8
RYSUNEK 5: SCHEMAT PROPAGACJI I GENEROWANIA PRZENIESIEŃ	9
RYSUNEK 6: BLOK GP	10
RYSUNEK 7: SUMATOR KOGGE - STONE.....	11
RYSUNEK 8: SUMATOR BRENT - KUNG	12
RYSUNEK 9: SUMATOR LADNER - FISHER	12
RYSUNEK 10: SUMATOR HAN - CARLSON	13
RYSUNEK 11: SUMATOR KNOWLES	13
RYSUNEK 12: SUMATOR SKLANSKY	14
RYSUNEK 13: 6-BITOWY SUMATOR SKLANSKY'EGO.....	15
RYSUNEK 14: MODUŁ GP	15
RYSUNEK 15: MODUŁ BLACK	16
RYSUNEK 16: MODUŁ GRAY.....	16
RYSUNEK 17: MODUŁ SKLANKSY_0	16
RYSUNEK 18: MODUŁ SKLANKSY_1	17
RYSUNEK 19: MODUŁ SKLANKSY_2	17
RYSUNEK 20: MODUŁ SKLANKSY_3	17
RYSUNEK 21: MODUŁ SKLANKSY_4	18
RYSUNEK 22: MODUŁ SKLANSKY_ALL	18
RYSUNEK 23: TEST WYCZERPUJĄCY	19
RYSUNEK 24: WYNIKI TESTU	19
RYSUNEK 25: MODUŁ GP - SYNTEZA LOGICZNA	20
RYSUNEK 26: MODUŁ GRAY - SYNTEZA LOGICZNA	20
RYSUNEK 27: MODUŁ BLACK - SYNTEZA LOGICZNA	20
RYSUNEK 28: MODUŁ SKLANSKY_0 - SYNTEZA LOGICZNA.....	21
RYSUNEK 29: MODUŁ SKLANSKY_1 - SYNTEZA LOGICZNA.....	22
RYSUNEK 30: MODUŁ SKLANSKY_2 - SYNTEZA LOGICZNA.....	23
RYSUNEK 31: MODUŁ SKLANSKY_3 - SYNTEZA LOGICZNA.....	23
RYSUNEK 32: MODUŁ SKLANSKY_4 - SYNTEZA LOGICZNA.....	24
RYSUNEK 33: MODUŁ SKLANSKY_ALL - SYNTEZA LOGICZNA	24
RYSUNEK 34: MODEL FIZYCZNY SUMATORA	25

12. Spis tabel

TABELA 1: PARAMETRY SIECI GP JAKO ELEMENTY SUMATORA PREFIKSOWEGO	14
TABELA 2: STATYSTYKI DLA SYNTEZY LOGICZNEJ.....	24
TABELA 3: STATYSTYKI DLA SYNTEZY FIZYCZNEJ.....	27
TABELA 4: CHARAKTERYSTYKI AT	28