Politechnika Wrocławska

Implementacja fizyczna układów cyfrowych

Architektura komputerów 2

Projekt

Aleksandra Zdrojowa 248896 piątek TP 7:30-9:00
Przemysław Rychter 248820 czwartek TP 17:05-18:45

Wrocław, dnia 28 Maja 2020r.

SPIS TREŚCI

I. Cel

Celem ogólnym projektu jest synteza logiczna i fizyczna 6-bitowego sumatora prefiksowego z wykorzystaniem narzędzi Yosys/Qflow.

II. Wprowadzenie

Sumatory są podstawowymi układami cyfrowymi, za ich pomocą realizowane są inne operacje: odejmowania, mnożenia oraz dzielenia. Wchodzą w skład ALU procesora - jednostki arytmetyczno-logicznej. Są również używane w innych częściach CPU np. w celu obliczenia adresu w pamięci. Najczęściej sumatory są projektowane w celu operacji na liczbach reprezentowanych w kodzie uzupełnień do dwóch (U2). Układy te w podstawowej wersji realizują algorytm dodawania [10].

$$x_i, y_i, s_i \in \{0, 1, ..., \beta - 1\} \quad c_i, c_{i+1} \in \{0, 1\}$$

$$\begin{cases} x_i + y_i + c_i \geq \beta & s_i = x_i + y_i + c_i - \beta \\ & c_{i+1} = 1 \\ x_i + y_i + c_i < \beta & s_i = x_i + y_i + c_i \\ & c_{i+1} = 0 \end{cases}$$

Co jest równoznaczne ze znajdywaniem kolejnych s_i oraz c_{i+1} dla równania:

$$x_i + y_i + c_i = \beta c_{i+1} + s_i$$

W podstawie $\beta = 2$ równaniu odpowiadają funkcje logiczne:

$$s_i = x_i \oplus y_i \oplus c_i = h_i \oplus c_i$$

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

$$g_i = x_i y_i \quad p_i = x_i \oplus y_i \quad \text{lub} \quad x_i + y_i$$

Funkcja g_i określa stan który wymusza przeniesienie wyjściowe $c_{i+1} = 1$ niezależnie od c_i. Funkcja p_i określa propagacje przeniesienia $c_{i+1} = c_i$ natomiast h_i określa półsumę. Używając

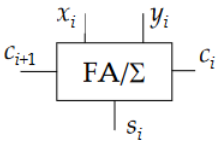

Fig. 1. Full adder - sumator 1-bitowy

układu pełnego sumatora 1-bitowego (Fig. 1) (realizującego wyżej podane funkcje logiczne), możemy zbudować sumatory operujące na "dłuższych" liczbach. Jak widać na rysunku (Fig. 2) kolejne pozycje są ze sobą kaskadowo powiązane (co wynika z algorytmu dodawania). Nie można wytworzyć przeniesienia oraz sumy na danej pozycji bez wyznaczenia przeniesienia z porzedniej pozycji. Sumator zbudowany z połączonych modułów FA nazywamy **RCA** Ripple-Carry Adder, ponieważ przeniesienia są wytwarzane kaskadowo. Gwarantowany czas wykonania dodawania zależy od szybkości wytworzenia przeniesienia na najwyższej pozycji. Propagacja przeniesienia w FA wymaga przejścia przez dwa poziomy logiczne więc opóźnienie w jednym takim układzie wynosi $T = 2$, stąd chcą dodać liczby o n pozycjach czas dodawania wymaga $T_c = 2n$ [11]. Opóźnienia wywołane sekwencyjną propagacją

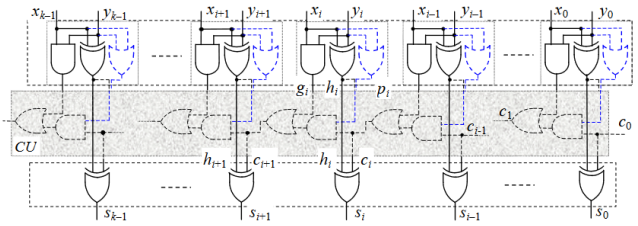

Fig. 2. Sumator kaskadowy RCA [10]

przeniesienień, która tworzy długą scieżkę, mogą być zmiejszone. Wiemy, że kolejne przeniesienia są powiązane rekurencyjnie [11]:

$$c_{i+1} = g_i + p_i c_i$$

$$c_{i+2} = g_{i+1} + p_{i+1} c_{i+1}$$

$$c_{i+2} = g_{i+1} + p_{i+1}(g_i + p_i c_i)$$

$$c_{i+2} = g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i$$

$$c_{i+2} = (g_{i+1} + p_{i+1} g_i) + (p_{i+1} p_i) c_i$$

$$c_{i+2} = G_{i+1:i} + P_{i+1:i} c_i$$

$$c_{i+3} = g_{i+2} + p_{i+2} c_{i+2}$$

$$c_{i+3} = g_{i+2} + p_{i+2}(...)$$

Powyższe wyrażenia zawierą dwa składniki G i P każde powiązane rekurencyjnie. $G_{k:m}$ Opisuje warunki wystarczające do generowania przeniesienia, $P_{k:m}$ Określa propagację przeniesienia z niższej pozycji . Ogólną zależność dwóch konkretnych przeniesień można oznaczyć jako:

$$c_{i+1} = G_{i:j} + P_{i:j} c_j \quad i \geq j$$

Powiązania funkcji G, P i związki przeniesień opisuje się używając operatora Brenta-Kunga (fco) [10].

$$(a, b) \circ (c, d) = (a + bc, bd)$$

Za jego pomocą można przedtawić powiązanie dowolnej części przeniesień

$$(c_{i+1}, ..) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ ... \circ (g_j, p_j) \circ (c_j, ...)$$

Z uwagi na łączność operatora, przeniesienia można wytwarzać różnymi sposobami, w sumatorze RCA są one wytwarzania sekwencyjnie:

$$(c_1, 0) = (g_0, p_0) \circ (c_0, 0)$$

$$(c_2, 0) = (g_1, p_1) \circ (c_1, 0)$$

$$...$$

W celu przyspieszenia wytwarzania przeniesień sumatory **CLA** (Fig. 3) wytwarzają przeniesienia w dwupoziomowym układzie AND/OR (np. dla każdych 4 bitów) jako:

$$(c_1, 0) = (g_0, p_0) \circ (c_0, 0)$$

$$(c_2, 0) = (g_1, p_1) \circ (g_0, p_0) \circ (c_0, 0)$$

$$(c_3, 0) = (g_2, p_2) \circ (g_1, p_1) \circ (g_0, p_0) \circ s(c_0, 0)$$

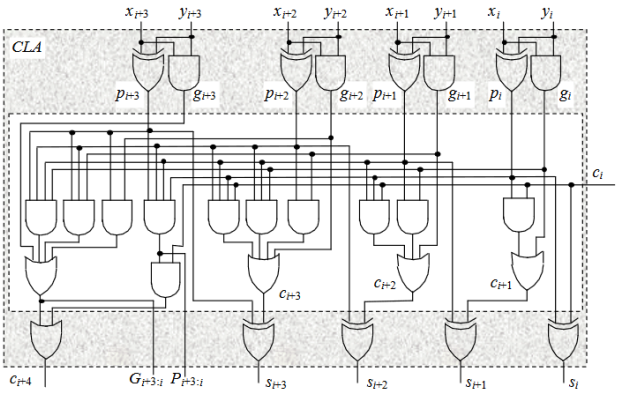

Fig. 3. Czterobitowy sumator CLA z sygnałami G,P dla bloku CLG [10]

.

$$...$$

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_2 c_0$$

$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

Wadą takiego rozwiązania jest duża ilość elementów. Bramki mają także ogarniczoną ilość wejść oraz obciążenie prądowe. Przy użyciu bramek dwu-wejściowych układ CLA, będzie miał opóźnienie $T_c = log_2 2n$ [11]. Struktury CLA można również łączyć blokiem CLG wtedy opóźnienie będzie wynosić $T_c = 4log_b n$ [11], gdzie b to rozmiar (najmiejszej) grupy CLA. Zmieniając strukturę generacji przeniesień dla sumatora CLA na sieć prefiksową możemy poprawić wydajność utrzymując logarytmiczny wzrost opóźnienia. Sumatory prefiksowe **PPA** wykorzystują nie tylko rekurencyjną zależność przeniesień, ale także rekurencyjne skojarzenie funkcji G i P [10].

$$(G_{k:m}, P_{k:m}) = (g_k, p_k) \circ (g_{k-1}, p_{k-1}) \circ ... \circ (g_m, p_m)$$

$$G_{k:m} = g_k + p_k g_{k-1} + p_k p_{k-1} g_{k-2} + ... + p_k ... p_{m+1} g_m$$

$$P_{k:m} = p_k p_{k-1} ... p_m \qquad (k \geq m)$$

Strukturę sumatorów PPA można podzielić na 3 bloki (Fig. 4) [7]

- blok wytwarzania bitowych funkcji półsumy h_i, generacji przeniesienia g_i oraz oraz propagacji przeniesienia p_i, którą opcjonalnie można zastąpić półsumą
- blok GP wywtarzający przeniesienia c_i na podstawie funkcji G i P
- blok wytwarzania sum bitowych s_i

Zrealizowany przez autorów sumator nie będzie uwzględniał przeniesienia wejściowego c_0, więc wyrażenie opisujące sumę uprości się:

$$s_i = h_i \oplus c_i$$

$$(c_i, ..) = (g_{i-1}, p_{i-1}) \circ (g_{i-2}, p_{i-2}) \circ ... \circ (g_0, p_0) \circ (c_0, ..)$$

$$(G_{i:j}, P_{i:j}) = (g_i, p_i) \circ (g_{i-1}, p_{i-1}) \circ ... \circ (g_j, p_j)$$

$$(c_i, ..) = (G_{i-1:0}, P_{i-1:0}) \circ (c_0, ..)$$

$$c_i = G_{i-1:0} + P_{i-1:0} c_0 \quad \text{ale} \quad c_0 = 0 \quad \text{więc}$$

$$c_i = G_{i-1:0} \quad \text{dlatego} \quad s_i = h_i \oplus G_{i-1:0}$$

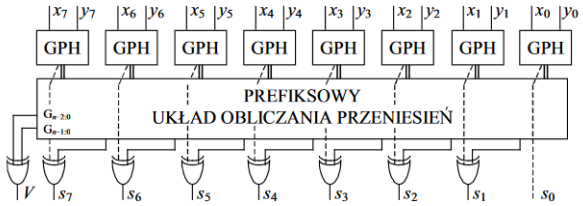

Fig. 4. Ogólna struktura sumatorów prefiksowych. [14]

III. Wybór architektury sumatora PPA

Znanych jest wiele sposobów realizacji struktury prefiskowej do obliczania funkcji G i P w sieci logicznych o głębokości $O(log_2 n)$ poziomów. W naszym projekcie chcielibyśmy zrealizować 6-bitowy sumator BKA - Brent Kung Adder. Richard Peirce Brent i Hsiang Te Kung w 1982 roku zaproponowali układ [7] który wprowadził większą regularność do struktury sumatora, zredukował przeciążenie okablowania co doprowadziło do lepszej wydajności i mniejszej powierzchni w porównaniu do sumatora Kogge-Stone. BKA ma jednak największą głębokość logiczną z sumatorów PP, co implikuje dłuższy czas obliczeń. W celu stworzenia struktury prefiksowej użyliśmy przekształcenia Brenta-Kunga [11].

- poziom 1 - rozłączne funkcje 2-bitowe
  $(i = 0, 1, ..., 2^{-1}n - 1)$, $(G_{2i+1:2i}, P_{2i+1:2i}) =$
  $= (G_{2i+1:2i+1}, P_{2i+1:2i+1}) \circ (G_{2i:2i}, P_{2i:2i})$
- poziom 2 - rozłaczne funkcje 4-bitowe
  $(i = 0, 1, ..., 2^{-2}n - 1)$, $(G_{4i+3:4i}, P_{4i+3:4i}) =$
  $= (G_{4i+3:4i+2}, P_{4i+3:4i+2}) \circ (G_{4i+1:4i}, P_{4i+1:4i})$
- poziom 3 - rozłaczne funkcje 8-bitowe
  $(i = 0, 1, ..., 2^{-3}n - 1)$, $(G_{8i+7:8i}, P_{8i+7:8i}) =$
  $= (G_{8i+7:8i+4}, P_{8i+7:8i+4}) \circ (G_{8i+3:8i}, P_{8i+3:8i})$
- poziom $m = log_2 n$ - funkcję obejmujące 3K i 4K bitow,
  $K - 2^{m-2}$, $(G_{sK-1:0}, P_{sK-1:0}) =$
  $= (G_{sK-1:2K}, P_{sK-1:2K}) \circ (G_{2K-1:0}, P_{2K-1:0})$ $(s = 3, 4)$
- poziom $2m - 1$ - funkcje obejmujące parzystą liczbę bitów dotychczas nie wytworzone
  $(i = 1, 2, ..., \frac{1}{4}n - 1)$, $(G_{4i+1:0}, P_{4i+1:0}) =$
  $= (G_{4i+1:4i}, P_{4i+1:4i}) \circ (G_{4i-1:0}, P_{4i-1:0})$
- poziom $2m - 2$ - pozostałe brakujące funkcje
  $(i = 1, 2, ..., \frac{1}{2}n)$, $(G_{2i:0}, P_{2i:0}) =$
  $= (G_{2i:2i}, P_{2i:2i}) \circ (G_{2i-1:0}, P_{2i-1:0})$

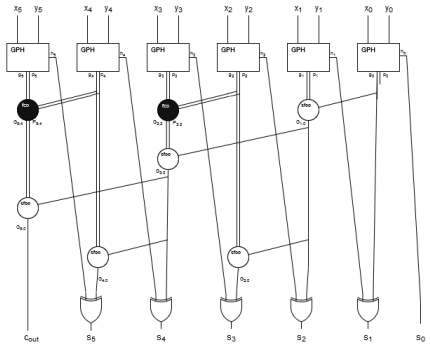

Fig. 5. Schemat 6-bitowego sumatora BKA.

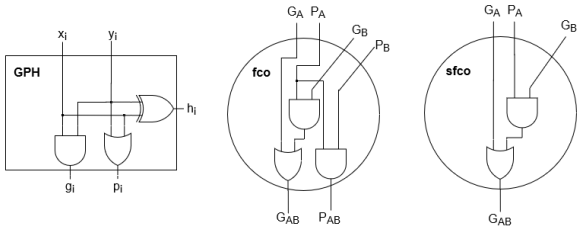

Fig. 6. Schematy elementów konstrukcyjnych.

Układ sześciu prostokątów to poziom obliczający funkcje g, p oraz h na każdej pozycji. Czarna kropka to węzeł konstrukcyjny odpowiadający działaniu operatora FCO (Brenta-Kunga, ○). Ponieważ w sumatorze $c_0 = 0$ to w stopniach wyjściowych zbędna jest funkcja propagacji, dlatego zastał użyty uproszczony węzeł - biała kropka.

IV. Zapis sumatora w HDL

Mianem Hardware description language (HDL) określa się języki opisujące sprzęt, działanie sumatora zostało opisane w języku Verilog. Języki HDL oferują możliwość wyboru poziomu abstrakcji opisywanych układów. W celu dalszej syntezy układu poprzez narzędzia Yosys oraz qFlow sumator został opisany na poziomie RTL (Register Transfer Level), który jest połączeniem poziomu behawioralnego (opisu zachowania) i przepływu danych. Struktura sumatora została podzielona na moduły realizujące zadania poszczególnych elementów konstrukcyjnych - przedstawionych jako prostokąt oraz kropki.

- generacji funkcji g,p,h dla każdej pozycji - gph.v
- realizacji funkcji operatora FCO(○) - fco.v
- uproszczonej realizacji funkcji operatora FCO przekazujący tylko wymuszenie przeniesienia - sfco.v
- 6-bitowego sumatora BKA - adder.v

Pozycje [4] oraz [5] umożliwiły zapoznanie się z językiem Verilog, za pomocą którego opisaliśmy działanie sumatora i w którym zrealizowaliśmy jego testy (symulacje).

V. Implementacja testu wyczerpującego

W 6-bitowym sumatorze istnieje 12 wejść, sygnał przyjmuje tylko dwa stany niski lub wysoki, dlatego całkowita ilość kombinacji sygnałów wejściowych wynosi 2^{12}, w celu udowodnienia poprawności działania sumatora należy przeprowadzić symulację wymuszającą wszystkie możliwe kombinacje wejść i sprawdzającą poprawność wytworzonej sumy i przeniesienia wyjściowego. Testbench - blok wymuszeń 'adder_tb3.v' to moduł rezlizujący test wyczerpujący, Pobiera dane z pliku 'test_vectors.tv' zawierającego wygenerowane wszystkie możliwości wektrów wejściowych oraz spodziewany wektor wyjściowy - sumę oraz przeniesienie (plik wektorów testowych wygenerowano skryptem matlabowym). Następnie tworzy instancę sumatora, w pętli podaje wymuszenia, sprawdza zgodność z oczekiwanym wektorem wyjściowym. Na końcu podaje informację o wyniku testu, ilości błedów oraz pozycji wektorów testowych z pliku 'test_vectors.tv' dla których wystąpiły błędy. W celu przeprowadzenia symulacji zostało użyte narzędzie symulacyjno-syntezujące Icarus Verilog, generujące formę vvp assembly, która jest uruchamiana poprzez komendę vvp. Test przebiegł prawidłowo - dla wszystkich kombinacji sygnałów wejściowych nie znaleziono przypadku w którym układ zwróciłby inne niż spodziewane dane. Test bench wygenerował również plik test.vcd (zawierący informację o zmianach poziomu sygnałów) który może być zwizualizowany narzędziem GTKWave.

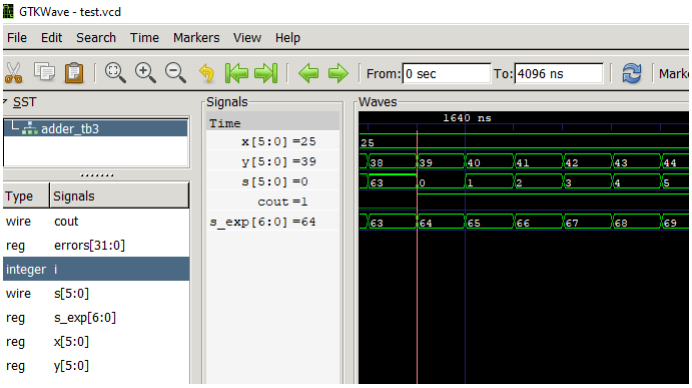

Fig. 7. Fragment sumulacji obrazujący działanie sumatora.

Na rysunku (Fig. 7) widoczny jest fragment testu wyczarpującego, sygnał cout = 1 oznacza wartość 64 ponieważ waga tego sygnału wynosi 2^6.

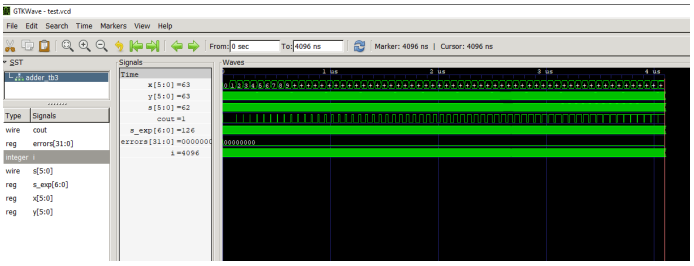

Fig. 8. Zrzut ekranu obrazujący cały przebieg testu.

VI. Synteza Yosys

A. Opis narzędzia

Yosys to pierwsze w pełni funkcjonalne oprogramowanie open-source do obsługi syntezy języka Verilog HDL. Jest szeroko używany jako narzędzię za pomocą którego można wykonywać szereg zadań w dziedzinie syntezy behawioralnej, syntezy RTL oraz syntezy logicznej. Yosys jako narzędzie do syntezy przyjmuje na wejście formalny zapis obwodów w języku HDL (behawioralny opis obwodu) oraz na wyjście generuje opis RTL zawierający bramki logiczne lub fizyczne. Do przeprowadzaonych przez Nas syntez użyliśmy wersji yosys-0.9 oraz programu oraz programu do wizualizacji Graphviz.

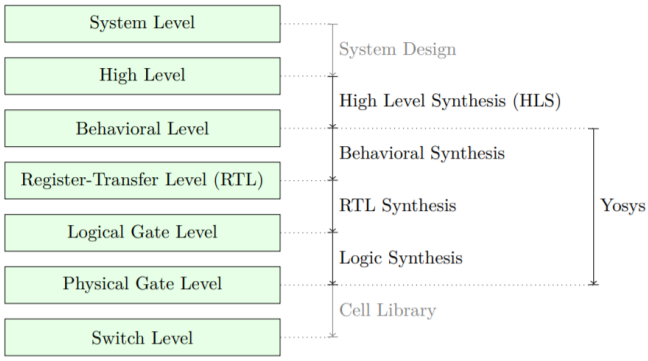

Fig. 9. Różne poziomy abstrakcjii syntezy [1]

B. Synteza RTL

Synteza RTL to synteza niezależna od żadnej biblioteki sposobu opisu obwodu cyfrowego. Oznacza to nic innego jak opis wysokiej abstrakcji nieuzależniony od konkretnej biblioteki technologicznej. Do wykonania tej syntezy stworzyliśmy plik syntezujący 'rtl_netlist.ys', w którym oprócz podstawowych komend syntezujących użyliśmy dodatkowych komend należących do poziomu wysokiej abstrakcji:

- *proc* - zastępuję procesy w projekcie multiplekserami oraz przerzutnikami
- *fsm* - wykonanie ekstrakcji oraz optymlizacji maszyny o skończonym stanie (finite-state-machine)
- *memory* - konwertuję pamieć na komórki [1]

Po wykoanniu komendy:

```
>yosys rtl_netlist.ys
```

Został wygenerowany diagram w pliku show.dot oraz kod w pliku 'rtl_adder.v'. W następnym kroku za pomocą programu do wizualizacji Graphviz, wygenerowaliśmy pełen obraz syntezy. Zgodnie z założeniem kod wygenerowany po tej syntezie niezależny jest od żadnych bramek pochodzących z innych bibliotek, dzięki czemu też podczas wykonywania test benchu nie trzeba było załączać dodatkowego pliku z modelami. Wszystkie pliki dotyczące tej syntezy znajdują się pod adresem https://github.com/PrzemekRychter/ak2-proj/tree/master/roboczy/rtl%20synteza

C. Synteza na poziomie bramek

Drugim rodzajem wykonanej przez nas syntezy była synteza na poziomie bramek, której celem było doprowadzenie do struktury płaskiej(bez modułów innych niż bramki). Pierwszą rzeczą potrzebną do wykonania tej syntezy było dołączenie dodatkowej bibliteki *cmos_cells.lib*.Do poprawnego mapowania tej biblioteki do archtektury docelowej zostało dołączone narzędzie ABC. Kluczową komendą użytą w celu spłaszczeni syntezy była instrukcja *flatten*.Spłaszcza ona projekt, zastępując komórki ich implementacją. Aby stworzyć bardziej czytelny obraz dodaliśmy komendę *splitnets*, która dzieli sieci wielobitowe na jednobitowe oraz dzięki opcji *-ports* następuję również podzielenie modułów, które domyślnie dzielone są tylko na sygnały wewnętrzne. Po wykoanniu komendy:

```
>yosys with_flatten_3.ys
```

Został wygenerowany diagram w pliku show.dot oraz kod w pliku 'with_flatten_3.v'. Następnie podobnie jak podczas przeprowadzania poprzedniej syntezy za pomocą programu Graphviz wygenerowaliśmy pełen jej obraz. Struktura pliku wygenerowanego po syntezie prawie w całości oparta jest na strukturze bramek. Podczas przeprowadzania test benchu, wystapiła potrzeba dodania dodatkowych modułów bramek co wykonałam poprzez dodanie ich do wygenerowanego kodu. Po tych krokach test bench został zakończony sukcesem.

VII. Synteza Qflow

A. Opis narzędzia

Narzędzie Qflow to kompletne narzędzie do syntezy obwodów cyfrowych, w których za wejście uznajemy opis obwodu napisanego w języku behawioralnym (Verilog) a za wyjście kompletny układ fizyczny oparty o kod wejściowy. Na jego działanie składa się szereg komponentów działających na zasadzie open source. Aby poprawnie przeprowadzić syntezę należe zainstalować najważniejsze z nich:

- *graywolf* - narzędzie odpowiedzialne za rozmieszczenie komórek w odpowiednim oraz najabrdziej optymalnym miejscu [9]
- *qrouter* - tworzy szczegółową trasę, opisującą dokładnie sposób generowania fizycznego okablowania łączącego piny [9]
- *yosys* - zostałe opisane w poprzedniej sekcji
- *magic* - narzędzie do układania VLSI (Very Large Scale Integration), odpowiedzialne za proces tworzenia układu scalonego [13]

Cały proces syntezy Qflow od roku 2018 można przeprowadzić za pomocą bardzo przystępnego GUI (od wersji 1.1.104), dzięki czemu cały proces jest bardziej przejrzysty oraz prostszy w wykonaniu. Wadą takiego sposobu jest brak dostępu do wszystkich możliwości Qflow, aby móc skorzystać z bardziej zaawansowanych opcji należałoby przeprowadzić syntezę poprzez linię komend. Na potrzeby zadania zdecydowaliśmy o wykonaniu zadania przy pomocy GUI.

B. Wykonanie syntezy

Jako pliki wejściowe do syntezy użyliśmy pliku znajdujące się pod adresem https://github.com/PrzemekRychter/ak2-proj/tree/master/6-bit%20Brent-Kung%20prefix%20adder. Po wywołaniu komendy

```
>qflow gui
```

W folderze z plikami, zostało otworzone okienko GUI, pozwalające na przeprowadzenie syntezy. Cały proces syntezy podzielony jest na podprocesy, których wykonywanie możemy śledzić, a wszelkie informacje na temat przeprowadzonych kroków znajdują się w wytworzonym przez Qflow folderze log. Pierwszym z etapów jest *Preparation*, w którym wybieramy moduł główny (w naszym wypadku plik 'adder.v') oraz standardową biblitoke wyświetlanych komórek, która instalowana jest razem z narzędziem Qflow. Na potrzeby mojej syntezy wybraliśmy technologię OSU18. Podczas pierwszego etapu dokonać jeszcze możemy wyboru czy po każdym kroku synteza ma się zatrzymać, czy kontynuować. Ja dokonałam wyboru o kontynuowaniu automatycznym po każdym poprawnie wykonanym kroku, dzięki czemu po kliknięciu *Run* przy etapie *Preparation*, synteza przebiegła samoistnie aż do samego końca czego wynikiem w okienku GUI był poniższy obraz:

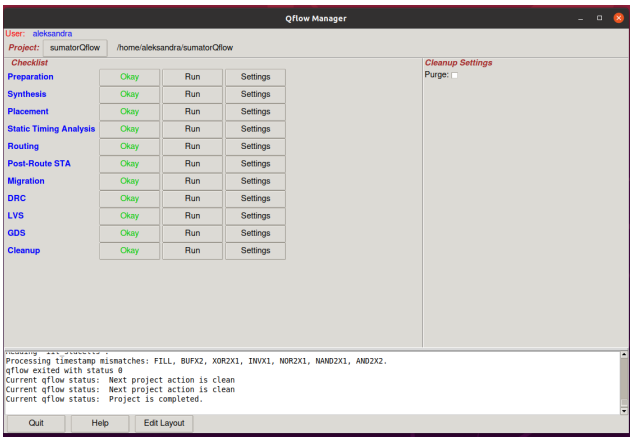

Fig. 10. Zrzut ekranu ze wszystkimi etapami syntezy przeprowadzonymi pomyślnie

Ostateczny wynik syntezy wytworzony przez narzedzię Qflow prezentuję się w następujący sposób:

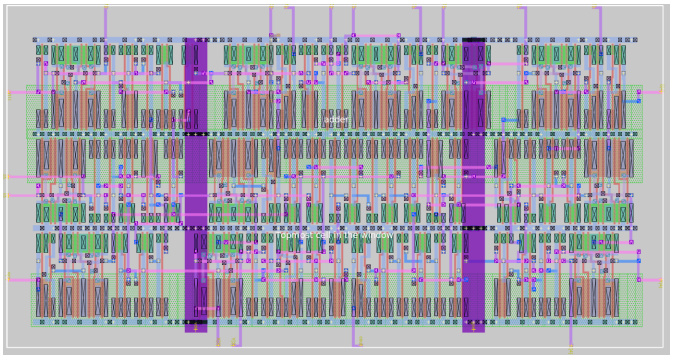

Fig. 11. Ostateczny obraz układu fizycznego

Reszta plików wytworzona automatycznie przez program Qflow podczas naszej syntezy znajduje się pod adresem https://github.com/PrzemekRychter/ak2-proj/tree/master/roboczy/synteza%20qflow

VIII. Wnioski

Projekt wymagał zrozumienia działania sumatorów prefiksowych, oraz narzędzi pozwalających takie układy tworzyć. Po uzupełnieniu wiedzy na temat sumatorów oraz określeniu dokładnych cech układu, który mieliśmy stworzyć, przeszliśmy do implementacji w Verilogu a potem syntezy. W fazie realizacji nie napotkaliśmy większych problemów. Dzięki zapoznaniu się z sumatorami prefiksowymi, zrozumieliśmy podstawową idee przyspieszenia wykonywania operacji arytmetycznych. W projektowaniu układów cyfrowych bardzo ważne są testy, które powinny być skrupulatnie sprawdzane przez więcej niż jedną osobe, oraz powinny konfrontować dane otrzymane z testu z zewnętrznie wygenerowanymi (oczekiwanymi) danymi. Aktualnie projektowanie układów nie wymaga znajomości niskopoziomowej elektroniki. Dzięki istnieniu oprogramowania typu open-source takiego jak Yosys oraz Qflow, które pozwalają intuicyjnie syntezować układy każdemu użytkownikowi. Podczas pracy z narzędziem Qflow, natknęliśmy na problem z błędną wersją qroutera dostępna na stronie *http://opencircuitdesign.com/qflow*, a dokładnie w oficjalnym repozytorium pomysłodawcy na githubie. Po konsultacji mailowej z pomysłodawcą narzędzia, otrzymaliśmy informację że kilka dni wcześniej została umieszczona w repozytorium nie do końca kompatybilna wersja, przez co wykonanie zadania przez pewien czas było niemożliwe. Jest to sytuacja na którą narażeni jesteśmy podczas pracy z narzędziami typu open-source, na szczęście szybka reakcja administratora repozytorium pozwoliła nam na pomyślne ukończenie syntezy. Nasz projekt nie został zrealizowany sprzętowo, dlatego nie mieliśmy okazji dokonania pomiarów czasu wykonywanych operacji. Gdybyśmy zrealizowali projekt w sprzęcie, potrzebny byłby również punkt odniesienia np. w postaci innego sumatora PPA również 6-bitowego, co pozwoliłoby zaobserwować rożnicę w szybkości wykonywania działań z literaturą, która może podawać czasy, ale dla różnych sumatorów(ilości bitów i typów) oraz dla różnych technologii. Pomijając powyższy fakt, można oczekiwać, że nasz układ wykonywałby dodawanie w adekwatnym czasie w stosunku do innych sumatorów 6-bitowych w tej samej technologii, ponieważ jego projekt wykonaliśmy zgodnie z literaturą [11] która odnosi się do [12].

Literatura

[1] Wolf, Clifford. "Yosys manual."(2019). http://www.clifford.at/yosys/files/yosys_manual.pdf

[2] Wolf, Clifford. "Yosys open synthesis suite."(2016). http://www.clifford.at/yosys/files/yosys_presentation.pdf

[3] Wolf, Clifford, Johann Glaser, and Johannes Kepler. "Yosys-a free Verilog synthesis suite."Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip). 2013. http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf

[4] Materiały wykładowe z języka Verilog, WEiTI PW, https://neo.dmcs.pl/jhdl/private/verilog.pdf

[5] Bieganowski, Jacek, and Grzegorz Wawrzyniak. "„Język Verilog w projektowaniu układów FPGA.Źielona Góra (2001). http://staff.uz.zgora.pl/rwisniew/isnstrukcje/inne/verilog/verilog_kurs.pdf

[6] Glaser, Johann, and Clifford Wolf. "Methodology and example-driven interconnect synthesis for designing heterogeneous coarse-grain reconfigurable architectures."Models, Methods, and Tools for Complex Chip Design. Springer, Cham, 2014. 201-221.

[7] Payal, Ravi, Mahima Goel, and Prachi Manglik. "Design and Implementation of Parallel Prefix Adder for Improving the Performance of Carry Lookahead Adder."International Journal of Engineering Research Technology 4.12 (2015): 566-571.

[8] Roy, K. Sripath, et al. "Development of graphical user interface for open source VLSI digital synthesis tool Qflow."International Journal of Engineering Technology 7.1.1 (2018): 710-713.

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

[10] Computer Architecture Group Materials for students - Sumatory równoległe http://zak.ict.pwr.wroc.pl/materials/architektura/wyklad%20AK1/AK1-7-18%20Szybkie%20sumatory.pdf

[11] Biernat, Janusz. Architektura układów arytmetyki resztowej. Akademicka Oficyna Wydawnicza EXIT, 2007.

[12] Brent, Richard P., and Hsiang T. Kung. Ą regular layout for parallel adders."IEEE transactions on Computers 3 (1982): 260-264.

[13] http://opencircuitdesign.com/magic/

[14] http://www.knws.uz.zgora.pl/history/pdf/knws_08_biernat_j_759-762.pdf