POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA

Architektura komputerów 2
Projekt

Implementacja fizyczna układów cyfrowych

AUTORZY:

Adam Jędryka 249443 czwartek TN 18:55 (AK2)
Jakub Pawleniak 248897 środa TP 18:55 (OiAK)

PROWADZĄCY:

Dr inż. Piotr Patronik

14.06.2021

Spis treści

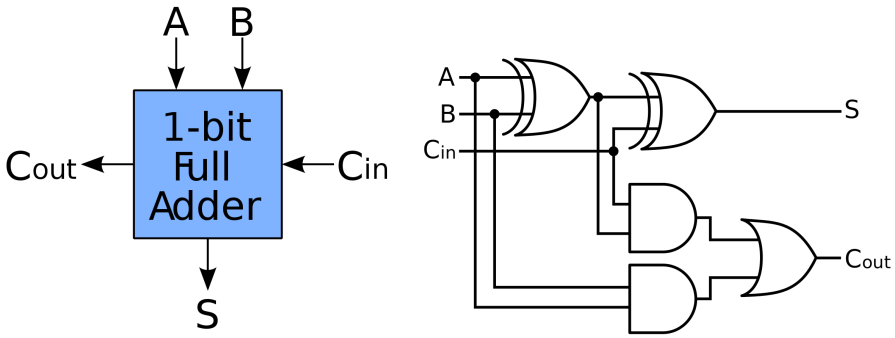
Rozdział 1

Wstęp

1.1. Cel projektu

Celem ogólnym projektu była synteza logiczna i fizyczna modułu full-addera z wykorzystaniem narzędzi **Yosys**[1] oraz **Qflow**[2]. Dodatkowo celem było przeprowadzenie symulacji stworzonego modułu, uwzględniając opóźnienia czasowe na bramkach.

1.2. Układ full-adder'a



(a) Symbol (b) Gate level

Rys. 1.1: Full-adder

Powyższy schemat przedstawia układ full-adder'a, który realizacje dodawanie dwóch liczb dwójkowych. Układ wykorzystuje 2 bramki XOR, 2 bramki AND oraz 1 bramka OR. Działanie układu można przedstawić za pomocą wyrażenia boolowskiego:

$$o\_sum = i\_carry \oplus i\_x \oplus i\_y$$

$$o\_carry = i\_x * i\_y + i\_carry(i\_x \oplus i\_y)$$

Rozdział 2

Verilog

2.1. Moduł full adder'a.

Poniższy moduł zapisany w języku Verilog realizuje działanie układu Full-adder'a zaprezentowanego w poprzednim rozdziale.

Listing 2.1: Plik *full_adder.v*.

```
1  module full_adder
2  (
3    input wire i_x, i_y, i_carry,
4    output wire o_sum, o_carry
5  );
6
7    assign o_sum = i_x ^ i_y ^ i_carry;
8    assign o_carry = i_x & i_y | (i_x ^ i_y) & i_carry;
9
10 endmodule
```

2.2. Test bench

Poniżej zaprezentowany został test bench modułu full adder'a, testujący wszystkie możliwe kombinacje sygnałów wejściowych w odstępach co 20ns.

Listing 2.2: Plik *full_adder_tb.v*.

```
11 `include "full_adder.v"
12 `timescale 1 ns/10 ps  // time-unit = 1 ns, precision = 10 ps
13
14 module full_adder_tb;
15
16   reg x, y, c;
17   wire sum, o_carry;
18
19   localparam period = 20;
20
21   full_adder adder_inst (
22     .i_x(x),
23     .i_y(y),
24     .i_carry(c),
25     .o_sum(sum),
26     .o_carry(o_carry)
27   );
```

```
28
29   initial
30     begin
31       $dumpfile("test_bench.vcd");
32       $dumpvars(1, full_adder_tb);
33       x = 0; y = 0; c = 0; #period;
34       x = 0; y = 0; c = 1; #period;
35       x = 0; y = 1; c = 0; #period;
36       x = 0; y = 1; c = 1; #period;
37       x = 1; y = 0; c = 0; #period;
38       x = 1; y = 0; c = 1; #period;
39       x = 1; y = 1; c = 0; #period;
40       x = 1; y = 1; c = 1; #period;
41     end
42 endmodule
```

Rozdział 3

Synteza Yosys

3.1. Narzędzie

Yosys [1] to narzędzie *open-source* służące do obsługi syntezy języka Verilog HDL. Jest szeroko używany jako narzędzie, za pomocą którego można wykonywać zadania w dziedzinie syntezy behawioralnej, syntezy RTL oraz syntezy logicznej. W projekcie wykorzystana została wersja yosys-0.9.

3.2. Synteza gate level

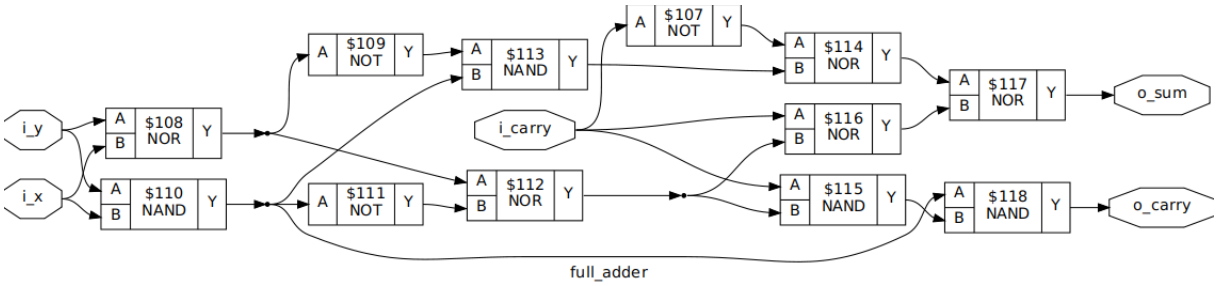
W ramach projektu wykonana została synteza na poziomie bramek przygotowanego modułu full-addera, w wyniku której powinien postać program o strukturze płaskiej (*flatten*), czyli program składający się wyłącznie z bramek logicznych. W celu optymalizacji struktury, do skryptu dodaliśmy zapis wywołujący optymalizację na poziomie wysokim oraz niskim. W celu realizacji tego rodzaju syntezy niezbędne okazało się dołączenie dodatkowej biblioteki **cmos_cells.lib**[3]. W skrypcie znalazła się również komenda odpowiedzialna za podział sieci wielobitowych (*splitnets*). Końcowy fragment skryptu odpowiada za zapis zsyntezowanego modułu, a następnie zapisaniu reprezentacji graficznej logiki układu do pliku .dot.

Listing 3.1: Plik *full_adder.ys*.

```
44 # read design
45 read_verilog ../verilog/full_adder.v
46 hierarchy -check
47
48 synth -flatten
49
50 # high-level synthesis
51 proc; opt; memory; opt; fsm; opt
52
53 # low-level synthesis
54 techmap; opt
55
56 # map to target architecture
57 read_liberty -lib ../libs/cmos_cells.lib
58 dfflibmap -liberty ../libs/cmos_cells.lib
59 abc -liberty ../libs/cmos_cells.lib
60
61 # split larger signals
62 splitnets -ports; opt
63
64 # cleanup
```

```
clean

# write synthesized design
write_verilog ./build/full_adder_synth.v

# show
show -format dot -lib ./build/full_adder_synth.v -prefix ./build/
    ↪ full_adder_cmos
```

Do struktury zsyntezowanego modułu należy zaimportować plik **cmos_cells.v** zawierający moduły bramek użytych w syntezowanym pliku.



Rys. 3.1: Plik *full_adder_cmos.dot*

Aby przeprowadzić powyższy proces, należy użyć jednej z dwóch metod w konsoli:

```
# Use Makefile
make synthG

# or manually

# Synthesis
cd ./yosys && yosys full_adder.ys
# Opening graphics design
@xdot ./build/full_adder_cmos.dot
```

Rozdział 4

Synteza Qflow

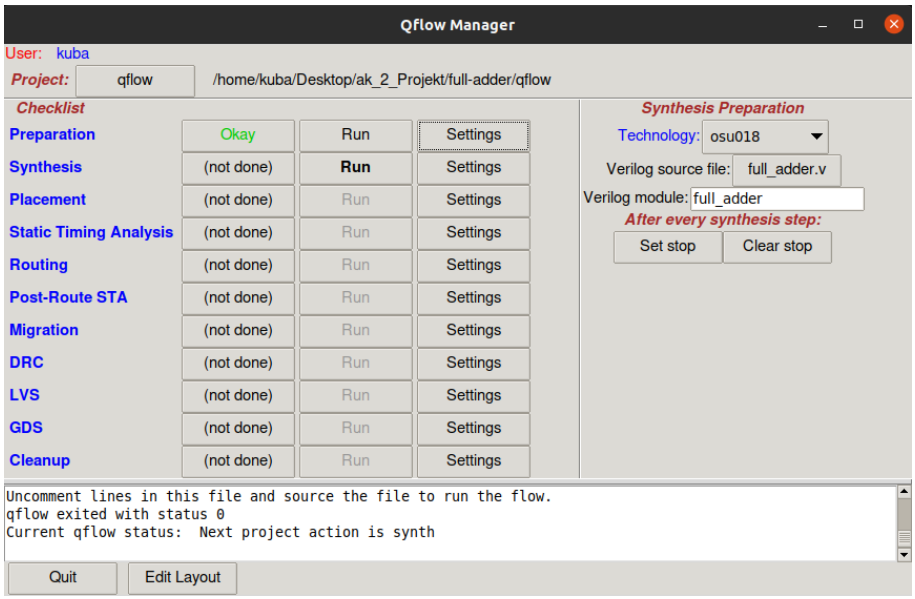
4.1. Narzędzie

Qflow to kompletne narzędzie używane do syntezy obwodów cyfrowych. Na jego działanie składa się szereg komponentów działających na zasadzie *open-source*. Należą do nich m.in.:

- yosys,
- graywolf,
- qrouter,
- magic.

Proces syntezy ułatwia przystępne GUI, dzięki któremu proces jest przejrzysty i prosty w wykonaniu.

W projekcie wykorzystujemy Qflow w wersji 1.3.17.

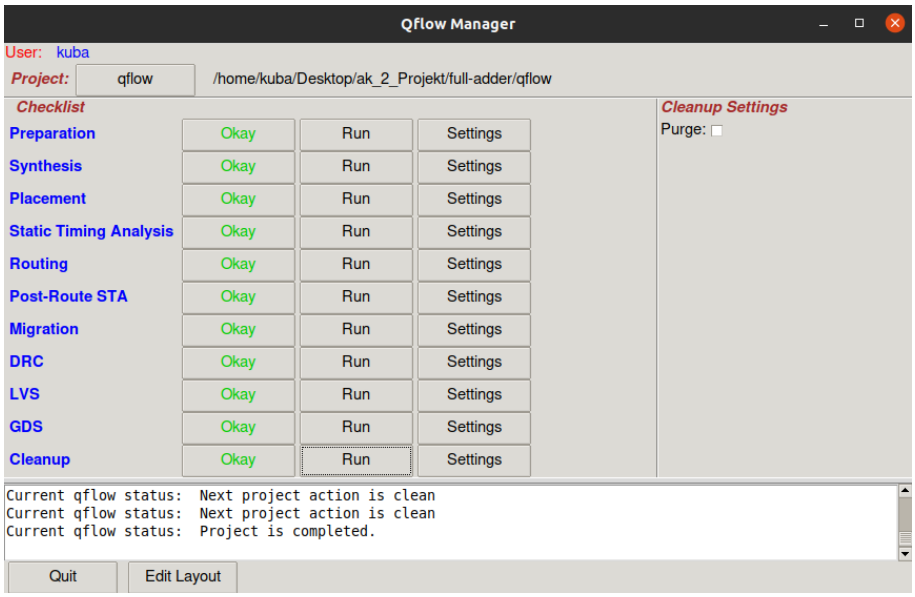


Rys. 4.1: Zrzut ekranu GUI Qflow.

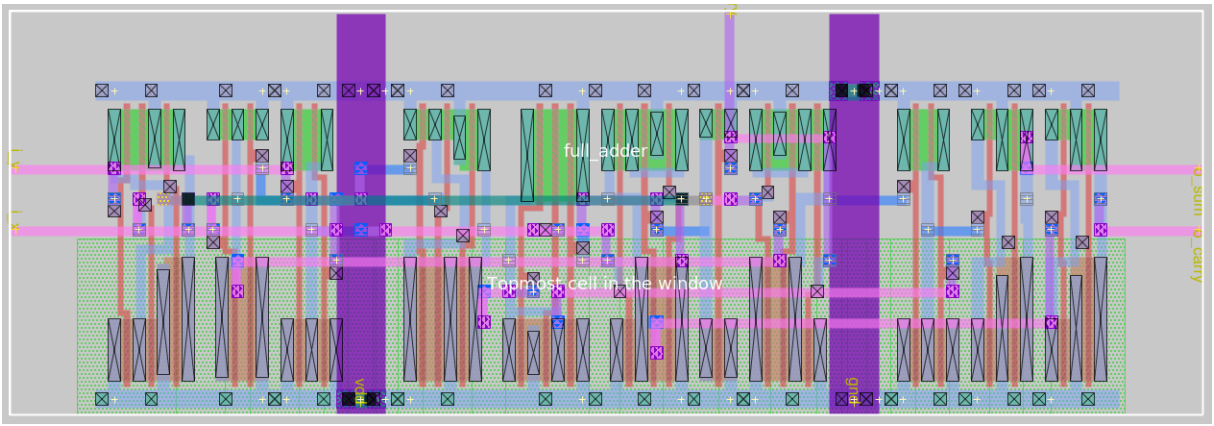
Wadą użycia GUI jest brak dostępu do wszystkich możliwości narzędzia. Zaawansowane opcje dostępne są poprzez linię komend.

4.2. Przeprowadzenie syntezy

Cały proces syntezy podzielony jest na podprocesy, których status możemy śledzić z poziomu GUI, a szczegółowe informacje można odczytać z plików wyjściowych podprocesów utworzonych w folderze projektu. Pierwszym z etapów jest *Preparation*, w którym wybierany jest moduł (w naszym przypadku *full_adder.v*) oraz technologię docelową. Domyślną technologią jest *osu018* i właśnie z niej skorzystaliśmy w procesie syntezy. Domyślnie wszystkie etapy po procesie *Preparation* wykonują się bez zatrzymania (jeżeli nie napotkały po drodze żadnego problemu). Po rozpoczęciu syntezy przyciskiem *Run* wykonuje się etap po etapie.



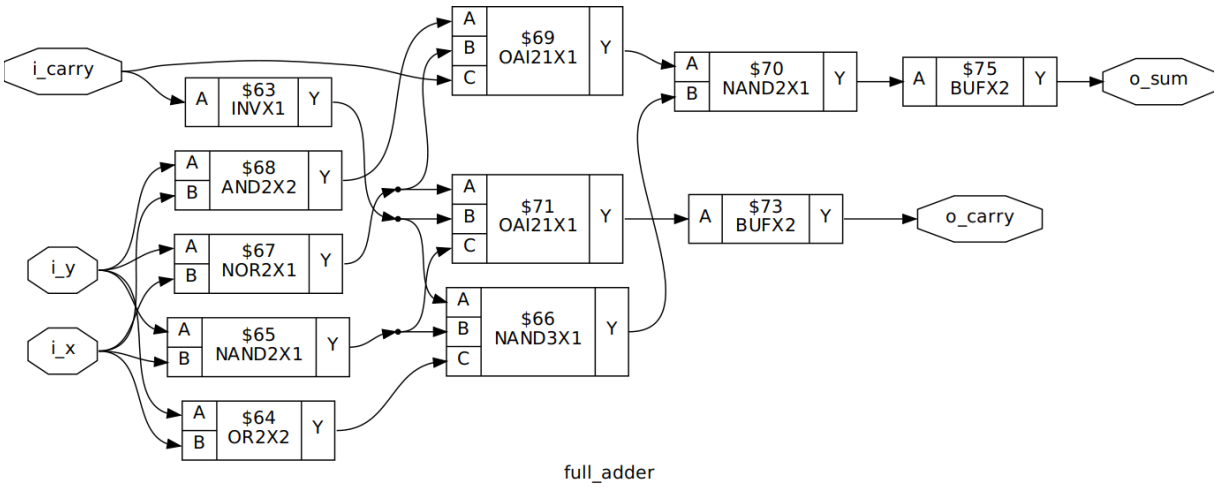
Rys. 4.2: Widok programu Qflow Manager po poprawnie zakończonej syntezie.



Rys. 4.3: Schemat fizyczny układu powstały w wyniku syntezy.

4.3. Reprezentacja bramek w układzie fizycznym

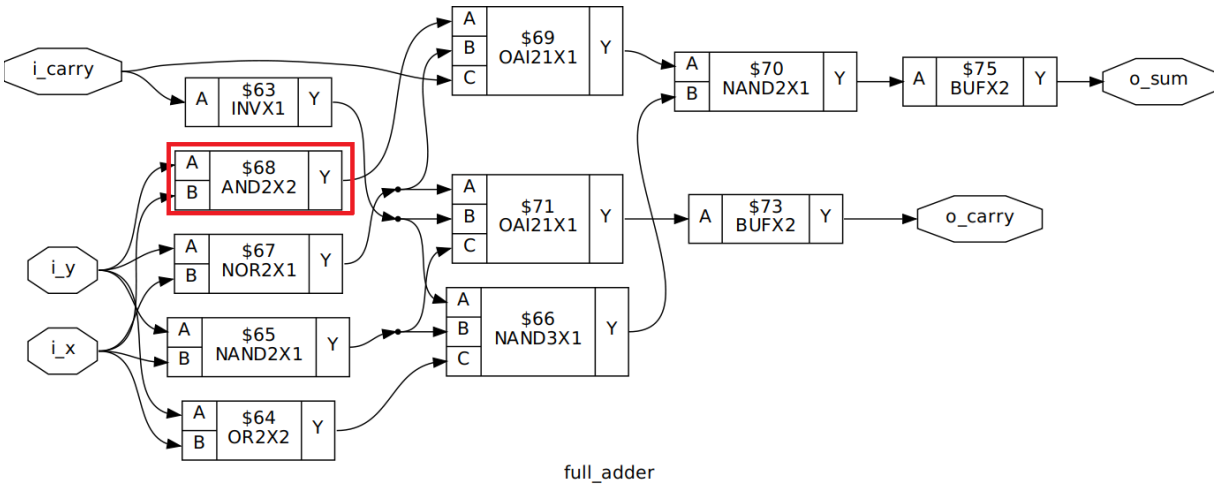
Z racji korzystania z technologi *osu18* [5] w programie Qflow, ponownie wygenerowaliśmy plik schematu logicznego na poziomie bramek korzystając tym razem z domyślnego skryptu yosys wygenerowanego w Qflow dla tej właśnie technologi.



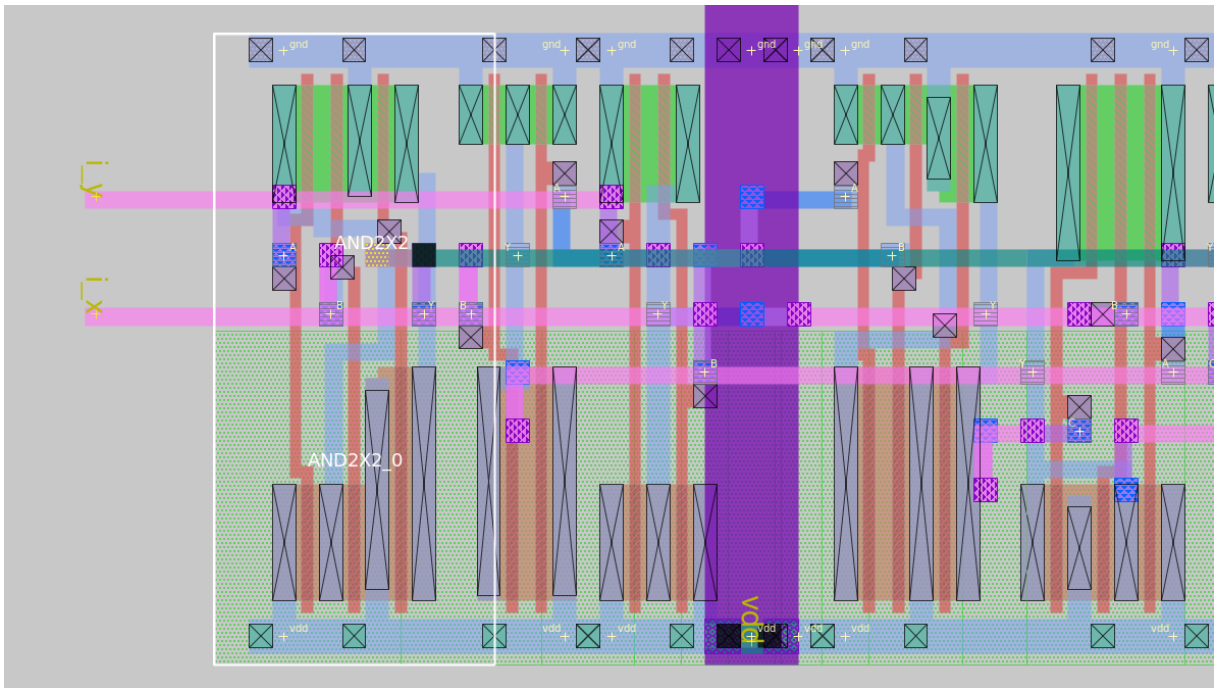
Rys. 4.4: Plik *full_adder_osu18.dot*.

Listing 4.1: Fragment pliku *full_adder.v* po syntezie.

```
/* Verilog module written by vlog2Verilog (qflow) */

module full_adder(
    input i_carry,
    input i_x,
    input i_y,
    output o_carry,
    output o_sum
);

wire vdd = 1'b1;
wire gnd = 1'b0;

wire _7_ ;
wire i_x ;
wire i_y ;
wire _4_ ;
wire _1_ ;
wire o_sum ;
wire i_carry ;
wire _6_ ;
wire _3_ ;
wire _0_ ;
wire o_carry ;
wire _8_ ;
wire _5_ ;
wire _2_ ;

/***/

AND2X2 _14_ (
    .A(i_y),
    .B(i_x),
    .Y(_1_)
);

/***/

endmodule
```



(a) Przykładowa bramka AND



(b) Fragment układu odpowiedzialny za bramkę AND

Rys. 4.5: Implentacja bramki AND.

Jak możemy zauważyć wygenerowany plik schematu logicznego na poziomie bramek odpowiada wygenerowanemu układowi fizycznemu. Na zdjęciu A zaznaczono bramkę AND oraz na zdjęciu B zaznaczono jej odpowiednik w układzie fizycznym.

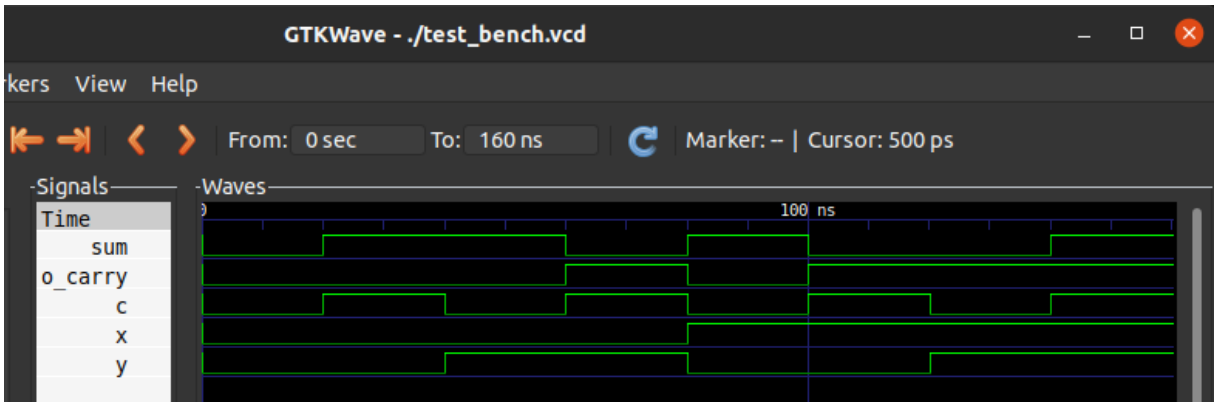
Aby przeprowadzić powyższy proces, należy użyć jednej z dwóch metod w konsoli:

```
# Use Makefile
make qflow

# or manually

# Run Qflow process
cd ./qflow && qflow gui -T osu018 full_adder
```

Rozdział 5

Symulacja cyfrowa

5.1. Symulacja bez opóźnień czasowych



Rys. 5.1: Symulacja modułu full-adder'a bez uwzględnienia opóźnień czasowych.

Input			Output	
x	y	carry	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tab. 5.1: Tabela prawdy dla prawidłowego układu full-adder'a.

Powyższy zrzut ekranu z przebiegów sygnałów w czasie poprawnie pokrywa się z oczekiwaniami zapisanymi w tabeli prawdy dla tego samego układu.

Plik test-bench'a (*full_adder_synth_tb.v*), który testuje moduł full-adder'a po syntezie wymaga, aby dołączyć do niego plik opisujący poszczególne bramki logiczne z pliku technologi osu018: *libs/osu018_stdcells.v*. Jest to spowodowane tym, synteza yosys mapuje na bramki z biblioteki osu018, które następnie używane są w wygenerowanym pliku.

Aby przeprowadzić powyższy proces, należy użyć jednej z dwóch metod w konsoli:

```
# Use Makefile
make synth_sym

# or manually

# Synthesis
@cd ./yosys && yosys full_adder.ys
# Compiling
iverilog -o ./build/dsn.out full_adder_synth_tb.v
# Generate VCD dump
cd ./build && vvp ./dsn.out
#Show waves
gtkwave ./test_bench.vcd --rcvar 'enable_vcd_autosave yes' --rcvar '
    ↪ do_initial_zoom_fit yes'
```

5.2. Symulacja uwzględniająca opóźnienia czasowe

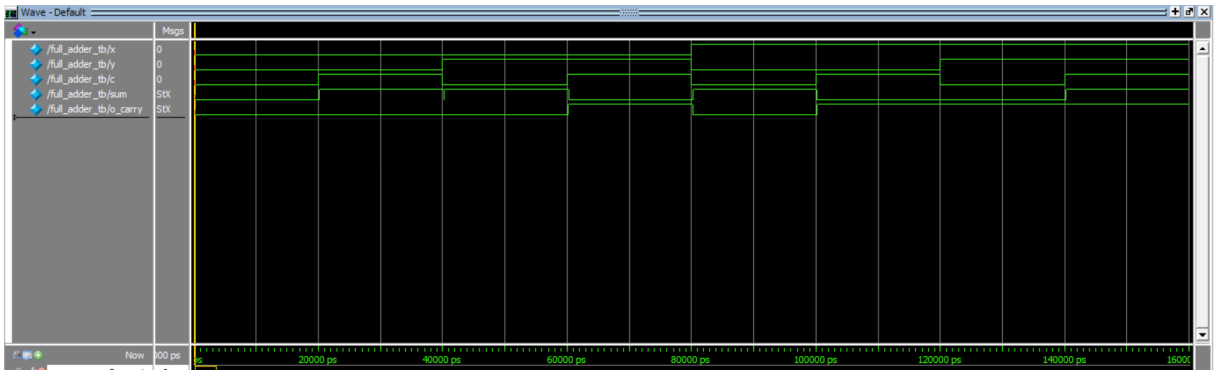
Dla potrzeby wykonania takiej symulacji wymagany jest PDK (process design kit), czyli zestaw plików opisujących modele fizyczne np. bramek wraz z opisami opóźnień na bramkach. Następnie w procesie Post-Route STA należy wygenerować plik .sdf opisujący m.in. opóźnienia na ścieżkach oraz bramkach. Przy pomocy tego plik oraz pliku modułu verilog'owego po syntezie można przeprowadzić testbench'a, który to w efekcie pozwoli nam na zarejestrowanie zmian sygnałów w czasie uwzględniająca opóźnienia czasowe. Następnie będzie w stanie wyświetlić to na wykresie w czasie.

Symulację tą probowaliśmy uzyskać modyfikując procesy programu Qflow. Ręczne użycie komend do poszczególnych procesów mogłoby być bardzo czasochłonne, aby je zrozumieć i poprawnie uruchomić. Jednak po wielu godzin próbach symulacja uwzględniająca opóźnienia czasowe nie została wykonana. Nie byliśmy w stanie wygenerować pliku .sdf z zapisanymi opóźnieniami dla bramek, zapisane tam były tylko opóźnienia dla ścieżek. Oraz program GTKWave[7] z nieznanych nam przyczyn nie pokazywał opóźnień na ścieżkach. Po konsultacjach z prowadzącym polecono nam przeprowadzić tę symulację w programie ModelSim[8]. Program sprawiał nam problemy w instalacji na systemie Linux, dystrybucji Ubuntu, z tego powodu testy przeprowadziliśmy w tym programie na systemie Windows 10.

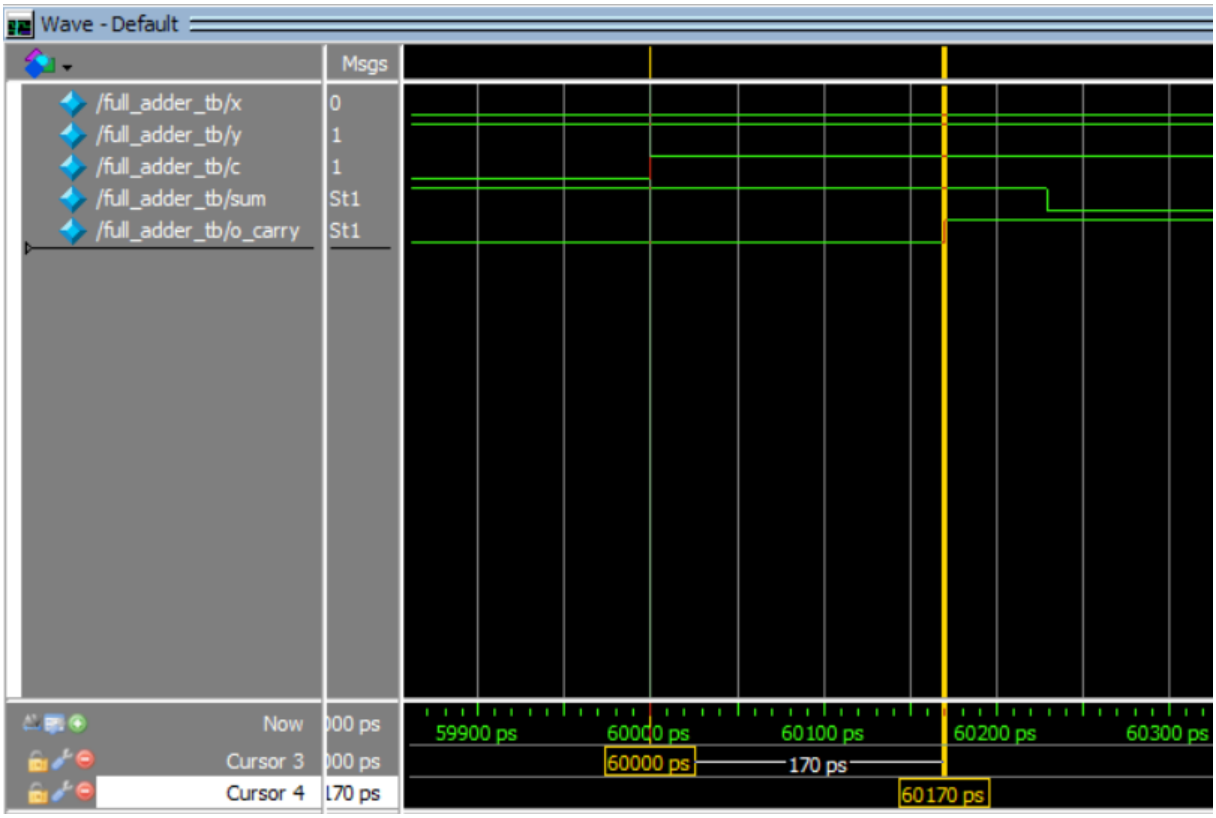
W programie ModelSim należało załączyć pliki modułu po syntezie i test-bench'a wraz z modułami osu018 (*libs/osu018_stdcells.v*). Następnie w ModelSim przeprowadzamy kompilację i otwieramy widok symulacji. Dodajemy wszystkie sygnały do wyświetlenia w przebiegu czasowym oraz uruchamiamy symulację. Symulacja ta obsługuje tylko opóźnienia na bramkach logicznych, które są zawarte w pliku *libs/osu018_stdcells.v*. Nie obsługuje natomiast opóźnień na ścieżkach, gdyż nie byliśmy w stanie załączyć wcześniej wygenerowanego pliku .sdf, gdyż ModelSim wyświetlał mało wymowny komunikat o błędzie: *Fatal: SDF files require Intel FPGA Edition primitive library*. Nie znaleźliśmy żadnej pomocy na ten temat.

W celu uruchomienia symulacji należy znajdując się w folderze *fa-sim* użyć komendy "`vsim -do vsimScript.do`" uruchamiającej poniższy skrypt:

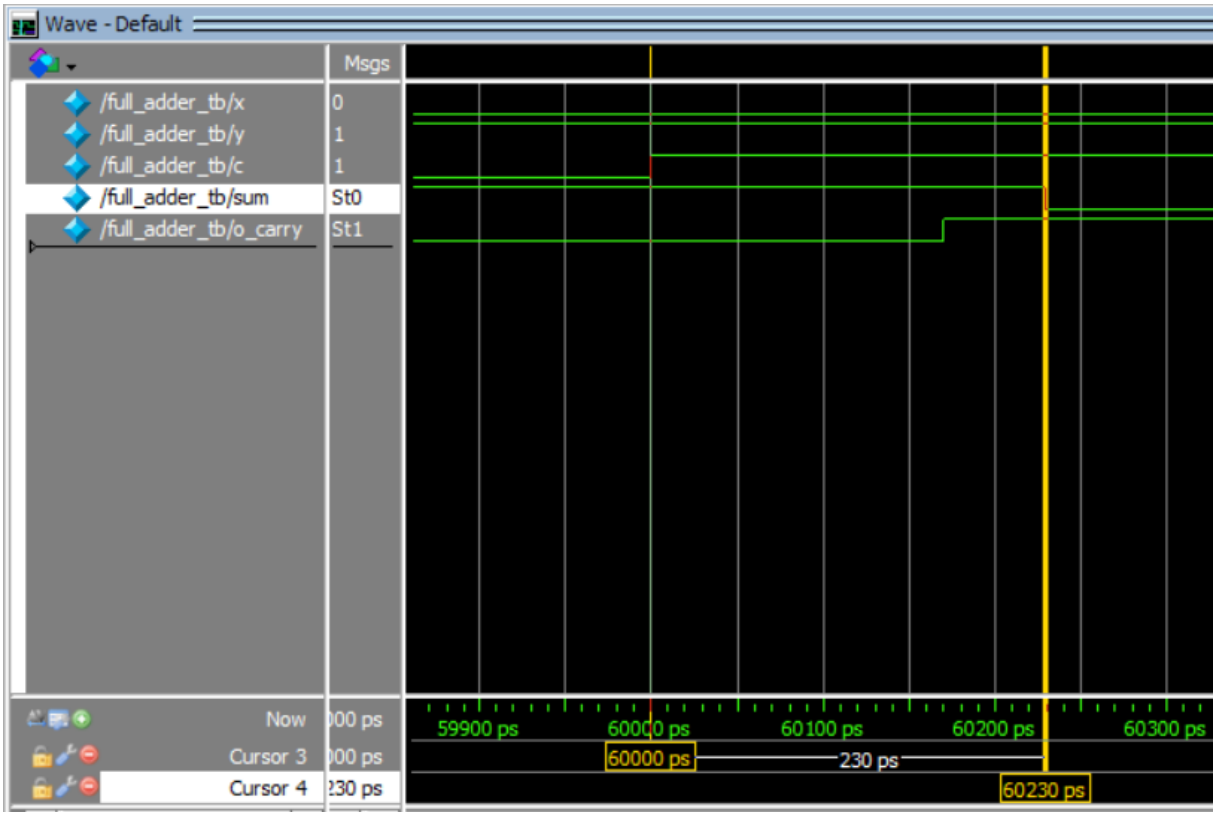
```
1 vsim full_adder.full_adder_tb
2 add wave sim:/full_adder_tb/*
3 run -all
4 wave zoom full
```



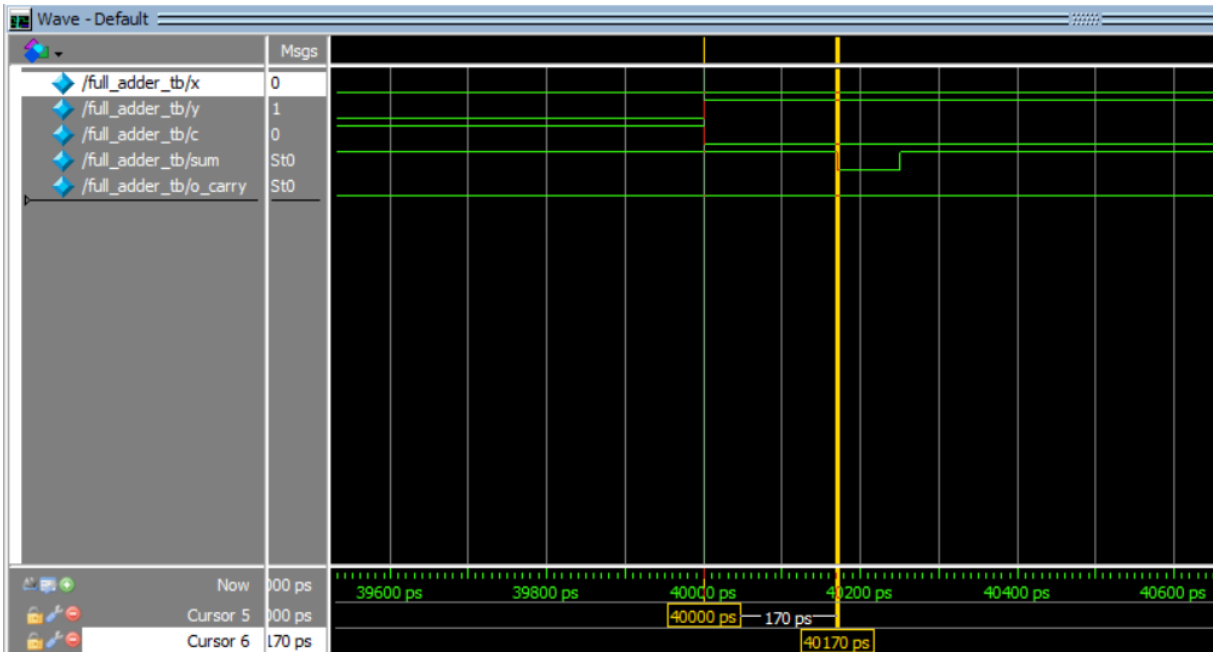
Rys. 5.2: Symulacja modułu full-adder'a z uwzględnieniem opóźnień czasowych na bramkach logicznych.



Rys. 5.3: Symulacja modułu full-adder'a z uwzględnieniem opóźnień czasowych na bramkach logicznych - propagacja wyjścia o_carry.



Rys. 5.4: Symulacja modułu full-adder'a z uwzględnieniem opóźnień czasowych na bramkach logicznych - propagacja wyjścia o_sum.



Rys. 5.5: Symulacja modułu full-adder'a z uwzględnieniem opóźnień czasowych na bramkach logicznych - zbliżenie na opóźnienie 2

Całościowy przebieg możemy zaobserwować na rysunku 5.2.Natomiast kolejne rysunki przedstawiają zbliżenie na zmianę wejść modułu, czas opóźnienia reakcji całego modułu (suma opóźnień bramek) oraz wyjścia modułu full-adder'a.

Na przebiegach z rysunków 5.3 oraz 5.4 możemy zaobserwować, że sygnał *sum* wymaga dłuższego czasu na stabilizację niż sygnał *o_carry*. Jest to spowodowane tym, że do uzyskania ustabilizowanego *sum* sygnały składowe muszą przejść przez większą ilość bramek co w rezultacie daje większą sumę opóźnień niż w przypadku *o_carry*.

Na przebiegu z rysunku 5.5 widzimy zjawisko, w którym sygnał składowy wpłynął na chwilową zmianę sygnału *sum*. Jednak po ustabilizowaniu sygnału sygnał *sum* zmienił swoją wartość na prawidłową.

Odczyt rezultatu na wyjściu z modułu po zmianie wejść, powinien odbyć się okresie propagacji, czyli po maksymalnym opóźnieniu od czasu zmiany wejściowych sygnałów do czasu zmiany wyjściowych sygnałów.

Rozdział 6

Podsumowanie

Przygotowanie prostego modułu w języku Verilog oraz sprawdzenie jego działania przebiegło bezproblemowo. Natomiast podczas procedury syntezy układu zaczęły pojawiać się niedogodności.

Pierwsze problemy pojawiły się podczas instalacji narzędzia Qflow. Najnowsza wersja programu zainstalowana z oficjalnego repozytorium powodowała wiele błędów, natomiast wersja 1.3.17 zainstalowana poprzez standardowe repozytoria package Ubuntu działała poprawnie i była w stanie sama poprawnie skonfigurować swoje środowisko. Jednak w przypadku każdej wersji Qflow wykazywał brak pliku osu018_stdcells.gds2.

Problemem okazała się również biblioteka *OpenSTA* przy użyciu której próbowaliśmy przeprowadzić analizę opóźnień układu. Oficjalna strona wydawcy informuje bowiem, że dokumentacja jest przestarzała. Po wielu godzinach prób nie udało się nam przeprowadzić analizy opóźnień bramek oraz ich połączeń. I na koniec nie jesteśmy w stanie do końca stwierdzić gdzie leży problem.

Symulacje z uwzględnieniem opóźnień zostały wykonane w środowisko ModelSim, które wydaje się dobrym środowiskiem, udostępniającym wiele funkcji poprzez graficzny interfejs.

Ostatecznie można stwierdzić, że społeczność związana z tymi technologiami nie jest zbyt duża, co powoduje, że wsparcie ciężko uzyskać w danym problemie.

Poprzez przygotowanie projektu udało się nam poznać podstawy implementacji fizycznej układów cyfrowych. Dowiedzieliśmy się, z jakich etapów składa się taki proces, jakich narzędzi możemy do tego użyć oraz jakie elementy są nam potrzebne do przeprowadzenia tego procesu.

Cały kod znajduje się pod adresem: `https://github.com/Buglik/AK2-projekt`

Literatura

[1] C. Wolf: *Yosys open synthesis suite* - http://www.clifford.at/yosys/

[2] R. Timothy Edwards: *Qflow 1.3: An Open-Source Digital Synthesis Flow* - http://opencircuitdesign.com/qflow/

[3] C. Wolf: *CMOS Gate-Level Netlist* - https://github.com/YosysHQ/yosys/tree/master/examples/cmos

[4] dr inż. Cezary Maj: *Kurs Verilog* - https://fiona.dmcs.pl/čmaj/Verilog/Kurs%20verilog.pdf

[5] *PKD osu018* - https://download.libresilicon.com/technologies/OSU018/

[6] Wikipedia: *Full-adder* - https://pl.wikipedia.org/wiki/Sumator_(uk%C5%82ad_logiczny)

[7] Program: *GTKWave* - http://gtkwave.sourceforge.net/

[8] Program: *ModelSim* - https://www.intel.pl/content/www/pl/pl/software/programmable/quartus-prime/model-sim.html