

Projektowanie struktury układów FPGA

skrypt do ćwiczeń laboratoryjnych

Mateusz Komorkiewicz,
Tomasz Kryjak

Copyright © 2014
Mateusz Komorkiewicz,
Tomasz Kryjak

PUBLISHED BY AGH

First printing, March 2014

Contents

1	Wprowadzenie	7
1.1	Słowo wstępne	7
1.2	Od lampy elektronowej do układu FPGA - rys historyczny	8
1.3	Budowa układów FPGA serii Spartan 6 firmy Xilinx	8
1.3.1	Blok CLB	8
1.3.2	Pozostałe zasoby	9
2	Układy FPGA — pierwsze kroki	13
2.1	Wstęp	13
2.2	Język opisu sprzętu Verilog	13
2.3	ISE Design Suite — środowisko programistyczne	14
2.3.1	ISE WebPACK	16
2.4	Atlys — platforma sprzętowa	16
2.4.1	Podłączanie i odłączanie kart FPGA Atlys	17
2.5	Zadanie	17
2.6	Zadania do wykonania na laboratorium	17
2.7	Zadania do wykonania w domu	21
2.8	Podsumowanie	21
3	Wstęp do projektowania struktury układów FPGA	23
3.1	Język Verilog - wprowadzenie	23
3.1.1	Moduł	23
3.1.2	Opis połączeń	24
3.1.3	Zapis liczby	25
3.1.4	Łączenie modułów	25

3.1.5	Opis struktury a opis zachowania	26
3.1.6	Bramka AND	27
3.1.7	Bramka OR	27
3.1.8	Bramka NOT	27
3.1.9	Dekoder	27
3.1.10	Koder	28
3.1.11	Demultiplexer	28
3.1.12	Multiplexer	29
3.1.13	Rejestr	29
3.1.14	Liczniak	30
3.1.15	Instrukcja generate	31
3.1.16	Maszyna stanów	32
3.1.17	Moduły arytmetyczne	33
4	Weryfikacja i testowanie projektu	35
4.1	Język Verilog - konstrukcje symulacyjne	36
4.1.1	Środowisko testowe	37
4.1.2	Generacja sekwencji testowych	37
4.1.3	Weryfikacja uzyskanych wyników	39
4.2	Model programowy	40
4.2.1	Dostęp do plików na dysku komputera	41
5	Verilog i weryfikacja – praktyka	43
5.1	Zadania do realizacji na zajęciach	43
5.1.1	Kaskada bramek AND	43
5.1.2	Liczniak dzielący modulo N	45
5.1.3	Złożony moduł logiczny	46
5.2	Zadania do wykonania w domu	46
5.2.1	Linia opóźniająca	46
5.2.2	Tajemniczy moduł	47
6	Maszyny stanowe i zaawansowane testowanie	49
6.1	Zadania do realizacji na laboratorium	49
6.2	Zadania do realizacji w domu	51
6.3	Zadania dodatkowe	51
7	Operacje arytmetyczne	53
7.1	Format zapisu liczb	53
7.1.1	Całkowitoliczbowy bez znaku	53
7.1.2	Całkowitoliczbowy ze znakiem	54
7.1.3	Stałoprzecinkowy bez znaku	55
7.1.4	Stałoprzecinkowy ze znakiem	56

7.2	Zmienna długość słowa	57
7.3	Latencja	57
7.4	Pisanie a generowanie	60
7.5	Pierwiastkowanie, funkcje trygonometryczne, logarytmy	61
7.5.1	Tablicowanie wartości funkcji	61
7.6	Zadania do wykonania na laboratorium	62
7.7	Zadania do wykonania w domu	65
7.8	Zadania dodatkowe	68
8	Potokowe przetwarzanie i analiza obrazów	69
8.1	Wstęp teoretyczny	69
8.2	Typowy cyfrowy interfejs wizyjny	70
8.3	Model programowy przetwarzania obrazów	72
8.4	Uruchomienie toru wizyjnego na karcie Altys	73
8.5	Realizacja operacji LUT	73
8.6	Zadania do wykonania w domu	74
9	Segmentacja obszarów o kolorze skóry	77
9.1	Wprowadzenie	77
9.2	Konwersja RGB do YCbCr — podstawy	78
9.3	Binaryzacja	78
9.4	Filtracja	79
9.5	Wyznaczanie środka ciężkości	79
9.6	Przykład działania	80
9.7	Zadania do wykonania w domu	81
9.7.1	Model programowy	81
10	Konwersja RGB do YCbCr	83
10.1	Model programowy	83
10.2	Implementacja sprzętowa	83
10.3	Uruchomienie na karcie ATLYS	85
10.4	Implementacja programowania	86
10.5	Zadania dodatkowe	86
11	Wyznaczanie środka ciężkości oraz wizualizacja	93
11.1	Wyznaczanie środka ciężkości	93
11.2	Zadania do wykonania na laboratorium	94
11.3	Zadania do wykonania w domu	96

11.4	Zadania dodatkowe	96
12	Potokowa realizacja operacji kontekstowych	99
12.1	Koncepcja realizacji operacji kontekstowych w potokowym systemie wiz-yjnym	99
12.2	Zadania do wykonania na laboratorium	100
12.3	Zadania do wykonania w domu	102
12.4	Zadania dodatkowe	103

1 — Wprowadzenie

1.1 Słowo wstępne

Umiejętność programowania architektur równoległych jest w dzisiejszych czasach bardzo pożądana. Z przyczyn technologicznych maksymalne taktowanie procesorów sekwencyjnych zatrzymało się na ok. 5 GHz, przy czym praktycznie stosuje się rozwiązania o taktowaniu mniejszym od 4 GHz. Zatem postawiono na równoległość. Przykładem są procesory wielordzeniowe (CPU), programowalne karty graficzne (GPU), a także rozwiązania hybrydowe łączące obie architektury (np. procesory serii A firmy AMD). Przy czym jakoś tak dziwnie się składa, że o ile ludzki mózg działa w sposób bardzo równoległy (na razie niedościgły dla systemów technicznych) to myśleć wolimy w sposób sekwencyjny. Programowanie w klasycznych językach C/C++/Java stało się wiedzą powszechną i jest nauczane na wielu różnych szczeblach i kierunkach edukacji. Sekwencyjnie podejście jest bardzo intuicyjne i opisany w ten sposób algorytm łatwo jest zaimplementować. Z programowaniem równoległym sprawa wygląda inaczej. Wymaga ono innego, chyba znacznie mniej oczywistego, sposobu myślenia. Przykładowo na algorytm nie patrzy się jak na "sekwencję instrukcji", ale na "zbiór elementów obliczeniowych". Ponadto, aby wykorzystać możliwości jakie oferuje równoległość należy dość dobrze znać wykorzystywaną platformę sprzętową (np. GPU lub FPGA). W przypadku programowania na CPU nie ma to tak dużego znaczenia, bo dostępne kompilatory tworzą bardzo dobry kod maszynowy w sposób automatyczny.

Wszystko to powoduje, że programowanie równoległe jest trudniejsze od sekwencyjnego. Jednak wydaje się być na chwilę obecną koniecznością. Warto zaznaczyć, że wykształcenie umiejętności programowania równoległego, myślenia w kategoriach równoległości jest niezależne od platformy sprzętowej. Za każdym razem na wejściu mamy algorytm, który chcemy/musimy z jakiś powodów zrealizować w sposób równoległy. Zmieniają się jedynie narzędzia i architektura, ale idea pozostaje taka sama.

W ramach niniejszych zajęć zajmować się będziemy układami reprogramowalnymi FPGA (ang. *Field Programmable Gate Array*) – rozdział 1.3. Oprócz tego, że są wręcz idealną platformą do realizacji obliczeń równoległych, to nie mają zdefiniowanej na etapie produkcji funkcjonalności. Określenie jak będzie działał układ należy do projektanta logiki. Zatem "programowanie"¹ układów FPGA różni się od programowania CPU/GPU. W pierwszym przypadku

¹w dalszej części skryptu używane będzie pojęcie **projektowanie struktury układów FPGA** dla podkreślenia różnic pomiędzy implementacją algorytmu na platformach CPU/GPU, a FPGA

budujemy logikę (elementy obliczeniowe oraz pamiętające) z podstawowych elementów logicznych. Musimy też określić sposób przepływu danych pomiędzy poszczególnymi modułami. W drugim, architekturę mamy ściśle określona. Nasza rola ogranicza się tylko do utworzenia odpowiedniego zbioru instrukcji.

Nauka umiejętności projektowania logiki realizującej jakieś zadania obliczeniowe, najlepiej w sposób mocno równoległy, stanowić będzie "motto" kursu. W jego trakcie będziemy często odwoływać się do przykładów i aplikacji związanych z przetwarzaniem i analizą strumienia wideo, gdyż jest to jedna z dziedzin, gdzie układy FPGA są chętnie stosowane i mają realną przewagę nad innymi rozwiązaniami. Zadanie jakie stoi przed nami nie jest najłatwiejsze, ale pozwala w odmienny sposób spojrzeć na "programowanie", co jest bardzo rozwijające.

1.2 Od lampy elektronowej do układu FPGA - rys historyczny

Zasada działania większości maszyn cyfrowych jest podobna. Wykorzystują one odpowiednio połączone podstawowe elementy takie jak bramki logiczne (element realizujący obliczenia) i rejesty (element pamiętający) w celu realizacji bardziej złożonych funkcji. W pierwszych komputerach wykorzystywano przekaźniki i lampy elektronowe. Wraz z rozwojem elektroniki zastosowano tranzystory, a później układy scalone. Te ostatnie uległy miniaturyzacji - od dużych obudów, które zawierały kilka bramek logicznych, po nowoczesne procesory wykorzystywane w aplikacjach mobilnych, które charakteryzują się niskim zużyciem energii i dużą wydajnością obliczeniową.

Ten spektakularny rozwój nie byłby możliwy, gdyby nie odpowiednie narzędzia, które umożliwiały projektowanie coraz bardziej skomplikowanych architektur. W latach 80 powstał problem opisu struktury i zachowania układów scalonych. Ciągła miniaturyzacja wymuszała tworzenie coraz bardziej skomplikowanych struktur, natomiast wykorzystywane metody projektowania układów przy pomocy schematu ideowego nie pozwalały na osiągnięcie tego celu w prosty sposób. W związku z tym, powstało zapotrzebowanie na język opisu sprzętu. Jednym z lepiej znanych jest opracowany w latach 80 przez Departament Obrony Stanów Zjednoczonych język VHDL.

Języki opisu sprzętu miały szereg zalet w stosunku do schematów ideowych. Po pierwsze pozwalały na łatwe wykorzystanie poprzednio stworzonych systemów w nowych rozwiązaniach. Po drugie szybko zaproponowano narzędzia, które na podstawie opisu na wyższym poziomie pozwalały na stworzenie rzeczywistej struktury układu cyfrowego (na poziomie tranzystorów). Zauważono również, że zapis struktury i zachowania układów cyfrowych przy pomocy języka umożliwia weryfikację i symulację powstałego rozwiązania. Cechy te okazały się bardzo ważne i przesądziły o sukcesie języków opisu sprzętu. Pozwalały one ograniczyć czas wymagany na tworzenie nowych układów częściowo opartych o poprzednie rozwiązania. Poza tym, symulacja umożliwiała usunięcie wielu wad, bez konieczności długotrwałego i kosztownego procesu produkcji i testowania prototypowych układów scalonych.

1.3 Budowa układów FPGA serii Spartan 6 firmy Xilinx

Jak już zostało wspominane, aby dobrze projektować logikę dla układów FPGA należy poznać podstawy ich budowy. W niniejszym rozdziale zostaną zatem omówione podstawowe zasoby logiczne dostępne w układach Spartan 6 firmy Xilinx. Ogólny opis rodziny Spartan 6 można odnaleźć w dokumencie [3].

1.3.1 Blok CLB

Podstawowym elementem, z którego zbudowany jest układ FPGA, to blok CLB (ang. *Configurable Logic Block*). Złożony on jest z dwóch elementów *Slice* i połączony bezpośrednio

z matrycą przełączzeń (ang. *Switch Matrix*). Zostało to pokazane na rysunku 1.3.1. Widoczne linie CIN i COUT, to szybka logika przeniesienia, wykorzystywana przy realizacji operacji arytmetycznych.

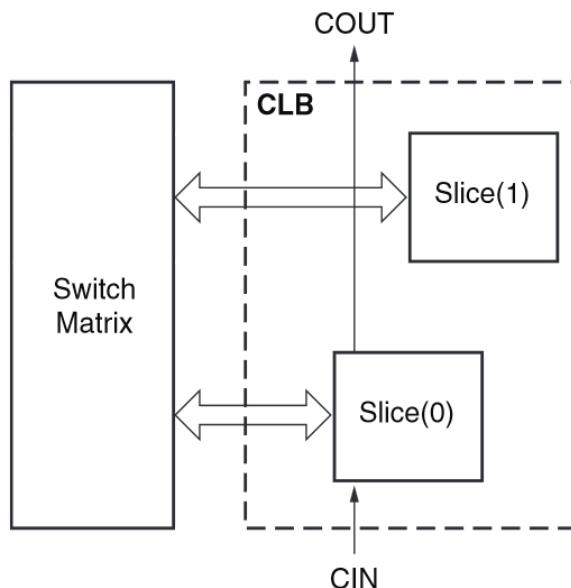


Figure 1.1: Schemat budowy bloku CLB. Źródło: [3]

W układzie Spartan 6 występują trzy rodzaje slice'ów: SLICEX, SLICEL, SLICEM (odpowiednio 50%, 25%, 25% wszystkich w układzie). Schemat budowy pojedynczego *slice'u* typu M zamieszczono na rysunku 1.3.1.

Złożony on jest z następujących elementów:

- generator funkcyjny (4 sztuki) — został zrealizowany jako LUT (ang. *look-up table*) posiadający 6 wejść i dwa niezależne wyjścia. Zatem możliwe jest zaimplementowanie: 6-wejściowej funkcji logicznej, dwóch 5-wejściowych funkcji logicznych (ze wspólnym wejściem), dwóch funkcji logicznych z 3 i 2 wejściami. Ponadto multipleksery umożliwiają tworzenie funkcji 7- i 8-wejściowych poprzez łączenie elementów LUT (SLICEL i SLICEM). Generator funkcyjny może zostać też skonfigurowany jako (tylko SLICEM):
 - synchroniczna pamięć RAM, zwana pamięcią rozproszoną (ang. *Distributed RAM*) — o różnym rozmiarze i liczbie portów (256×1 jednoportowa, 128×1 dwuportowa i 64×1 czteroportowa), przy czym jeden port umożliwia synchroniczny zapis i asynchroniczny odczyt, a pozostałe asynchroniczny odczyt,
 - 32-bitowy rejestr przesuwny wykorzystywany przy tworzeniu linii opóźniających.
 - przerzutnik typu D (FF — ang. *Flip-Flop*) — 8 sztuk), z czego 4 mogą zostać skonfigurowane jako przerzutnik typu D lub zatrzask (ang. *latch*), a 4 tylko jako przerzutnik D,
 - multipleksery — do łączenia elementów LUT (tylko SLICEL i SLICEM),
 - szybka logika przeniesienia — wykorzystywana przy realizacji operacji arytmetycznych.
- Bardziej szczegółowe informacje dostępne są w dokumencie [3] na stronie www.xilinx.com.

1.3.2 Pozostałe zasoby

Wśród pozostałych zasobów dostępnych w układzie FPGA Spartan 6 warto wymienić:

- CMT (ang. *Clock Management Tiles*) — bloki zarządzania sygnałem zegarowym, które zapewniają generację różnych częstotliwości zegara, równomierną propagację sygnału

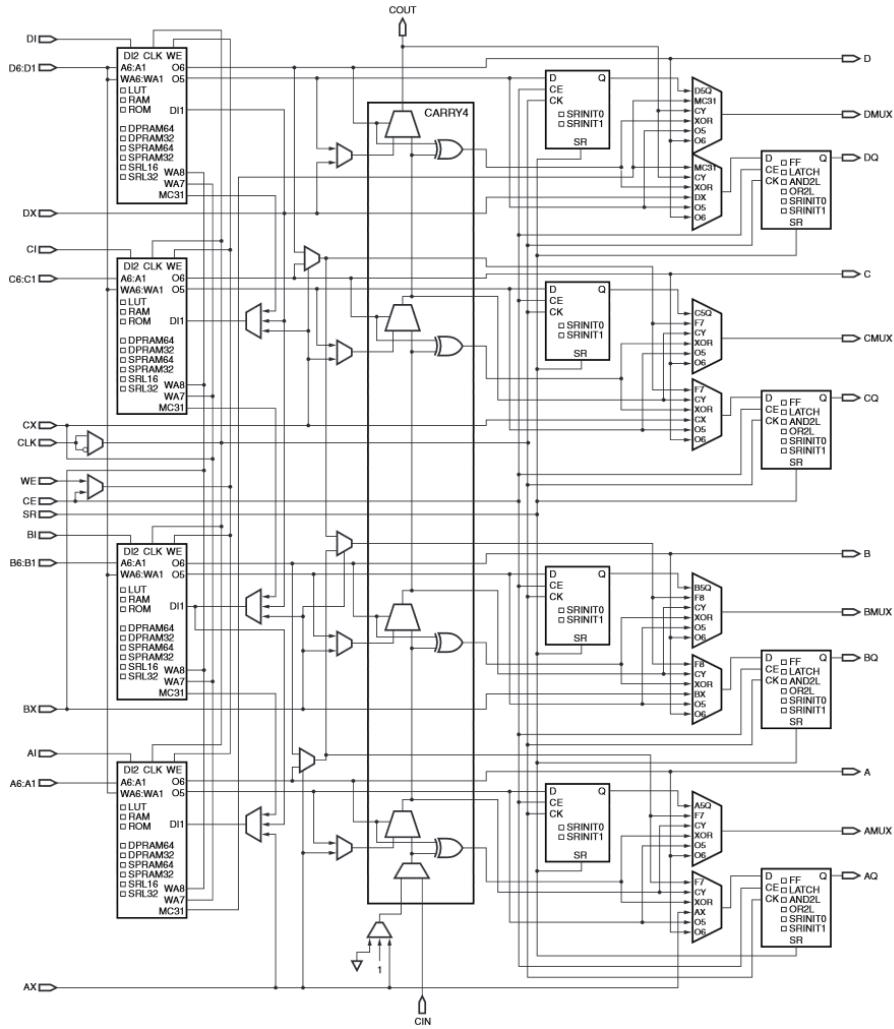


Figure 1.2: Schemat budowy slice'u typu M. Z lewej cztery elementy LUT, pośrodku szybka logika przeniesienia, po prawej przerzutniki (4+4) oraz multipleksery. Źródło: [3]

zegarowego oraz tłumienie zjawiska *jitter* (zakłócenia fazy zegara). W układzie znajdują się od 2 do 6 tego typu modułów. Szczegółowe informacje dostępne w dokumentacji [9],

- Block RAM (BRAM) — bloki dedykowanej dwuportowej pamięci RAM o rozmiarze 18 Kb, które mogą zostać również skonfigurowane jako moduły FIFO. Dostępny rozmiar pamięci w zakresie od 216 Kb do 4824 Kb. Szczegółowe informacje dostępne w dokumentacji [10],
- DSP48A1 — moduły z mnożarką 18×18 bitów oraz 48-bitowym akumulatorem. Ich liczba waha się od 9 do 180 w zależności od rozmiaru układu. Szczegółowe informacje dostępne w dokumentacji [4],
- Select I/O — zasoby wejścia/wyjścia, podzielone na banki (liczba banków zależy od typu, rozmiaru i obudowy układu i waha się od 102 do 576 końcówek). Mogą zostać skonfigurowane do pracy z wieloma standardami (pojedynczymi i różnicowymi): LVCMOS, LVTTL, HSTL, PCI, SSTL, LVDS i innym. Szczegółowe informacje dostępne w dokumentacji [11],
- GTP Transceivers — moduły nadawczo-odbiorcze umożliwiające szybką transmisję szeregową z prędkością do 3,2 Gb/s. Wykorzystywane w interfejsach: Serial ATA, Aurora, 1G

Ethernet, PCI Express i innych. Ich liczba waha się od 0 do 8. Nie występują we wszystkich układach z rodziną Spartan 6. Szczegółowe informacje dostępne w dokumencie [5],

- Zintegrowany moduł PCI Express — wspiera transmisję z przepustowością 2,5 Gb/s w standardzie PCI Express 1.1. Nie występuje we wszystkich układach z rodziną Spartan 6. Szczegółowe informacje dostępne w dokumentach [6] i [7],
- Zintegrowane kontrolery zewnętrznej pamięci RAM — moduły stanowiące kontrolery dla pamięci DDR, DDR2, DDR3 oraz LPDDR. Obsługują transmisję do 800 Mb/s oraz umożliwiają tworzenie dostępów wieloportowych. Ich liczba waha się od 0 do 4. Nie występują we wszystkich układach z rodziną Spartan 6. Szczegółowe informacje dostępne w dokumencie [8]

Ponadto warto zwrócić uwagę, że oprócz zasobów logicznych, w układzie FPGA występuje cały szereg zasobów połączeniowych w formie globalnych i lokalnych linii. Wyróżnia się linie lokalne o pojedynczej, podwójnej i poczwórnej długości oraz globalne. Osobne zasoby połączeniowe służą do dystrybucji sygnału zegarowego.

2 — Układy FPGA — pierwsze kroki

2.1 Wstęp

Aby projektować strukturę układów FPGA potrzebne są trzy elementy:

- język w którym opiszemy tworzoną architekturę,
- środowisko programistyczne i "kompilator",
- platforma sprzętowa – karta z układem FPGA.

W ramach niniejszego ćwiczenia zapoznamy się, w stopniu podstawowym, z każdym z elementów. Stworzymy także pierwszą logikę i wgramy ją na kartę z układem FPGA. Zatem przejdziemy, w dużym uproszczeniu, cały proces tworzenia logiki – od pomysłu, poprzez wykonanie, po uruchomienie i weryfikację sprzętową (tj .sprawdzenie czy działa na karcie tak jak sobie to wyobrażałyśmy).

Zacząć musimy jednak od kilku podstawowych informacji o wspominanych trzech elementach.

2.2 Język opisu sprzętu Verilog

Na laboratoriach będziemy wykorzystywać język opisu sprzętu Verilog, który został zaproponowany około roku 1984 przez firmę Gateway Design Automation Inc. Od tego czasu przeszedł wiele modyfikacji, został upublicznyony na zasadzie otwartego standardu i dokonano jego oficjalnej standaryzacji jako norma IEEE 1364. W porównaniu do języka VHDL, którego składnia oparta jest o język ADA, język Verilog został częściowo oparty o składnię języka C.

Oba języki nie były oryginalnie pomyślane jako narzędzia do projektowania struktury układów elektronicznych. Ich początki to poszukiwanie dobrego rozwiązania do dokumentowania i symulowania coraz bardziej złożonych systemów elektronicznych (połowa lat 80 XX wieku). Pomyśl, aby "przetwarzać" kod na logikę (co określa się mianem syntezy) pojawił się dopiero później. Stąd w językach tych występuje szereg instrukcji, których nie da się zrealizować w FPGA, np. otwieranie plików.

Jeśli porównać kod o identycznej funkcjonalności w VHDL'u i Verilog'u, to pierwsze co rzuci nam się w oczy to liczba linii. Zapis w Verilog'u jest dużo bardziej zwarty. Ma to swoje zalety (szybciej tworzy się logikę), ma też i wady (łatwiej o błąd). Od strony "możliwości" oba języki są zbliżone. Ponadto warto podkreślić, że możliwe jest bezproblemowe łączenie obu w ramach jednego projektu tj. część modułów może być opisana w VHDL'u, a część w Verilog'u

(pod warunkiem, że nazwy portów nie są słowami kluczowymi w żadnym z języków).

2.3 ISE Design Suite — środowisko programistyczne

Na laboratoriach będziemy korzystać w układów FPGA serii Spartan 6 firmy Xilinx. Dedykowane dla nich środowisko programistyczne to ISE Design Suite. W pracach używać będziemy wersji 14.6. Środowisko to nie różni się znacząco od typowych IDE np. Visual Studio lub Eclipse.

Jednakże kluczowe dla zrozumienia istoty projektowania logiki jest przeanalizowanie, co dzieje się z napisanym kodem zanim można go wgrać na kartę z układem FPGA. Etapy tworzenia logiki w układzie FPGA przedstawiono na rysunku 2.3:

- stworzenie projektu — rozumiane zarówno jako stworzenie nowego projektu w środowisku ISE (ustawienie parametrów), jak i opisanie wykorzystywanej logiki (VHDL, Verilog),
- synteza (Synthesize-XST) — na wejściu dostępne są pliku HDL (VHDL, Verilog), które są komplikowane do specyficznej dla danej architektury netlisty (tj. opisu logiki w postaci dostępnych dla danej architektury modułów i połączeń pomiędzy nimi),
- dodanie ograniczeń użytkownika (User Constraints) — przypisanie poszczególnych sygnałów występujących w projekcie do pinów układu FPGA, ustalenie ograniczeń czasowych, ustalenie ograniczeń lokacyjnych,
- implementacja projektu (Implement Design) — składa się z trzech podetapów:
 - translacji (Translate) — na tym etapie wszystkie netlisty łączone są z ograniczeniami i tworzony jest plik NGD (*Xilinx Native Generic Database*), który stanowi opis logiki zredukowany do modułów sprzętowych dostępnych w konkretnym układzie firmy Xilinx,
 - mapowania (Map) — logika opisana w pliku NGD jest mapowana na konkretne elementy występujące w układzie FPGA (bloki CLB i IOB). W wyniku powstaje plik NCD (ang. *Native Circuit Description*),
 - rozmieszczania i łączenia (Place & Route) — logika z pliku NCD jest rozmieszczana i łączona w docelowym układzie,
- analiza rezultatów implementacji — głównie interesuje nas spełnienie ograniczeń czasowych, a także ogólne zużycie zasobów i ew. zużycie energii,
- generowanie pliku konfiguracyjnego (Generate Programming File) — na podstawie wyników poprzedniej fazy tworzony jest plik konfiguracyjny (tzw. plik bit), który następnie może być wgrany do układu FPGA.
- zaprogramowanie układu FPGA i weryfikacja w sprzęcie — ostateczną pewność co do poprawności działania wykonanej logiki zyskuje się po wgraniu (program iMPACT) i uruchomieniu jej na docelowej platformie sprzętowej i poddaniu szeregu testów.

Na poszczególnym etapach możliwe są:

- symulacja (Simulation) — weryfikacja sprzętowa tj. na karcie z układem FPGA nie jest podstawowym narzędziem sprawdzenia czy stworzona logika działa dobrze. Jest to spowodowane przez co najmniej dwa czynniki: proces implementacji projektu zwykle jest dość czasochłonny i nawet dla średnio skomplikowanych systemów może trwać kilka godzin, po wgraniu logiki na kartę zwykle uzyskujemy dość ubogą informację pt. działa/nie działa. O ewentualnych przyczynach nie mamy informacji (wyjątek ChipScope omówiony poniżej). Dużo lepszym i szybszym rozwiązaniem jest symulacja, gdzie praktycznie możemy uzyskać kompletną informację o zachowaniu się modułu. Możemy ją wykonać na różnym etapie projektu. Zagadnienie to zostanie szczegółowo omówione w rozdziale ??.
- wprowadzanie poprawek do projektu.

Analiza działania logiki w układzie FPGA możliwa jest z wykorzystaniem programu ChipScope (Analyze Design Using ChipScope). Jest to analizator stanów logicznych, który może

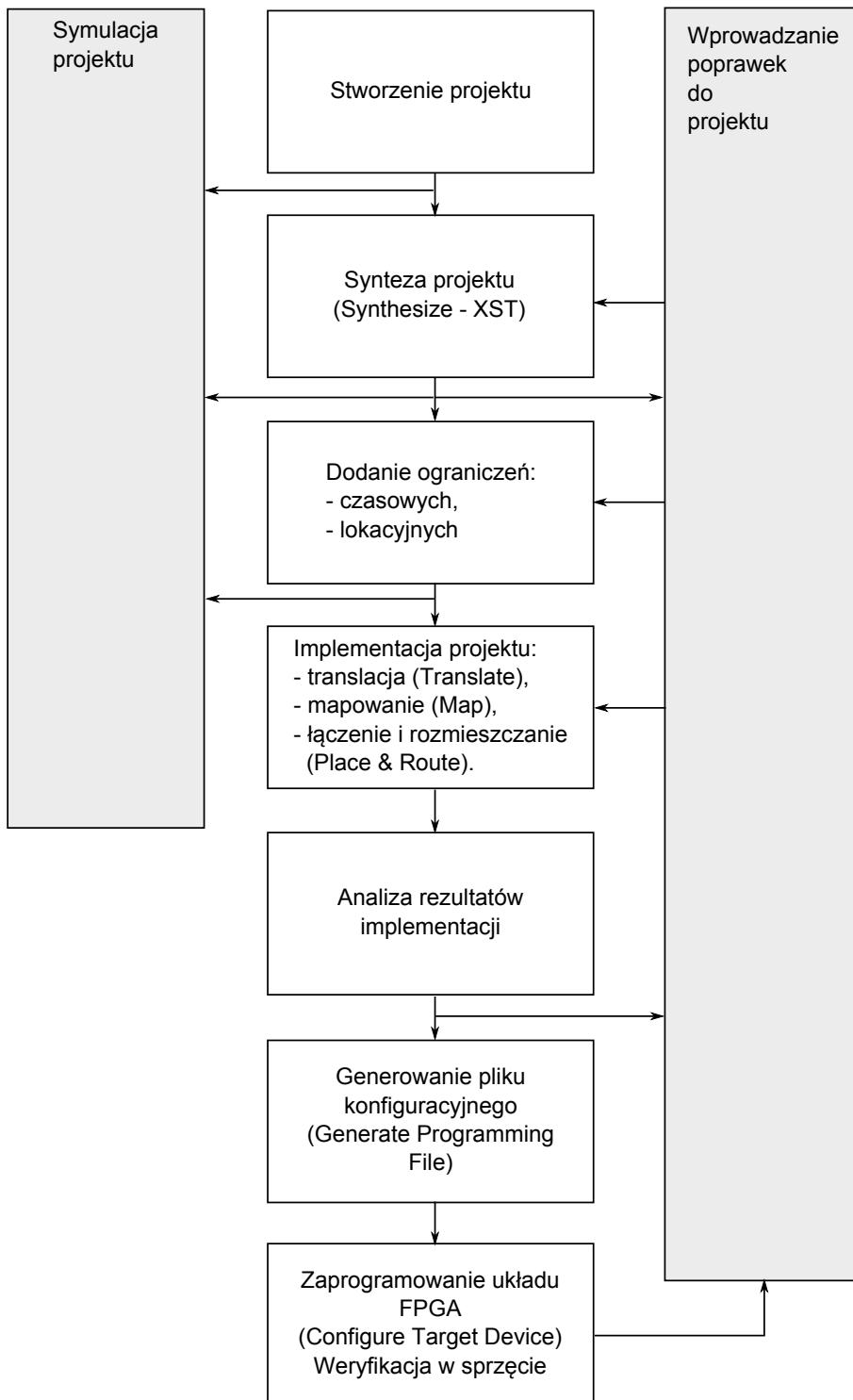


Figure 2.1: Etapy tworzenia logiki w układzie FPGA. Źródło: opracowanie własne na podstawie materiałów firmy Xilinx

zostać dołączony do logiki w układzie FPGA. Umożliwia podgląd wartości sygnałów podczas pracy układu. Jest on przydatny przy tworzeniu interfejsów do urządzeń zewnętrznych, gdyż w tym przypadku zwykle nie jest możliwe wykonanie symulacji rozwiązania.

Z powyższego opisu wyraźnie wynikają różnice, w tworzeniu projektu na CPU i FPGA.

2.3.1 ISE WebPACK

Program ISE Design Suite występuje w trzech wersjach.

- darmowej – ISE WebPACK,
- do urządzeń wbudowanych – Embedded Edition (bez narzędzia System Generator for DSP),
- pełnej – System Edition.

Na laboratorium będziemy używać wersji System Edition. Ponieważ w ramach kursu proponowane będą różne zadania domowe oraz dodatkowe celowe wydaje się zainstalowanie wersji darmowej tj. ISE WebPACK. Z punktu widzenia funkcjonalności nie różni się ona od wersji System Edition. Ograniczono tylko możliwe do wyboru układy FPGA (do tych mniejszych). Układ FPGA zawarty na płycie Atlys jest wspierany przez wersję WebPACK.

Instalacja jest dość prosta. Na stronie:

<http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html>

należy wybrać Download ISE WebPACK software for Windows and Linux. Zostaniemy przeniesieni na stronę **Downloads**. Tam wybieramy **ISE Design Tools** i **ISE Design Suite - 14.7 Full Product Installation**. Oczywiście system *de gustibus*. Niestety należy się zarejestrować na stronie Xilinx'a. Po rejestracji i logowaniu uzyskujemy dostęp do ściągania. Można ściągnąć via downloader lub bezpośrednio (na stronie Downloads jest opis jak). Po instalacji należy wybrać licencję WebPACK i uzyskać ją na stronie www (wykorzystuje się to samo konto).

2.4 Atlys — platforma sprzętowa

Zdjęcie używanej na zajęciach platformy sprzętowej przedstawiono na rysunku 2.4.

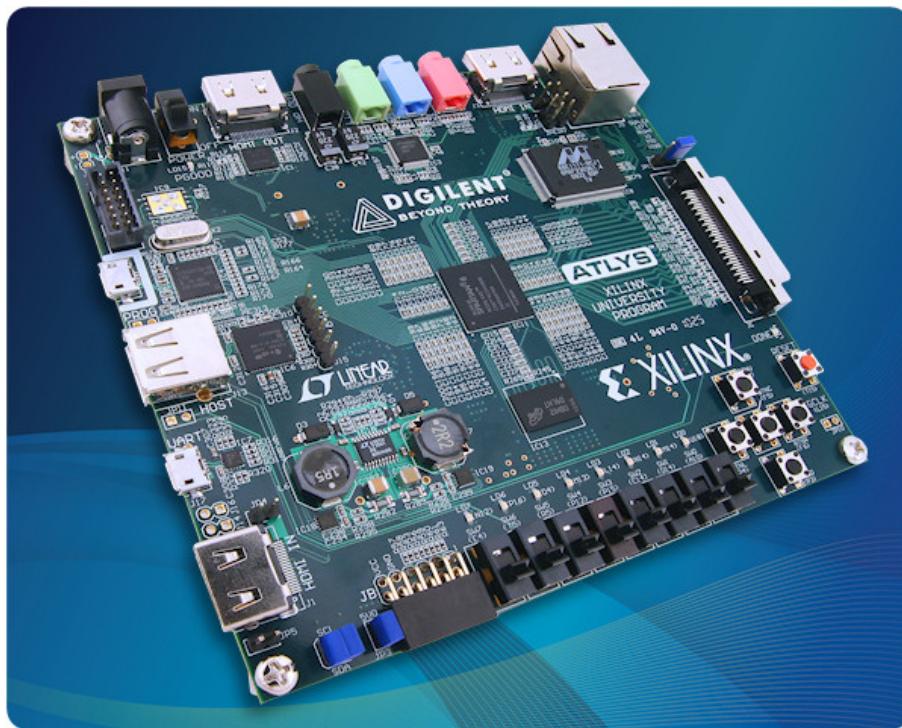


Figure 2.2: Karta uruchomieniowa Atlys firmy Digilent z układem FPGA Spartan 6 firmy Xilinx

Jej podstawowym elementem jest układ FPGA Spartan 6 LX45 firmy Xilinx. Płasuje się on "w środku" rodziny Spartan 6. Ponadto na płycie umieszczono:

- 128 MB pamięci RAM DDR2,

- kontroler Ethernet (10/100/1000),
- porty USB (programowanie, transfer danych, obsługa klawiatury lub myszy),
- dwa wejścia HDMI oraz dwa wyjścia HDMI,
- moduł AC-97 (wejście liniowe, wyjście liniowe, mikrofon, słuchawki),
- 16 MB pamięci SPI Flash (x4) do przechowywania konfiguracji i danych,
- oscylator CMOS 100 MHz,
- 48 uniwersalnych wejść/wyjść,
- 8 diod LED, 6 przycisków oraz 8 przełączników.

2.4.1 Podłączanie i odłączanie kart FPGA Atlys

Podłączanie:

- wyciągamy karty z pudełka,
- podpinamy kabel/kable USB (są dwa jeden oznaczony jako PROG służy do programowania układu, a drugi (UART) do komunikacji szeregowej),
- podpinamy zasilacz,
- przełączamy wyłącznik na płytce.

Odłączanie:

- przełączamy wyłącznik na płytce,
- odpinamy zasilacz (ale nie chowamy go do pudełka !),
- odpinamy kabel/kable USB,
- chowamy do pudełka kartę oraz kable USB.

2.5 Zadanie

Stworzyć logikę, która umożliwia sterowanie diodami za pomocą przełączników.

Uwaga. Problem jest dość trywialny, jednak na tym laboratorium zademonstrowane zostaną ważne aspekty pracy w środowisku ISE, które będą niezbędne do wykonania pozostałych ćwiczeń.

2.6 Zadania do wykonania na laboratorium

1. otwórz program **ISE Design Suite 14.X** (skrót do odpowiedniej wersji na pulpicie),
2. utwórz nowy projekt — **File->New Project**,
3. w oknie dialogowym ustal nazwę **Name** (np. intro) oraz folder **Location** (swój folder na dysku D),
4. ustal typ nadzawanego pliku na HDL, pozostałe formaty to (schemat – niewygodny przy dużych projektach, EDIF (forma netlisty) oraz plik NGC/NGD omówiony wcześniej),
5. w kolejnym oknie ustala się dalsze parametry projektu. W szczególności rozważany typ FPGA (Uwaga. Błędne wprowadzanie tych danych uniemożliwi poprawne zaprogramowanie układu FPGA). I tak:
 - rodzina (Family): Spartan 6,
 - konkretny układ (Device): XC6SLX45,
 - obudowa (Package): CSG324,
 - prędkość (Speed): -2 (im bliżej 0 tym układ szybszy),
 - narzędzie do syntezy (Synthesis Tool): XST,
 - narzędzie do symulacji (Simulator): ISim,
 - język HDL (Preferred Language): Verilog.
6. zakończ tworzenie projektu poprzez **Next** i **Finish**,
7. utwórz nowy plik (moduł) **Project->New Source->Verilog Module**, nazwij plik *led_button*, naciśnij **Next**,

8. ustal interfejs modułu (tj. jego sposób komunikacji ze światem zewnętrznym):
 - `sw` — input — sygnał z przełączników (8 bitów). Na płycie jest 8 przełączników dwupołożeniowych (należy zaznaczyć opcję "Bus" i w MSB wpisać 7, a LSB 0).
 - `led` — output — sygnał do diod (8 bitów). Na płycie jest 8 diod (należy zaznaczyć opcję "Bus" i w MSB wpisać 7, a LSB 0).
 - zakończ kreator **Next, Finish**. Uwaga kreator to nie jest jedyny sposób ustalania interfejsu modułu. Można to również zrobić po prostu w edytorze kodu (czasami tak nawet jest szybciej i łatwiej).
9. otworzy się okno edytora z opisem tworzonego modułu w języku Verilog. Proszę zwrócić uwagę na postać modułu (słowa kluczowe `module` i `endmodule` oraz wejście i wyjście).
10. napisz w języku Verilog następującą logikę : stan przełączników powinien być odzwierciedlony na diodach (podpowiedź wykorzystaj polecenie `assign A=B` – tzw. *continuous assignment*).
11. mając utworzony i skończony moduł omówimy środowisko ISE. Po lewej stronie ekranu (rozmieszczenie domyślne) znajduje się okno nawigacji projektu. Ma ono cztery zakładki: **Start** (tworzenie nowego projektu, otwarcie poprzedniego lub uruchomienie przykładu), **Design** (aktualnie otwarta – operacje możliwe do wykonania dla projektu), **Files** (lista wszystkich plików użytych w projekcie), **Libraries** (lista bibliotek użytych w projekcie). W trakcie zajęć będziemy praktycznie korzystać tylko z zakładki **Design**.
 Zakładka Design podzielona jest na dwa okna: **Hierarchy** i **Processes**. W pierwszej z nich pokazana jest hierarchia projektu. Obecnie jest tam tylko jeden plik `led_button`. Warto zwrócić uwagę, że jest on modułem nadzędnym (*Top Module*) dla projektu. Uwaga. Proces syntez i implementacji można wykonać tylko dla pliku oznaczonego jako nadzędny. Nad plikiem znajduje się ikonka symbolizująca użyty układ FPGA (podany jest jego typ). Dwukrotne kliknięcie na niego pozwala np. na modyfikację typu układu FPGA. Z lewej strony znajdują się przyciski, które odpowiadają podstawowej funkcjonalności (nowy moduł, dodaj moduł, dodaj kopię modułu itp.) To samo możemy uzyskać klikając prawym przyciskiem myszy w pole **Hierarchy**.
 W oknie **Processes** wyszczególnione są wszystkie czynności, które można zrobić z danym plikiem. W rozważanym przypadku nasz plik jest nadzędny zatem możemy dokonać jego implementacji. Proszę zwrócić uwagę, że poszczególne fazy zostały opisane w podrozdziale 2.3. I tak idąc od góry:
 - Design Summary / Reports — kliknięcie powoduje wyświetlanie ekranu z podsumowaniem projektu (raporty, używane zasoby itp.),
 - Design Utilites -> Create Schematic Symbol — pozwala stworzyć schemat dla danego modułu. Przydatne przy pracy ze schematami,
 - Design Utilites -> View Command Line Log File — wyświetlenie logu linii komend (poleceń, które zostały wykonane przez program ISE),
 - Design Utilites -> View HDL Instantiation Template — wyświetlenie szablonu instancji stworzonego modułu (w Verilog'u lub VHDL'u w zależności od ustawień). Bardzo przydatna funkcja,
 - User Constraints — tworzenie ograniczeń użytkownika, nie będzie wykorzystywane w ramach tego kursu,
 - Synthesize - XST — dokonanie syntezы danego projektu,
 - Synthesize - XST -> View RTL Schematic — przeglądarka schematu logiki na poziomie RTL (ang. *Register Transfer Level*). Jest to schemat na poziomie ogólnych elementów elektroniki cyfrowej (liczniuki, dekodery itp.), niezależny od docelowego układu FPGA.
 - Synthesize - XST -> View Technology Schematic — przeglądarka schematu technolo-

logicznego, czyli zbudowanego z podstawowych komponentów logicznych dostępnych w danym układzie FPGA,

- Synthesize - XST -> Check Syntax — sprawdzenie poprawności składniowej kodu,
- Synthesize - XST -> Generate Post-Synthesis Simulation Model — stworzenie modelu symulacyjnego po etapie syntezы,
- Implement Design -> Translate — uruchomienie procesu translacji,
- Implement Design -> Translate -> Generate Post-Translate Simulation Model — stworzenie modelu symulacyjnego po etapie translacji,
- Implement Design -> Map — uruchomienie procesu mapowania,
- Implement Design -> Map -> Generate Post-Map Static Timing — generacja statycznej analizy czasowej po etapie mapowania,
- Implement Design -> Map -> Generate Post-Map Static Timing -> Analyze Post-Map Static Timing — analiza statycznych zależności czasowych po etapie mapowania,
- Implement Design-> Map -> Manually Place & Route (FPGA Editor) — narzędzie do "ręcznej" realizacji procesu rozmieszczenia i łączenia,
- Implement Design-> Map -> Generate Post-Map Simulation Model — stworzenie modelu symulacyjnego po etapie mapowania,
- Implement Design -> Place&Route — uruchomienie procesu rozmieszczenia i łączenia,
- Implement Design -> Place&Route -> Generate Post-Place & Route Static Timing — generacja statycznej analizy czasowej po etapie rozmieszczenia i łączenia,
- Implement Design -> Place&Route -> Generate Post-Place & Route Static Timing -> Analyze Post-Place&Route Static Timing — analiza statycznych zależności czasowych po etapie rozmieszczenia i łączenia,
- Implement Design -> Place&Route -> Analyze Timing / Floorplan Design (Plan Ahead) — narzędzie do analizy czasowej oraz ręcznej korekty rozmieszczenia logiki,
- Implement Design -> Place&Route -> View/Edit Routed Design (FPGA Editor) — przeglądarka schematu po fazie place & route (widok rzeczywistego układu FPGA) wraz z możliwością dokonywania korekty w połączeniach,
- Implement Design -> Place&Route -> Analyze Power Distribution (XPower Analyzer) — narzędzie do estymacji zużycia energii przez daną logikę,
- Implement Design -> Place&Route -> Generate Text Power Report — generacja raportu o używanych napięciach i źródłach energii,
- Implement Design -> Place&Route -> Generate Post-Place&Route Simulation Model — stworzenie modelu symulacyjnego po etapie rozmieszczenia i łączenia,
- Implement Design -> Place&Route -> Generate IBIS model — tworzenie modelu IBIS (ang. *Input Output Buffer Information Specification*) tj. listy pinów oraz modeli modułów wejścia/wyjścia,
- Implement Design -> Place&Route -> Generate IBIS model -> View IBIS model — przeglądarka modelu IBIS,
- Implement Design -> Place&Route -> Back-annotate Pin Locations — wstępna analiza lokalizacji pinów tj. uaktualnianie pliku UCF na podstawie rezultatów fazy rozmieszczenia i łączenia,
- Implement Design -> Place&Route -> Back-annotate Pin Locations -> View Locked Pin Constraints — wstępna analiza plików, przy czym wyniki nie są zapisywane do pliku UCF, a odrębnego LPC,
- Generate Programming File — utworzenie pliku konfiguracyjnego bit,
- Configure Target Device — uruchomienie narzędzia iMPACT, które służy do ładowania konfiguracji do układu FPGA,

- Configure Target Device -> Generate Target PROM/ACE File — tworzy plik, który może zostać wgrany do układu FPGA z zewnętrznego procesora, pamięci PROM, Flash etc.,
- Configure Target Device -> Manage Configuration Project (iMPACT) — możliwość konfiguracji programu iMPACT,
- Analyze Design Using ChipScope — uruchomienie aplikacji analizatora stanów logicznych ChipScope.

Uwagi:

- istnieje możliwość kliknięcia prawym przyciskiem myszy na każdą z faz. Pojawiają się wtedy dodatkowe opcje.
- dla modułu (pliku Verilog), który nie jest nadzędny w projekcie (Top Module) możliwości ograniczają się do: **Create Schematic Symbol**, **View HDL Instantiation Template**, **Check Syntax**.
- ogólny schemat postępowania. W oknie **Hierarchy** ustawiamy plik, a w oknie **Processes** co chcemy z nim zrobić. Zawsze warto sprawdzić co zaznaczyliśmy.
- u góry, nad oknem **Hierarchy** istnieje możliwość wyboru pomiędzy dwoma widokami (**View**): **Implementation** i **Simulation**. Pierwszy używamy podczas implementacji projektu na układ FPGA, a drugi podczas symulowania logiki (por. rozdział ??).

Od tej pory uznaje się, że techniczne aspekty syntezy oraz implementacji projektu są znane i w dalszych instrukcjach nie będą opisywane (zawsze można wrócić do powyższego opisu).

Bardziej szczegółowy opis wszystkich etapów dostępny jest w pomocy do programu ISE:
http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/isehelp_start.htm

12. sprawdź poprawność składniową stworzonego modułu (**Check Syntax**),
13. dodaj do projektu plik *AtlysGeneral_intro.ucf* (plik w którym podane są połączenia między sygnałami użytymi w projekcie, a fizycznymi portami I/O FPGA) — plik dostępny w archiwum dołączonym do ćwiczenia.
14. dodany plik UCF pojawi się w hierarchii. Zaznacz go i w **Processes** wybierz **User Constraints -> Edit Constraints (Text)**
15. odszukaj sekcje odpowiedzialne za diody (Leds) i przełączniki (Switches). Odkomentuj stosowne linijki.
16. projekt jest gotowy do syntezy i implementacji. W oknie **Sources** wybierz *led_button*. W oknie **Processes** wybierz **Generate Programming File**,
17. przejrzyj raport — Design Sumary. Zwróć szczególną uwagę na zużycie zasobów logicznych, a raczej jego brak (*Slice Registers* i *LUTs*) oraz *IOBs*,
18. skonfiguruj układ FPGA karty Atlys. Uruchom **Configure Target Device**. Otworzy się okno programu ISE iMPACT. Upewnij się, że karta jest podłączona do zasilania oraz komputera PC kablem USB (do portu PROG, a nie UART !),
19. wybierz **File -> New Project**. Zaakceptuj automatyczne stworzenie projektu. W oknie dialogowym, które się pojawi kliknij OK. Na pytanie czy przypisać pliki konfiguracyjne odpowiedz TAK. Odszukaj plik *led_button.bit* (w swoim folderze). Na pytanie o SPI lub BPI PROM odpowiedz NIE. W kolejnym oknie kliknij OK,
20. zaprogramuj układ. Kliknij prawym klawiszem myszy na ikonce układu FPGA i wybierz **Program**,
21. przetestuj działanie układu tj. czy zmiana stanu przełącznika skutkuje zaświeceniem się odpowiedniej diody.

2.7 Zadania do wykonania w domu

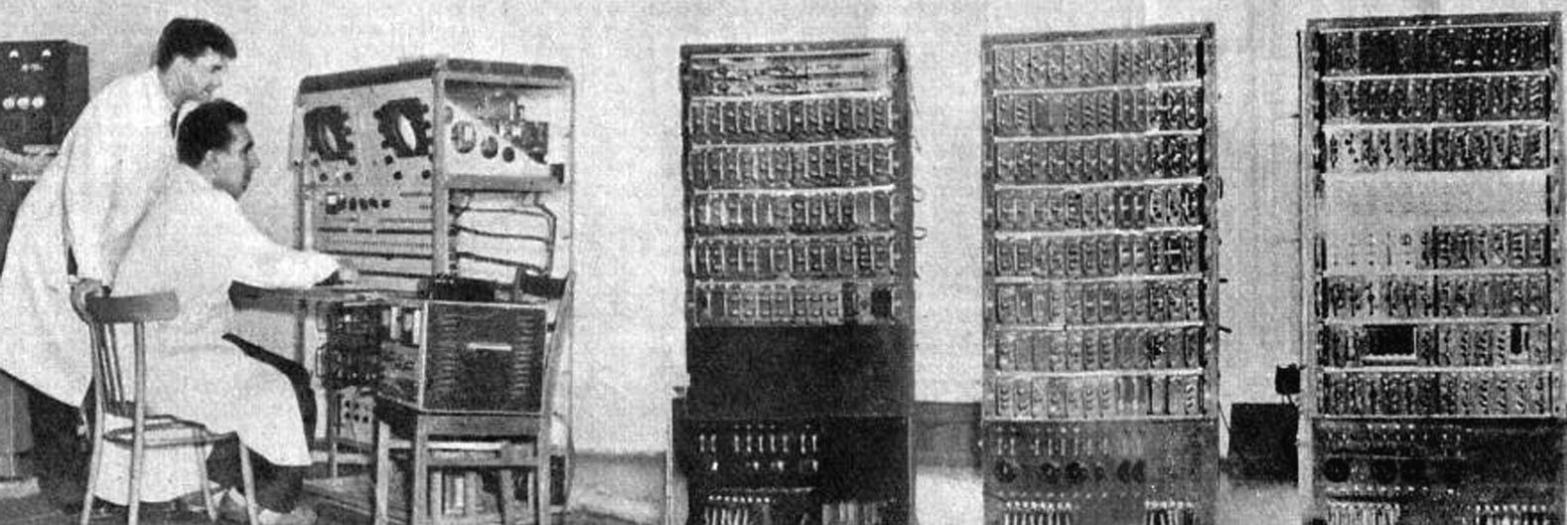
Zadanie 2.1 Proszę pobrać i zainstalować program ISE Design Suite ze strony www.xilinx.com – opis w rozdziale 2.3.1.

Zadanie 2.2 Proszę zapoznać się z podstawowymi informacjami o budowanie układów FPGA – rozdział 1 niniejszego skryptu.

2.8 Podsumowanie

Po ukończeniu niniejszego laboratorium, zakłada się, że każdy uczestnik potrafi:

- wykorzystywać narzędzie ISE DS w zakresie tworzenia nowego projektu, dodawania do niego plików oraz ich syntezy i implementacji do postaci plików konfiguracyjnych (tzw. bitów),
 - odpowiednio podłączyć kartę Atlys do komputera oraz zaprogramować układ FPGA.
- Zakłada się również, że uczestnik laboratorium zna i rozumie:
- etapy prowadzące od pliku w języku HDL do jego realizacji w postaci pliku konfiguracyjnego,
 - specyfikację i dostępne peryferia układów FPGA z rodziny Spartan 6 oraz karty ewaluacyjnej Atlys.



3 — Wstęp do projektowania struktury układów FPGA

3.1 Język Verilog - wprowadzenie

W niniejszym rozdziale zostaną przedstawione podstawowe elementy strukturalne występujące w języku Verilog, które umożliwiają projektowanie logiki w układach rekonfigurowalnych.

3.1.1 Moduł

Moduł jest podstawowym elementem, który jest wykorzystywany do opisywania struktury układów scalonych w języku Verilog. W zależności od potrzeb programisty, może realizować funkcjonalność pojedynczej bramki, rejestru lub wielordzeniowego procesora. Moduł jest niejako "czarną skrzynką", która posiada określony zbiór portów wejścia i wyjścia (por. rysunek 3.1.1). Z zewnątrz można również dostarczyć zestaw parametrów, które mogą zmieniać zachowanie elementów wewnętrznych modułu – przykładowo szerokość danych wejściowych lub wyjściowych (por. rysunek 3.1.1). W języku Verilog, moduł odpowiada najczęściej jednemu plikowi o rozszerzeniu .v i jest definiowany następującym kodem:

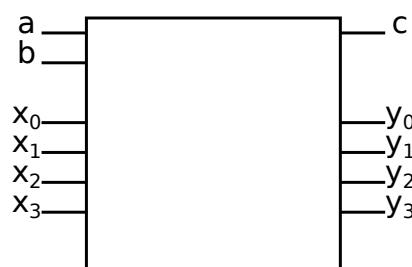


Figure 3.1: Przykładowy moduł

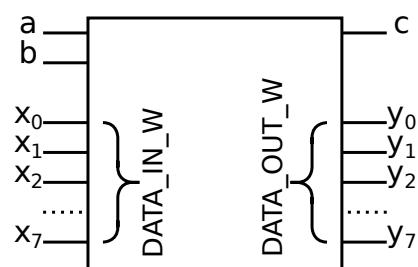


Figure 3.2: Moduł z portami o parametryzowalnej szerokości

Kod 3.1.1 — Moduł:

```
module simple_module
(
    //input ports
    input a,
    input b,
    input [3:0]x,
    //output ports
    output c,
    output [3:0]y
);
//module content
endmodule
```

Kod 3.1.2 — Moduł parametryzowalny:

```
module module_with_param#
(
    parameter DATA_IN_W=8,
    parameter DATA_OUT_W=8
)
(
    //input ports
    input a,
    input b,
    input [DATA_IN_W-1:0]x,
    //output ports
    output c,
    output [DATA_OUT_W-1:0]y
);
//module content
endmodule
```

3.1.2 Opis połączeń

Drugim podstawowym elementem wykorzystywanym do opisu struktury układów jest "ścieżka" (wire). Jest ona używana do łączenia poszczególnych modułów między sobą i tworzenia bardziej złożonych struktur. Do ścieżki można również przypisać stałą wartość przy inicjalizacji lub przy pomocy wyrażenia **assign**. Ścieżka może składać się z pojedynczej linii lub być wielobitową szyną danych. W języku Verilog jest definiowana przy pomocy wyrażenia **wire**.

Kod 3.1.3 — Moduł z połączeniami:

```

module module_with_wires
(
    //input ports
    input a,
    input b,
    //output ports
    output c
);
//module content
wire wire0;
wire bus0;
wire [7:0]fixed0=8'hff;
wire [7:0]fixed1;

assign fixed1=8'hcc;
endmodule

```

Kod 3.1.4 — Zmiana połączeń:

```

module module_with_wires
(
    //input ports
    input a,
    input b,
    //output ports
    output c
);
//module content
wire wire0=1'b0;
wire wire1=1'b1;
wire [7:0]fixed0=8'hff;
wire [1:0]bus0;
wire [1:0]bus1;

assign bus0={wire0,wire1};
assign bus1=fixed0[4:3];
endmodule

```

Sygnały mogą być łączone w jeden, przy pomocy wyrażenia $\{sygnal1, sygnal2\}$ lub z danej szyny danych można wybrać interesujący zakres bitów (od a do b), przy pomocy wyrażenia $sygnal[a : b]$. Proszę zwrócić uwagę, że język Verilog dopuszcza indeksowanie szyn "od góry" np. $[7 : 0]$, jak i "od dołu" $[0 : 7]$. W trakcie laboratoriów będziemy stosować numerowanie "od góry", co pozwoli na uniknięcie błędów wynikających z mieszania sposobów indeksowania.

3.1.3 Zapis liczby

Do zapisu liczb w różnych formatach w języku Verilog wykorzystuje się następujące wyrażenie:

$$X'Yv \quad (3.1)$$

gdzie: X – to wartość określająca liczbę bitów zapisywanej liczby,

Y – jest określa sposób zapisu v (b-binarny, h-heksadecymalny, d-dziesiętny)

v – określa wartość wyrażenia w odpowiednim zapisie

Np. jeśli portowi ma zostać przypisana liczba 123 zapisana na 8 bitach można tego dokonać na kilka sposobów:

Kod 3.1.5 — Zapisanie wartości w różnych formatach:

```

wire [7:0]value;
assign value=8'd123; //decimal
assign value=8'h7b; //hexadecimal
assign value=8'b01111011; //binary

```

3.1.4 Łączenie modułów

Raz zdefiniowany moduł może zostać wielokrotnie wykorzystany w innym module. Instantacja modułów odbywa się na dwa sposoby, w zależności od tego czy wykorzystywany jest moduł z parametrami lub bez. Z prostego modułu danego kodem 3.1.6 (por. rysunek 3.1.4), zbudowano moduł dany kodem 3.1.7 (por. rysunek 3.1.4), który wykorzystuje dwie instancje pierwszego

z modułów.

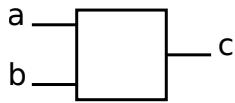


Figure 3.3: Moduł podstawowy

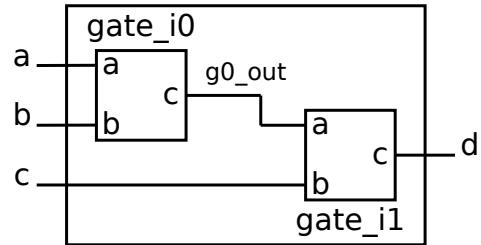


Figure 3.4: Moduł złożony

Kod 3.1.6 — Moduł:

```

module simple_gate #
(
    parameter A=16,
    parameter B=8
)
(
    //input ports
    input a,
    input b,
    //output ports
    output c
);
//module content
endmodule
  
```

Kod 3.1.7 — Złożenie modułów:

```

module module_gates
(
    //input ports
    input a,
    input b,
    input c,
    //output ports
    output d
);
//module content
wire g0_out;

simple_gate gate_i0
(
    .a(a),
    .b(b),
    .c(g0_out)
);

simple_gate #
(
    .A(8),
    .B(4)
)
gate_i1
(
    .a(g0_out),
    .b(c),
    .c(d)
);

endmodule
  
```

Można zauważyć, że ponieważ w każdym module podane są domyślne wartości parametrów, podczas instantacji nie ma konieczności ich ustalania (moduł `gate_i0`), o ile oczywiście nie chce się zmienić ich wartości (moduł `gate_i1`). Proszę zwrócić uwagę na specyficzną składnię modułu parametryzowanego tj. użycie znaku #.

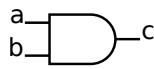
3.1.5 Opis struktury a opis zachowania

Do tej pory, nauczyliśmy się opisywać strukturę układów scalonych na bardzo niskim poziomie (tj. strukturalnym). W dalszej kolejności przejdziemy do opisu zachowania (tzw. opis behawioralny). Naukę tę zaczniemy od przedstawienia modułów, które realizują podstawowe funkcje

logiczne.

3.1.6 Bramka AND

Schemat blokowy, tabela prawdy oraz opis bramki AND w języku Verilog został przedstawiony poniżej:



a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Kod 3.1.8 — Bramka AND:

```
module and_gate
(
    input a,
    input b,
    output c
);
assign c=a&b;
endmodule
```

3.1.7 Bramka OR

Schemat blokowy, tabela prawdy oraz opis bramki OR w języku Verilog został przedstawiony poniżej:



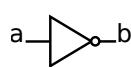
a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

Kod 3.1.9 — Bramka OR:

```
module or_gate
(
    input a,
    input b,
    output c
);
assign c=a|b;
endmodule
```

3.1.8 Bramka NOT

Schemat blokowy, tabela prawdy oraz opis bramki NOT w języku Verilog został przedstawiony poniżej:



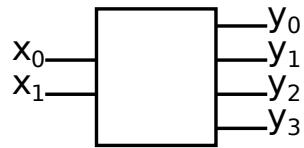
a	b
0	1
1	0

Kod 3.1.10 — Bramka NOT:

```
module not_gate
(
    input a,
    output b
);
assign b=~a;
endmodule
```

3.1.9 Dekoder

Dekoder jest układem cyfrowym, który na wejściu przyjmuje zakodowany numer wyjścia które powinno zostać wyróżnione. Zamienia kod binarny na kod 1 z N. Schemat blokowy, tabela prawdy oraz opis dekodera w języku Verilog został przedstawiony poniżej:



x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

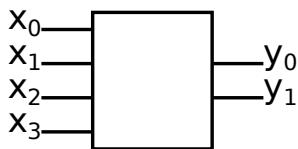
Kod 3.1.11 — Dekoder:

```
module decoder
(
    input [1:0]x,
    output [3:0]y
);
assign y[0]=( (x==2'b00) ?1'b1:1'b0);
assign y[1]=( (x==2'b01) ?1'b1:1'b0);
assign y[2]=( (x==2'b10) ?1'b1:1'b0);
assign y[3]=( (x==2'b11) ?1'b1:1'b0);
endmodule
```

Zwróć uwagę na wyrażenie: $y = \text{warunek logiczny} ? \text{opcja 1} : \text{opcja 2}$.

3.1.10 Koder

Koder jest układem cyfrowym, który na wejście przyjmuje kod 1 z N i zamienia go na kod binarny. Schemat blokowy, tabela prawdy oraz opis kodera w języku Verilog został przedstawiony poniżej:



x_3	x_2	x_1	x_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Kod 3.1.12 — Koder:

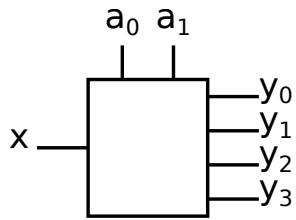
```
module encoder
(
    input [3:0]x,
    output [1:0]y
);

assign y= (x[0]) ? 2'b00:
            (x[1]) ? 2'b01:
            (x[2]) ? 2'b10:
            2'b11;

endmodule
```

3.1.11 Demultiplexer

Demultiplexer jest układem cyfrowym, który w zależności od adresu przełącza wartość wejścia x na jedno z N wyjść y. Schemat blokowy, tabela prawdy oraz opis demultipleksera w języku Verilog został przedstawiony poniżej:



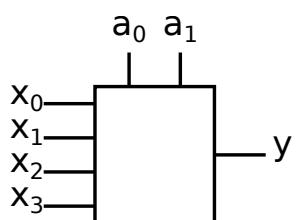
x	a ₁	a ₀	y ₃	y ₂	y ₁	y ₀
v	0	0	0	0	0	v
v	0	1	0	0	v	0
v	1	0	0	v	0	0
v	1	1	v	0	0	0

Kod 3.1.13 — Demultiplexer:

```
module demux
(
    input x,
    input [1:0]a,
    output [3:0]y
);
assign y[0]=((a==2'b00)?x:0);
assign y[1]=((a==2'b01)?x:0);
assign y[2]=((a==2'b10)?x:0);
assign y[3]=((a==2'b11)?x:0);
endmodule
```

3.1.12 Multiplexer

Multiplexer jest układem cyfrowym, który w zależności od adresu przełącza wartość jednego z N wejść x na wyjście y. Schemat blokowy, tabela prawdy oraz opis multipleksera w języku Verilog został przedstawiony poniżej:



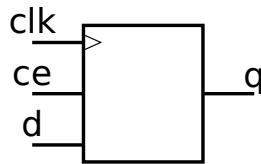
x ₃	x ₂	x ₁	x ₀	a ₁	a ₀	y
v ₃	v ₂	v ₁	v ₀	0	0	v ₀
v ₃	v ₂	v ₁	v ₀	0	1	v ₁
v ₃	v ₂	v ₁	v ₀	1	0	v ₂
v ₃	v ₂	v ₁	v ₀	1	1	v ₃

Kod 3.1.14 — Multiplexer:

```
module mux
(
    input [3:0]x,
    input [1:0]a,
    output y
);
assign y=x[a];
endmodule
```

3.1.13 Rejestr

Rejestr jest podstawowym elementem "z pamięcią". Wartość wyjścia nie zmienia się odpowiednio do każdej zmiany wejścia, ale zmiany są zsynchronizowane z narastającym (lub opadającym) zboczem zegara. Pomiędzy zboczami wartością wyjścia jest ustalona (zarejestrowana). Wartość wyjściowa jest opóźniona o jeden takt zegara w stosunku do wartości wejściowej. Dodatkowo możliwe jest włączanie/wyłączanie rejestru przy pomocy wejścia ce (ang. *clock enable*). Rejestry można łączyć szeregowo i równolegle. W pierwszym przypadku pozwalają na zaprojektowanie tzw. linii opóźniających, w drugim przypadku umożliwiają rejestrowanie wielu bitów. Schemat blokowy, tabela prawdy oraz opis rejestru w języku Verilog został przedstawiony poniżej:



d	ce	clk	q
v	0	↑	v
v	1	↑	v

Kod 3.1.15 — Rejestr:

```

module register
(
    input clk,
    input ce,
    input d,
    output q
);
reg val=1'b0;

always @ (posedge clk)
begin
    if(ce) val<=d;
    else val<=val;
end

assign q=val;

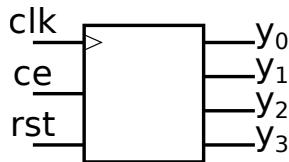
endmodule

```

Warto zwrócić uwagę na kilka aspektów zaprezentowanych w powyższym kodzie. Do zapamiętania wartości `val` wykorzystywany jest rejestr `reg`. Zaprezentowany wcześniej typ `wire` nie "pamięta wartości", a jedynie ją przekazuje (działa jak kabel, ścieżka). Rejestry należy zawsze inicjalizować wartością domyślną. Znaczco ułatwia to późniejszą symulację modułu. Składnia `always @ (posedge clk)` oznacza, że kod zawarty wewnątrz wykona się tylko przy narastającym zboczu sygnału zegarowego (`clk`). W tak opisanym elemencie instrukcje wykonują się sekwencyjnie. Dlatego można użyć polecenia `if` `else`. Poza blokami `always` wszystko wykonuje się równolegle. Również poszczególne bloki `always` (procesy) wykonują się względem siebie równolegle. Zarejestrowaną wartość należy wyprowadzić na port wyjściowy (instrukcja `assign`).

3.1.14 Licznik

Liczniok jest układem cyfrowym, który umożliwia zliczanie czasu (w taktach zegara) trwania danego sygnału. Oprócz wejścia zegarowego, posiada on jeszcze wejście `rst` umożliwiające wyzerowanie licznika oraz wejście `ce`, które aktywuje lub wstrzymuje proces zliczania. W podstawowym trybie pracy, w każdym taktie zegara wartość wyjścia jest inkrementowana o jeden.



ce	rst	clk	y
0	0	↑	y
0	1	↑	0
1	0	↑	y+1
1	1	↑	0

Kod 3.1.16 — Licznik:

```
module cnt
(
    input clk,
    input ce,
    input rst,
    output [3:0]y
);
reg [3:0]val=4'b0000;

always @(posedge clk)
begin
    if(rst) val<=4'b0000;
    else
        if(ce) val<=val+1;
        else val<=val;
end

assign y=val;

endmodule
```

3.1.15 Instrukcja generate

Jedną z najbardziej przydatnych instrukcji jest **generate**. Jej działanie jest zbliżone do makra preprocessora w języku C (#define, #ifdef itd.). Instrukcja ta pozwala na automatyczną generację kodu. Jej działanie może być uwarunkowane przez wartość parametrów modułu. Jej wykorzystanie pozwala na znaczne zaoszczędzenie czasu programisty, poprzez automatyczne tworzenie fragmentów, które się powtarzają. Umożliwia to efektywne tworzenie takich konstrukcji jak drzewa sumacyjne, kaskadowo połączone bramki itd.

W pierwszym przykładzie (kod 3.1.17) zaprezentowano wykorzystanie instrukcji **generate** do opisania bramki, która w zależności od podanego parametru (*mode*) może pełnić rolę bramki AND lub OR.

W drugim przykładzie wykorzystano instrukcję **generate** do opisania modułu, który realizuje funkcjonalność bramki OR o parametryzowej liczbie wejść. Na rysunku 3.5 przedstawiono przykładowy moduł składający się z czterech bramek OR. Przedstawiony kod (kod 3.1.18) generuje odpowiednią liczbę dwuwejściowych bramek OR i łączy je w zadaną strukturę:

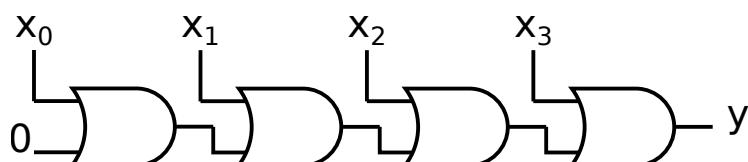


Figure 3.5: Bramka OR o czterech wejściach

Kod 3.1.17 — Bramka OR lub AND:

```

    module or_and_gate #
(
    parameter mode=0
)
(
    input a,
    input b,
    output c
);

generate
    if(mode==0)
        begin
            or_gate gate_i
            (
                .a(a),
                .b(b),
                .c(c)
            );
        end
        else
        begin
            and_gate gate_i
            (
                .a(a),
                .b(b),
                .c(c)
            );
        end
    end
endgenerate
endmodule

```

Kod 3.1.18 — Łańcuch bramek OR:

```

    module long_or #
(
    parameter LENGTH=4
)
(
    input [LENGTH-1:0]x,
    output y
);

wire [LENGTH:0] chain;
assign chain[0]=1'b0;

genvar i;
generate
    for(i=0; i<LENGTH; i=i+1)
        begin
            or_gate gate_i
            (
                .a(x[i]),
                .b(chain[i]),
                .c(chain[i+1])
            );
        end
endgenerate
assign y=chain[LENGTH];
endmodule

```

3.1.16 Maszyna stanów

Maszyny stanów to bardziej złożone moduły, które poprzez sekwencję stanów mogą realizować praktycznie dowolną funkcjonalność. Są wykorzystywane do realizacji protokołów komunikacyjnych, obsługi pamięci RAM, buforów FIFO i wielu innych celów. Wartość wyjścia jest zależna od wartości wejścia oraz od stanu w którym aktualnie znajduje się moduł. Na rysunku 3.6 przedstawiono schemat modułu, diagram blokowy poszczególnych stanów oraz warunków przejścia pomiędzy nimi. Powyższej maszynie stanów odpowiada kod 3.1.19. Stan jest przechowywany w zmiennej *state*, która może przyjmować wartości 0, 1 lub 2. Przy pomocy polecenia **localparam** zdefiniowano trzy parametry (STATE0 – STATE1), w celu oznaczenia poszczególnych stanów nazwami literowymi. Wartość wyjścia jest przechowywana w 2-bitowym rejestrze *r_y*, którego wartość jest podłączona do wyjścia *y*. W celu realizacji maszyny stanów wykorzystano instrukcję **case**. W każdym stanie zdefiniowano wartość, jaka powinna się pojawić na wyjściu *y* oraz warunek na przejście do kolejnego stanu.

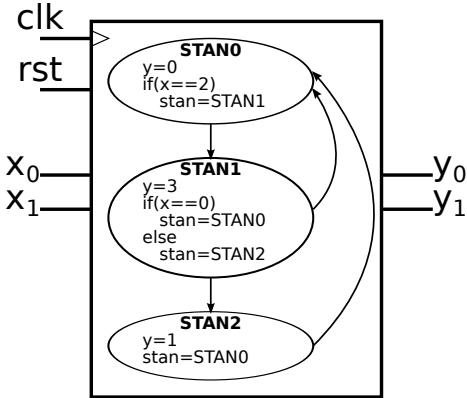


Figure 3.6: Przykładowa maszyna stanów

Kod 3.1.19 — FSM:

```

module fsm
(
    input clk,
    input rst,
    input [1:0]x,
    output [1:0]y
);

localparam STATE0=2'd0;
localparam STATE1=2'd1;
localparam STATE2=2'd2;

reg [1:0] state=STATE0;
reg [1:0] r_y;

always @ (posedge clk)
begin
    if(rst) state<=STATE0;
    else
    begin
        case(state)
            STATE0:
            begin
                r_y<=2'b0;
                if(x==2'b10) state<=STATE1;
            end
            STATE1:
            begin
                r_y<=2'b11;
                if(x==2'b00) state<=STATE0;
                else state<=STATE2;
            end
            STATE2:
            begin
                r_y<=2'b01;
                state<=STATE0;
            end
        endcase
    end
end

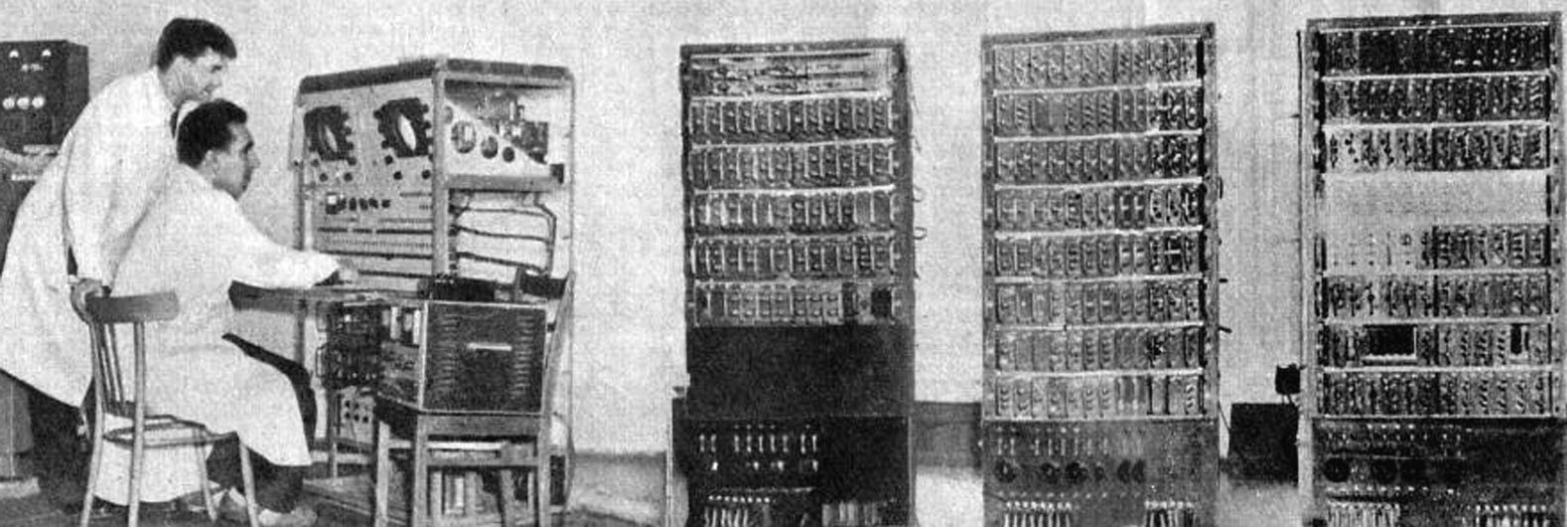
assign y=r_y;

endmodule

```

3.1.17 Moduły arytmetyczne

Moduły arytmetyczne takie jak sumator, subtraktor, mnożarka, dzielarka i inne zostaną omówione w rozdziale 7.



4 — Weryfikacja i testowanie projektu

Testowanie i weryfikacja zaprojektowanych modułów sprzętowych jest bardzo złożonym zagadnieniem. Jest to spowodowane tym, że podczas pracy układu FPGA trudno jest wygenerować odpowiednie sekwencje testowe z wysoką częstotliwością, bez konieczności wykorzystania specjalizowanych urządzeń takich jak generatory sygnałów. Trudno jest również "podejrzeć" stan modułu wewnętrz układow FPGA, co ogranicza możliwości lokalizacji i usuwania błędów. Pewien wyjątek stanowi narzędzie ChipScope Pro, o którym podano kilka informacji w rozdziale ??

W związku z tym, jednym z najczęściej wykorzystywanych sposobów wstępnej weryfikacji zaprojektowanego modułu sprzętowego jest jego symulacja przy pomocy odpowiednich narzędzi programowych. Pozwala to na dokładną analizę na ekranie monitora wyników działania, które mogłyby być trudne do weryfikacji w układzie pracującym z wysoką częstotliwością. Po drugie pozwala na przetestowanie wielu sytuacji, których wygenerowanie w działającym systemie mogłyby być kłopotliwe. Po trzecie, symulacja pozwala na zaoszczędzenie czasu potrzebnego na zsyntezowanie logiki do pliku bit, który umożliwia zaprogramowanie układu FPGA.

Ponieważ podstawowym elementem wykorzystywanym do opisu struktury układów scalonych w języku Verilog jest moduł, weryfikacja zaprojektowanego rozwiązania również opiera się o weryfikację działania modułów. W tym celu wykorzystywana jest metodologia przedstawiona na rysunku 4.1. Tworzone jest środowisko testowe (ang. *testbench*), które będzie podlegało symulacji. Jest ono zbudowane z trzech elementów. Testowany moduł oznaczony jako DUT (ang. *design under test*) i jest umieszczony w środku pomiędzy dwoma modułami. Rolą pierwszego bloku jest generacja odpowiedniej sekwencji sygnałów wejściowych do modułu. Rolą trzeciego elementu jest weryfikacja, czy sygnały wyjściowe z testowanego modułu mają odpowiednie wartości (czy moduł działa poprawnie).

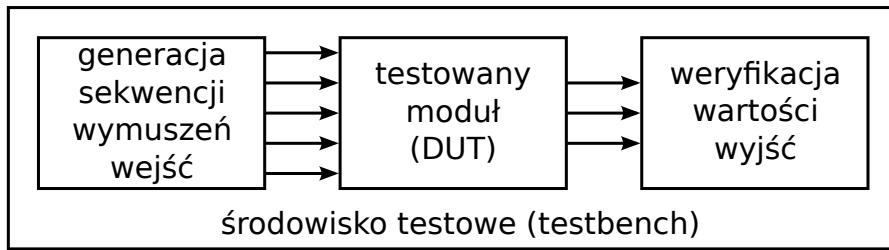


Figure 4.1: Metoda testowania

Wartości wyjść najczęściej zależą nie tylko od wartości wejść, ale również od poprzedniego stanu modułu (np. linia opóźniająca, maszyna stanów itd.). Z tego powodu sekwencje testowe przedstawia się przy pomocy wykresów czasowych (ang. *waveform*). Pokazują one wartości poszczególnych wejść i wyznaczone w trakcie symulacji wartości wyjść w czasie. Na rysunku 4.2 przedstawiono przykład, który umożliwia sprawdzenie czy zaprojektowana bramka AND działa poprawnie. Bramka ma dwa wejścia (a i b) oraz wyjście c. Zgodnie z tabelą prawdy, wyjście c powinno mieć wartość 1, tylko wtedy, gdy zarówno a i b mają wartość 1. Wymuszono więc na wejściach a i b w poszczególnych chwilach czasu sygnały, które pokrywają wszystkie możliwe kombinacje wejść. Zarejestrowano również odpowiedź bramki na takie wymuszenie (sygnał oznaczony jako c). Można zauważyć, że dla przypadku, gdy $a=0$ i $b=1$, wyjście c ma wartość 1. Jest to błąd.

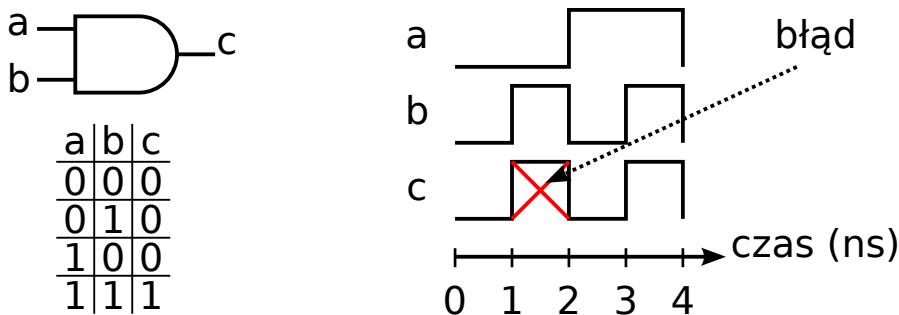


Figure 4.2: Testowanie bramki AND

Oczywiście w przedstawionym przypadku sprawdzenie poprawności jest bardzo proste. Łatwo jest określić wszystkie możliwe stany wejść i zauważać błąd. Dla bardziej skomplikowanych modułów, które posiadają dziesiątki portów, rejestrów opóźniające i maszyn stanów, określenie prawidłowych sygnałów stymulacyjnych oraz określenie czy otrzymane wyniki są poprawne, może stanowić nie lada wyzwanie.

4.1 Język Verilog - konstrukcje symulacyjne

Do tej pory wykorzystywaliśmy instrukcje języka Verilog, które pozwalały na ustawienie wartości wyjść w zależności od zmiany stanu wejścia. Należały do nich komendy:

- **always @ (posedge clk)** – dla logiki synchronicznej,
- **assign** – dla logiki asynchronicznej

Do tworzenia środowisk testowych do symulacji innych modułów twórcy języka przewidzieli szereg specjalnych instrukcji, które pozwalają na wygodną i szybką pracę. Należy jednak zauważyć, że instrukcje te mogą być wykonywane jedynie przez narzędzia symulacyjne i nie ma możliwości ich stosowania w modułach implementowanych w docelowym układzie FPGA.

4.1.1 Środowisko testowe

Środowisko testowe (testbench) jest najczęściej realizowane poprzez zdefiniowanie modułu, który nie posiada żadnych portów wejścia i wyjścia. W module takim znajduje się instancja testowanego modułu (DUT) i opisane są jej połączenia z resztą bloków, które służą generacji sygnałów testowych i weryfikacji uzyskanej odpowiedzi. Przeanalizujmy jak wyglądałby kod opisujący takie środowisko dla bramki AND z rysunku 4.2:

Kod 4.1.1 — Środowisko testowe:

```
module testbench
(
);

wire a;
wire b;
wire c;

stimulate st_i
(
    .a(a),
    .b(b)
);

and_gate dut
(
    .a(a),
    .b(b),
    .c(c)
);

verify v_i
(
    .c(c)
);

endmodule
```

Moduł *stimulate* generuje sygnały, moduł *and_gate* jest testowaną bramką AND, a moduł *verify* jest odpowiedzialny za sprawdzanie stanu wyjścia. Symulacja w narzędziu ISE DS jest uruchamiana poprzez zaznaczenie *View->Simulation*, zaznaczeniu pliku, który jest środowiskiem testowym w oknie hierarchii projektu oraz kliknięciu na *Simulate Behavioral Model* w oknie *Processes*.

4.1.2 Generacja sekwencji testowych

Do opisu sekwencji wejściowej najlepiej użyć wyrażenia **initial begin end;**. Definiuje ono obszar czasowy, w którym kolejne instrukcje są wykonywane bezpośrednio w tym samym czasie, a do przejścia do innej skali czasu wykorzystuje się instrukcję opóźnienia **# N;** gdzie N określa ile nanosekund trwa opóźnienie. W ten sposób możliwe jest np. wygenerowanie szeregu kolejnych danych wejściowych dla testowanego modułu. Przykład pokazano w kodzie 4.1.2.

W środowisku **initial** istnieje również możliwość wykorzystania pętli **for** lub **while**, należy jednak pamiętać, że wewnątrz pętli musi znajdować się instrukcja opóźniająca, w innym przypadku cała pętla wykona się w tym samym czasie 1 ns i uzyskany wynik nie będzie zadowalający. Przykład użycia pętli **while** zaprezentowano w kodzie 4.1.3. Wykorzystano ją do generacji sygnału zegarowego. Warto zwrócić uwagę, że w większości przypadków będziemy

mieli do czynienia z tzw. logiką synchroniczną, której to działanie uzależnione jest od sygnału zegarowego. Zatem moduł tego typu będzie występował w prawie wszystkich testbench'ach.

Do zapisu wartości wykorzystuje się rejesty (nie można przypisywać wartości do ścieżek - **wire**), przy czym wewnątrz bloku **initial** wykorzystujemy do przypisania operator **=** zamiast **<=**.

Kod 4.1.2 — Generacja sekwencji wejściowej:

```
module stimulate
(
    output a,
    output b
);
reg r_a=1'b0;
reg r_b=1'b0;

initial
begin
#2; r_a=1'b0;r_b=1'b0;
#2; r_a=1'b0;r_b=1'b1;
#2; r_a=1'b1;r_b=1'b0;
#2; r_a=1'b1;r_b=1'b1;
end

assign a=r_a;
assign b=r_b;

endmodule
```

Ręczna definicja wszystkich wartości wejściowych jest możliwa jedynie dla prostych modułów. W innych przypadkach, zamiast podawać bezpośrednio wartości, lepiej doprowadzić do ich automatycznej generacji, przy wykorzystaniu instrukcji języka Verilog. Możliwe jest również wykorzystanie znanych z maszyny stanów konstrukcji **always** @ (**posedge clk**), przykładowo następująca sekwencja wygeneruje ten sam test co kod 4.1.2:

Kod 4.1.3 — Generacja sekwencji wejściowej:

```
module stimulate_auto
(
    output a,
    output b
);

reg clk=1'b0;
reg [1:0] cnt=2'b0;

initial
begin
    while(1)
    begin
        #1; clk=1'b0;
        #1; clk=1'b1;
    end
end

always @ (posedge clk)
begin
    cnt<=cnt+1;
end

assign a=cnt[1];
assign b=cnt[0];
endmodule
```

W powyższym przykładzie, wykorzystano dwa dodatkowe rejesty *clk* i *cnt*. Generowany zegar jest wykorzystywany do uruchomienia 2-bitowego licznika. Przypisanie odpowiednich bitów licznika do wyjść a i b, pozwala na uzyskanie każdej kombinacji na wyjściach testowych. W większości przypadków użycie drugiej metody jest bardziej efektywne (np. jeśli moduł miałby zamiast dwóch osiem wejść). Wtedy zapisanie wszystkich możliwości łatwo przekracza cierpliwość programisty.

4.1.3 Weryfikacja uzyskanych wyników

Do sprawdzania wyników, również najlepiej wykorzystać instrukcję **initial**. W odpowiednich chwilach czasu, należy sprawdzić wartości na wyjściach testowanego modułu. Do tego celu można wykorzystać instrukcję **if**. Sprawdzenie wartości dla bramki AND może odbywać się następująco:

Kod 4.1.4 — Weryfikacja sekwencji wyjściowej:

```
module verify
(
    input c
);

initial
begin
    #2 if(c!=1'b0) $stop;
    #2 if(c!=1'b0) $stop;
    #2 if(c!=1'b0) $stop;
    #2 if(c!=1'b1) $stop;
end

endmodule
```

W razie wykrycia błędu, do zatrzymania symulacji można wykorzystać komendę `$stop`. Do zakończenia symulacji w sytuacji, gdy wygenerowano już całą sekwencję testową służy komenda `$finish`. Opcjonalnie, podczas działania symulacji, istnieje możliwość wypisania tekstu do okna konsoli przy wykorzystaniu komendy `$display("tekst do wypisania");`

4.2 Model programowy

Przedstawione wyżej rozwiązania dobrze sprawdzają się jedynie w przypadku prostych modułów. Weryfikacja bardziej zaawansowanych bloków wymaga zastosowania innych metod. W przypadku, gdy testowany moduł realizuje zaawansowany algorytm przetwarzania danych (np. obliczanie przepływu optycznego dla strumienia video z kamery, segmentację obiektów pierwszoplanowych itp.), konieczne jest stworzenie tak zwanego modelu programowego za projektowanej architektury. Model programowy, to program napisany w dowolnym języku programowania i wykonywany na komputerze PC, którego działanie dokładnie oddaje działanie algorytmu realizowanego przez testowany moduł. Mówimy tutaj o dokładności co do jednego bitu (ang. *bit-accurate model*).

Model programowy pobiera dane z plików (np. obrazy, czy pakiety zarejestrowane z karty sieciowej) i realizuje na tych danych żądany algorytm. Rezultaty zapisuje do pliku wynikowego. Do konwersji obu typów plików wykorzystywane są konwertery, które umożliwiają zapisanie danych w postaci paczki bitów (np. jeśli obraz jest skompresowany umożliwiają zapisanie każdego piksela w postaci trzech bajtów). W ten sposób pliki takie mogą zostać łatwo wczytane do środowiska testowego w języku Verilog.

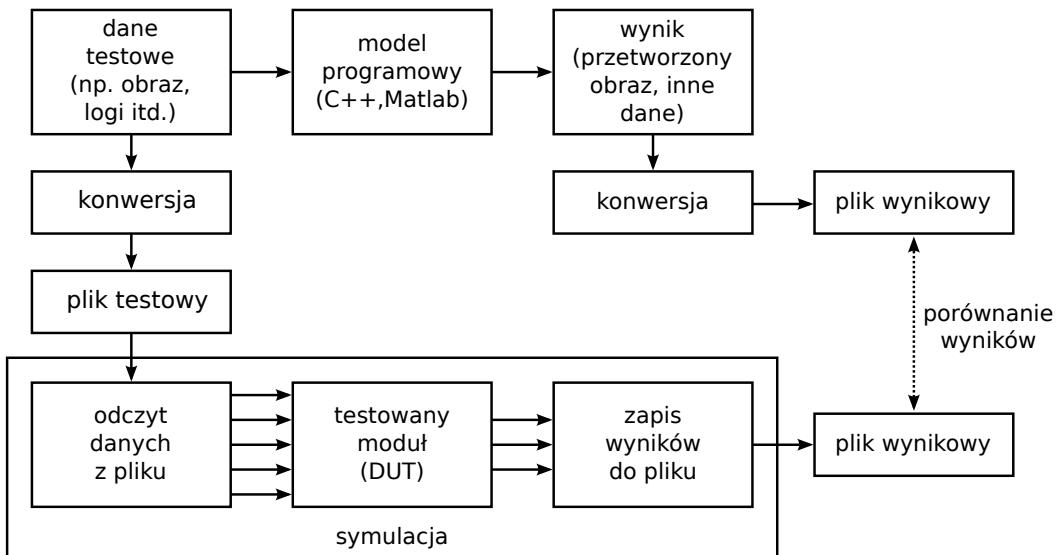


Figure 4.3: Model programowy

Wczytane wartości są następnie przetwarzane przez testowany moduł, a wyniki są zapisywane do pliku wynikowego. Porównanie wartości plików wyjściowych z modelem programowym i symulacji pozwala na sprawdzenie czy uzyskane wyniki są zgodne. Uzyskanie wyników niezgodnych świadczy o tym, że popełniono błąd albo podczas projektowania modułu sprzętowego albo podczas pisania modelu programowego. Warto również zwrócić na błędy popełniane w trakcie samego procesu symulacji np. wykorzystanie złych plików wejściowych, źle zrealizowany odczyt danych itp. Choć wydają się on dość "trywialne", doświadczenie uczy, że stanowią istotną przyczynę niepoprawnych wyników symulacji. Natomiast uzyskanie wyników zgodnych świadczy o tym, że z dużym prawdopodobieństwem moduł pracuje prawidłowo albo że popełniono te same błędy podczas projektowania modułu sprzętowego i modelu programowego. Dlatego w praktyce inżynierskiej stosuje się rozdzielenie obu zadań dla co najmniej dwóch programistów/projektantów.

Na koniec można jeszcze zauważyć, że najczęściej na układach reprogramowalnych implementuje się już istniejące algorytmy. Bądź to celem ich przyspieszenia, bądź uzyskania małych rozmiarów i możliwości użycia w urządzeniach typu wbudowanych (ang. *embedded*). W związku z tym, model programowy, w podstawowej wersji, istnieje już przed podjęciem prac nad implementacją sprzętową. Temat przejścia od algorytmu opisanego dla procesora ogólnego przeznaczenia np. w języku C lub Matlabie do poprawnego modelu programowego zostanie jeszcze poruszony w ramach niniejszego skryptu (por. rozdział 9).

4.2.1 Dostęp do plików na dysku komputera

Do odczytania wartości plików z dysku komputera, oraz zapisania wyników, stosowane są specjalne funkcje języka Verilog. Ich składnia jest bardzo podobna do znanych z języka C metod dostępu do plików przy pomocy funkcji `fopen`. W języku Verilog, nazwy tych funkcji są poprzedzone znakiem \$. Do przechowywania wskaźnika do pliku, wykorzystywana jest zmienna typu `integer`. Natomiast zapis i odczyt odbywa się do zmiennych typu rejestrowego `reg`.

Moduł, który umożliwia odczytanie czterech binarnych wartości z pliku oraz ich przypisanie do wyjść `a` i `b`, został przedstawiony w kodzie 4.2.1. Natomiast moduł, który umożliwia zapisanie wartości portu `c` do pliku wynikowego zaprezentowano w kodzie 4.2.2.

Kod 4.2.1 — Odczyt:

```
module load_file
(
    output a,
    output b
);

integer file;
reg [7:0]data;
reg [7:0]i;

initial
begin
    file=$fopen("ifile_path", "rb");
    for(i=0; i<4; i=i+1)
    begin
        #2;
        data=$fgetc(file);
    end
    $fclose(file);
end

assign a=data[0];
assign b=data[1];

endmodule
```

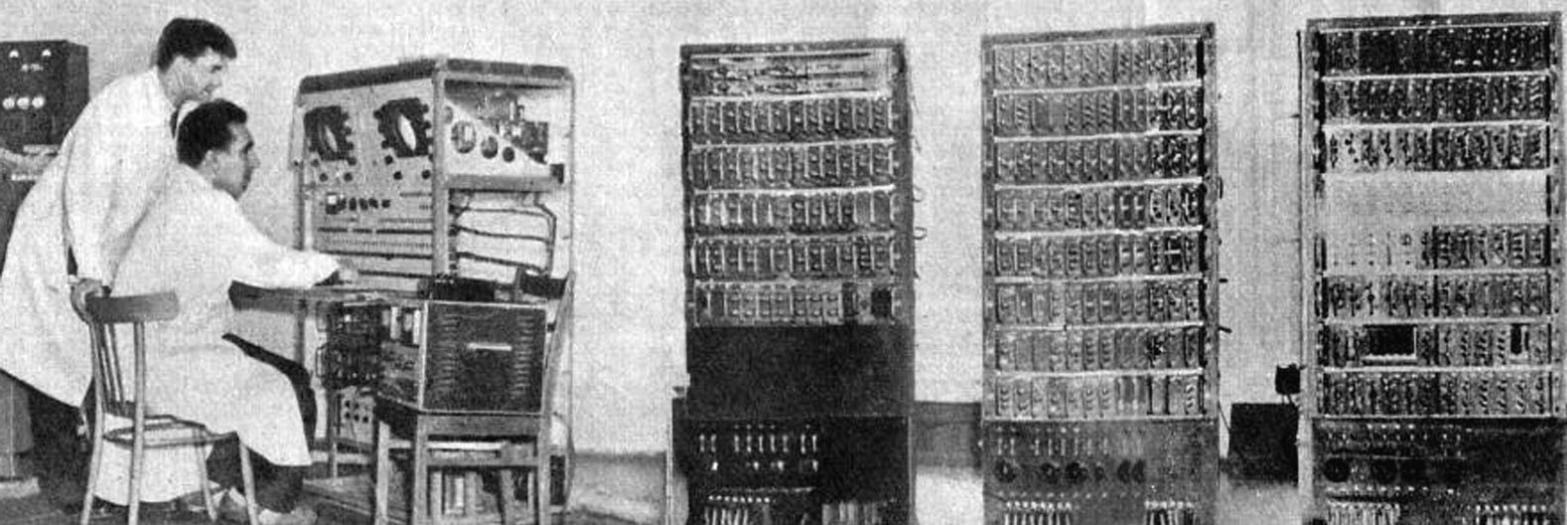
Kod 4.2.2 — Zapis:

```
module save_file
(
    input c
);

integer file;
reg [7:0]i;
wire [7:0]data={7'b0,c};

initial
begin
    file=$fopen("ofile_path", "wb");
    $fwrite(file, "To_jest_wynik:\n");
    for(i=0; i<4; i=i+1)
    begin
        #2;
        $fwrite(file, "%d\n", data);
    end
    $fclose(file);
end

endmodule
```



5 — Verilog i weryfikacja – praktyka

W rozdziale zamieszczono zadania, które stanowią podsumowanie rozdziałów 3 oraz 4. Układ taki podyktowany jest przekonaniem, że zaimplementowany moduł (nawet najprostszy) powinien od razu zostać poddany weryfikacji. Jest to również dobra praktyka inżynierska – testujemy (dokładnie) kolejne fragmenty większej aplikacji. W ten sposób zyskujemy pewność, że "budujemy" moduł z poprawnych komponentów. Warto zatem poświęcić więcej czasu na testy cząstkowe, niż po złożeniu całego systemu głosić się "dlaczego to nie działa".

5.1 Zadania do realizacji na zajęciach

5.1.1 Kaskada bramek AND

Zadanie 5.1 Bazując na przykładzie z rozdziału 3.1.15 proszę narysować i opisać w języku Verilog bramkę AND o parametryzowalnej liczbie wejść. Jej struktura powinna się opierać na odpowiednio połączonych dwuwejściowych bramkach AND. Stworzoną bramkę należy przetestować symulacyjnie. Proszę przyjąć, że używamy 8 wejść oraz sposobu przedstawionego w kodzie 4.1.3.

Podpowiedzi:

- utwórz nowy projekt w ISE oraz dodaj do niego nowy plik Verilog, w którym znajdzie się opis parametryzowej bramki AND.
- analizując wspomnianą implementację bramki OR, zrealizuj bramkę AND. Nie zapomnij o założonej liczbie wejść – 8. Uwaga. Nie trzeba realizować osobnego modułu pojedynczej (tj. dwuwejściowej) bramki AND. Wystarczy jak wewnątrz instrukcji generate wprost użyjemy składni: `assign c = a & b;`. Oczywiście pod `a,b,c` trzeba podstawić odpowiednie sygnały (patrz przykład z bramką OR).
- dokonaj syntezu modułu oraz sprawdź użycie zasobów (*Slice LUT's i LUT-FF pairs*). Czy mamy do czynienia z modelem synchronicznym, czy asynchronicznym ?
- w celu lepszego zrozumienia powiązania pomiędzy opisanym kodem, a faktycznymi zasobami układu FPGA oglądnijmy dwa schematy: RTL i technologiczny. Oba dostępne są w poleceniu Synthesize. Porównaj oba schematy. Zastanów się jakie zasoby FPGA wykorzystywane są do realizacji bramki AND.
- wygeneruj testbench. Można to zrobić na dwa sposoby: zupełnie "ręcznie" (jako moduł Verilog), albo wykorzystując kreator z pakietu ISE. W drugim przypadku dodajemy nowy

plik do projektu (*New Source*) i ustalamy go jako *Verilog Test Fixture*. Dobra praktyka to nazywanie testbench'a jako *tb_nazwa_modułu*. W ten sposób od razu można rozróżnić pliki do implementacji i testowania. Na następnym ekranie wybiera się powiązany plik. W tym przypadku wybór zbyt duży nie jest

- przeanalizuj wygenerowany testbench. Uzupełnij go analogicznie do kodu 4.1.3. Uwaga. Dla uproszczenia, w tym przypadku, nie realizujemy koncepcji trzech modułów tj. generatora sekwencji wejściowych, modułu testowanego oraz analizatora poprawności sygnałów wyjściowych. Ograniczmy się tylko do modułu nadziednego, w którym wygenerujemy sygnał testowy (wszystkie możliwe kombinacje sygnałów wejściowych) i instancji modułu testowanego (bramki AND). Poprawność sprawdzimy "na oko" – analizując wygenerowany przebieg.
- wykonaj symulację modułu. W tym celu przełącz widok z implementacji na symulację (*View->Simulation*) w górnej części zakładki *Design*. Następnie zaznacz testbench, sprawdź jego poprawność składniową (*Behavioral Check Syntax*) i uruchom symulację (*Simulate Behavioral Model*). Uwaga. Zawsze należy zwracać uwagę, jaki moduł wybrany jest w oknie *Hierarchy*.

Ponieważ jest to pierwsza styczność z programem ISim warto go omówić. Główne okno aplikacji podzielone jest na trzy części:

- Instance and Process Name (ew. zakładki Memory i Source Files),
- Simulation Objects for uut (uut - ang. unit under test),
- Przegieg sygnałów, ew. kod.

W pierwszej tj. *Instance and Process Name* uzyskujemy dostęp do hierarchii symulowanego modułu. W naszym przypadku mamy nadziedny moduł *tb_nazwa*, który składa się z *uut* oraz sekcji *initial* i *always*. Moduł *glbl* (globalny) nas nie interesuje.

Sam moduł *uut* składa się z ośmiu elementów – należało się tego spodziewać używając składni generate. Zaznaczając konkretny moduł w oknie *Objects* pojawiają się wszystkie związane z nim zmienne/sygnały: tj. wejścia, wyjścia, wejścia/wyjścia, sygnały wewnętrzne, stałe oraz zmienne. Różnice pomiędzy zmienną a sygnałem zostaną omówione w dalszej części kursu. Elementy z okna *Objects* możemy "przeciągać" na przebieg sygnałów. Warto zauważać, że domyślnie umieszczone są tam sygnały zdefiniowane w testbench'u (zegar oraz wejście i wyjście z modułu AND).

Dodamy teraz pomocniczy sygnał *chain*. Warto zauważać, że jego przebieg nie pojawi się. Aby tak się stało, należy symulację zrestartować. W tym celu wybieramy *Simulation->Restart* (lub ikona strzałki w lewo). Następnie symulację należy uruchomić. Tu mamy dwie opcje: Run All (uruchamia się cała symulacja i wykonywana jest do polecenia `$finish`) lub Run (dla zadanego czasu). Czas ustala się w oknie wyboru na pasku zadań. W rozważanym przypadku uruchomienie symulacji na 1 us jest wystarczające.

Przeanalizujemy teraz uzyskane wyniki. Pierwsze 100 ns to globalny reset określony w testbench'u. Następnie pojawia się sygnał zegara (należy korzystać z opcji zmiany skali przebiegów) oraz dane (np. *x* i *chain*). Należy zaobserwować jak zmieniają się te sygnały oraz w jakim przypadku uzyskujemy na wyjściu *y* wartość '1'.

Warto wspomnieć jeszcze o kilku funkcjonalnościach:

- ponowne uruchomienie symulacji (Re-lunch). Wymagane jest ono w przypadku dokonania zmian w plikach źródłowych. Uwaga. Zmiana interfejsu modułu i inne poważne ingerencje wymagają ponownego uruchomienia aplikacji ISim.
- zmiana formatu wyświetlanych liczb. Domyślnie wyświetlają się w postaci binarnej, co zwykle jest dość niewygodne. Aby to zmienić należy kliknąć prawym klawiszem na sygnale (z lewej strony okna) i wybrać *Radix*.
- na pasku występują dwa przyciski – *Previous Transition* i *Next Transition*. Powodują skok

do następnej zmiany sygnału. Przydają się w przypadku takim jak sygnał y , który zmienia się rzadko.

- inne funkcjonalności ISim (analiza plików, markery itp. zostaną zaprezentowane przy okazji kolejnych ćwiczeń).

Po przeprowadzeniu symulacji powinnyśmy mieć pewność, że poprawnie napisaliśmy bramkę AND.

[P] Zademonstruj prowadzącemu zajęcia uzyskane wyniki.

5.1.2 Licznik dzielący modulo N

Zadanie 5.2 Proszę opisać w języku Verilog moduł licznika, liczącego modulo N (parametr). Proszę również wykonać testbench do licznika i sprawdzić jego działanie dla co najmniej dwóch różnych wartości parametru N. ■

Zasadniczo należy się oprzeć na przedstawionym wcześniej module licznika (kod 3.1.16). Pewien problem stanowi określenie długości licznika – zmienna pomocnicza (*val*) i wyjście (*cnt*). Możemy w tym celu skorzystać z pomocniczego parametru WIDTH:

parameter WIDTH = \$clog2(N)

Warto zastanowić się, czy w tym przypadku należy opisać sygnał jako: [WIDTH:0] czy [WIDTH-1:0]? Ponieważ nie znamy "z góry" wartości WIDTH to zerowanie rejestru należy przeprowadzić po prostu jako przypisanie wartości 0. (tj. *val* = 0). Oczywiście kod licznika należy tak zmodyfikować aby zrealizować funkcjonalność "modulo N".

Tworzenie testbench'a jest względnie proste. Trzeba dodać generację sygnału zegara (jak w poprzednim ćwiczeniu). Warto także zmodyfikować instancję *uut*, tak aby móc było podawać parametr N modułu licznika. Przykład jak to zrobić przedstawiono poniżej:

Kod 5.1.1 — Przykład instancji parametryzowanego modułu:

```
nazwa_modulu # (
    .PARAM_1(wartosc_param_1)
)
nazwa_instancji_modulu
(
    .clk(clk),
    .ce(ce),
    .rst(rst),
    // itd
);
```

Opisany testbench należy przesymulować. Pewien problem stanowi *wire* związany ze wyjściem z modułu. Jego długość należy określić ręcznie, albo ew. za pomocą deklarowania parametrów w sposób zbliżony do zastosowanego w module licznika. Proszę przetestować licznika dla co najmniej dwóch wartości parametru N.

Symulator umożliwia pracę z kodem. Przejdz do zakładki *Source Files*. Wybierz plik z opisem modułu licznika. Otworzy się kod (w programie ISim). Tu drobna uwaga. Dość łatwo się pomylić, gdyż edytory w aplikacjach ISE i ISim są identyczne. Dobra jednak edytować kod tylko w ISE. Wybierz linijkę kodu i wstaw *breakpoint*. Może to być np. warunek logiczny na zerowanie licznika przy modulo N. Uruchom symulację i sprawdź funkcjonalność narzędzia tj.

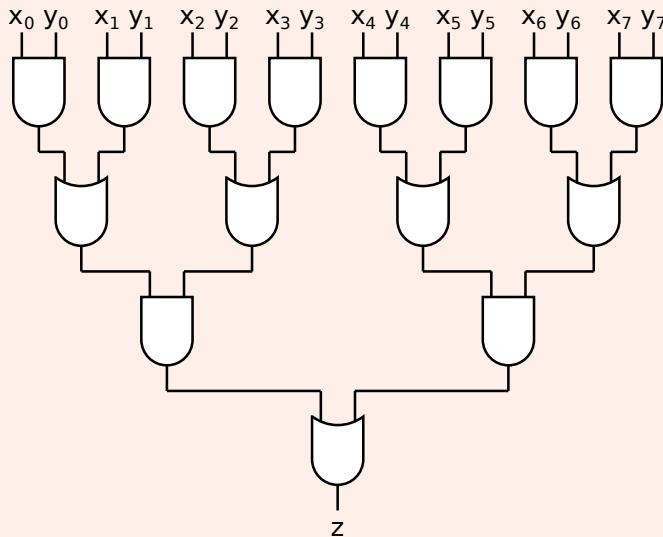
- możliwość "podglądania" wartości zmiennych – po ustawieniu na niej kurSORA,
- pracę krokową – F11.

Warto pamiętać o tej funkcjonalności, gdyż czasami bywa przydatna (np. analiza maszyn stanu).

[P] Zademonstruj prowadzącemu zajęcia uzyskane wyniki.

5.1.3 Złożony moduł logiczny

Zadanie 5.3 Proszę wykorzystać instrukcję generate i opisać przy pomocy języka Verilog następujący moduł:



Czy możliwe jest wykorzystanie tylko jednej instrukcji generate ?

Proszę do modułu dorobić testbech oraz samodzielnie "wygenerować" 8 wektorów testowych, które należy sprawdzić "ręcznie", a potem za ich pomocą przetestować stworzony moduł.

Podpowiedź. Rozwiążanie za pomocą jednej instrukcji generate wymaga trochę "gimastyki" indeksami. Operator modulo w Verilog jest taki sam jak w C/C++.

Uwaga. Warto podglądać schemat RTL modułu – dobry sposób na sprawdzenie poprawności implementacji.

5.2 Zadania do wykonania w domu

5.2.1 Linia opóźniająca

Zadanie 5.4 Jak powiedziano w rozdziale 3.1.13, szeregowo połączone rejesty mogą zostać wykorzystane do realizacji linii opóźniającej. Rozwiązanie takie zaprezentowano na rysunku 5.1.

Proszę zaprojektować moduł, który posiada dwa parametry:

- N – szerokość portów wejściowego i wyjściowego w bitach,
- DELAY – długość opóźnienia, które moduł powinien wprowadzać.

Wykorzystując instrukcję generate, proszę opisać moduł, który w zależności od wartości parametrów, będzie:

- dla DELAY = 0 – łączyć bezpośrednio wejście idata z wyjściem odata (assign)
- dla DELAY > 0 – generować DELAY bloków rejestrów o szerokości N, połączonych jak na rysunku 5.1

Pozostały interfejs proszę zrobić analogiczny jak modułu opóźniającego z rozdziału 3.1.13.

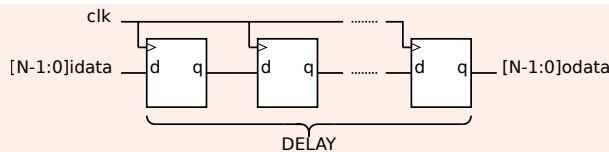


Figure 5.1: Linia opóźniająca z rejestrów

Dla modułu wygeneruj odpowiedni testbench. Sprawdź działanie dla dwóch przypadków: $DELAY = 0$ i $DELAY > 0$.

Zadanie należy zacząć od stworzenia modułu pojedynczego opóźnienia – analogicznie jak w rozdziale 3.1.13, przy czym trzeba "uzmiennić" szerokość danych. Następnie tworzymy moduły linii opóźniającej (np. *delay_line*). Powinien on mieć dwa parametry N oraz $DELAY$. Wewnątrz modułu konieczne jest zrealizowanie instrukcji wyboru z wykorzystaniem *generate* – por. kod 3.1.17. Dla przypadku $DELAY > 0$ należy wykorzystać kilka modułów *delay* – instrukcja *for*.

Potrzebną zmienną (instrukcja *genvar*) deklarujemy przed blokiem *generate*. Pewnym problemem jest wykonanie połączenia pomiędzy kolejnymi modułami *delay*. W tym celu wykorzystamy typ tablicowy w języku Verilog. Przykładowo składnia:

`wire [N-1:0] tdata [DELAY:0];`

oznacza, że połączenia ma szerokość N oraz takich połączeń jest $DELAY+1$. Wewnątrz *generate* możemy to wykorzystać w sposób następujący:

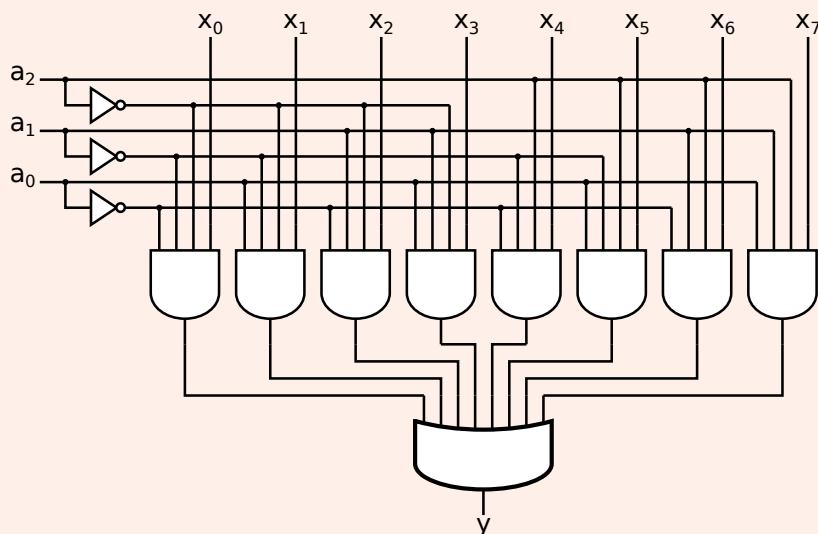
```
.idata(tdata[i]),
.odata(tdata[i+1])
```

Oczywiście trzeba jeszcze pamiętać o przypisaniu początkowym i końcowym tj. sygnału *idata* do *tdata* i *tdata* do *odata*.

Podczas testowania proszę sprawdzić, czy moduł wprowadza rzeczywiście takie opóźnienie jak deklarowane.

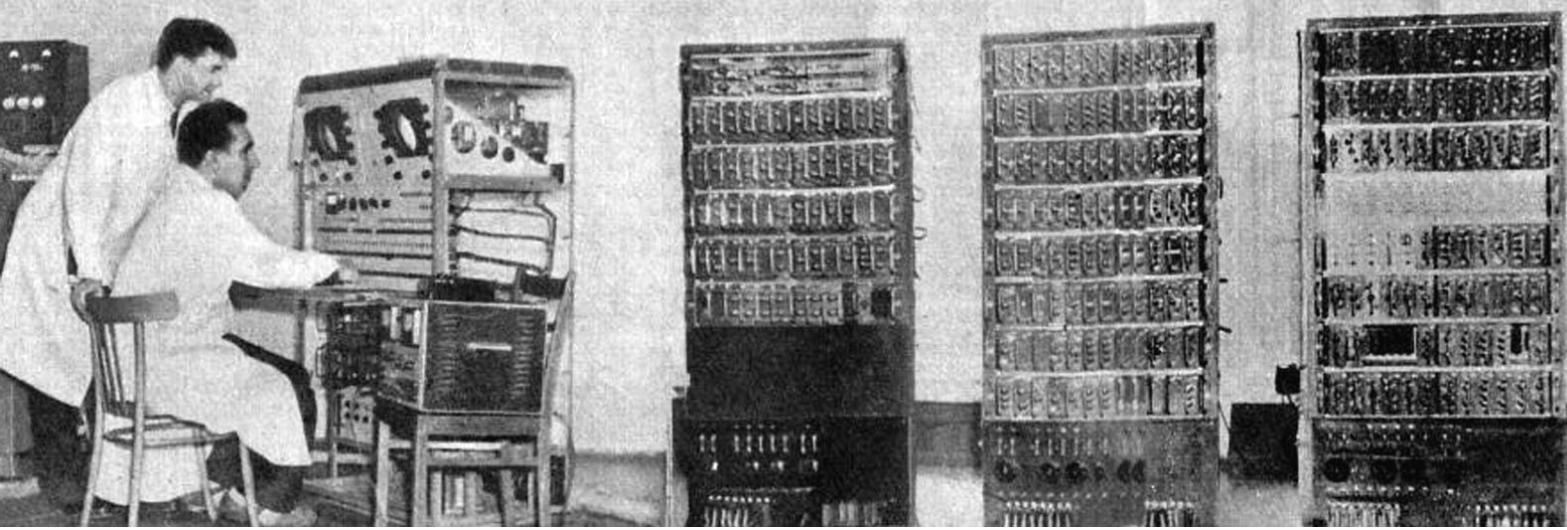
5.2.2 Tajemniczy moduł

Zadanie 5.5 Proszę opisać w języku Verilog następujący moduł:



Jaka jest funkcjonalność przedstawionego modułu ?
Tradycyjnie przetestuj tworzony moduł symulacyjnie.

Podpowiedź. Przed przystąpieniem do implementacji warto może się zastanowić nad odpowiedzią na zadane pytanie. Analiza funkcjonalności modułu może ułatwić jego implementację.



6 — Maszyny stanowe i zaawansowane testowanie

W rozdziale zamieszczone zostały zadania dotyczące realizacji maszyn stanowych w języku Verilog oraz zaawansowanego testowania m.in. odczytu i zapisu danych do pliku.

6.1 Zadania do realizacji na laboratorium

Zadanie 6.1 Proszę opisać przy pomocy języka Verilog następującą maszynę stanów:

Moduł powinien mieć trzy wejścia:

- clk – zegar,
 - rst – reset,
 - send – flaga oznaczająca że dane mają być wysłane,
 - [7 : 0]data – 8-bitowy sygnał danych
- oraz jedno wyjście:
- txd – wyjście danych

Maszyna powinna mieć 4 stany.

1. W pierwszym stanie należy sprawdzać, czy wejście *send* zmieniło swoją wartość od poprzedniego taktu zegara (ale tylko z 0 na 1 tj. zbocze narastające). Jeśli tak, to maszyna powinna przejść do stanu drugiego oraz wartość z wejścia *data* zapamiętana w wewnętrznym rejestrze (trzeba go stworzyć). W ten sposób za poprawne uznawane będą tylko te dane, które pojawią się wraz z narastającym zboczem zegara.
2. W stanie drugim, na wyjście *txd* powinna zostać podana wartość '1' oraz powinno nastąpić przejście do stanu trzeciego.
3. W trzecim stanie, na wyjście *txd* powinny być przesyłane kolejne bity portu *data* z utworzonego rejestrzu, od najmłodszego do najstarszego. Po przesłaniu wszystkich bitów należy przejść do stanu czwartego.
4. W ostatnim stanie, na wyjście *txd* powinna być podana wartość '0' oraz maszyna powinna wrócić do pierwszego stanu.

Jaką funkcjonalność i jaki protokół realizuje zaproponowana maszyna stanów ?

Wskazówki:

- zacząć należy oczywiście od nowego projektu w ISE.
- do realizacji detekcji narastającego zbocza sygnału *send* warto zapamiętać jego poprzednią

wartość.

- poza tym trzeba się wzorować na przykładzie 3.1.19.
- wysyłanie 8 bitów w ramach jednego stanu można rozwiązać dodaniem licznika oraz instrukcją `if else`.

Zadanie 6.2 W zadaniu 6.1 zaprojektowano maszynę stanów umożliwiającą serializację danych (tj. zamianę za postaci równoległej 8-bitowej na szeregową).

Proszę następnie utworzyć środowisko testowe, które pozwoli w sposób automatyczny przetestować zaprojektowany moduł. Dane wejściowe powinny być odczytane z pliku binarnego, a rezultaty zapisane do pliku binarnego. Plik wejściowy należy utworzyć samodzielnie i wypełnić go 16 losowymi bajtami (tj. znakami ASCII np. "alamapsaidwakoty", a nie '0' i '1').

Proszę napisać skrypt w pakiecie Matlab lub program w C/C++, który dokona serializacji danych z pliku wejściowego i zapisze dane do pliku wyjściowego. Będzie to nasz programowy model referencyjny (por. rozdział 4.2).

W przypadku wykrycia niezgodności obu plików wynikowych, proszę je odnotować oraz zmodyfikować moduł maszyny stanów w ten sposób, aby usunąćauważone błędy. ■

Wskazówki:

- zaczynamy od wygenerowania nadrzednego testbench'a (*Verilog Text Fixture*), w tym przypadku będziemy realizować koncepcję testowania opisaną w rozdziale 4.1.1 – z trzema odrębnymi modułami,
- następnie tworzymy moduł do odczytu danych z pliku. Ma on mieć dwa wyjścia: *data* (8 bitów) i *send* (1 bit). Powinien być zbliżony do kodu 4.2.1. Wewnątrz pętli `for` odczytujemy kolejne bajty z pliku. Ustawiamy też sygnał *send*.
- Uwaga. Chcemy, aby sygnał *send* pojawiał się tylko na jeden takt zegara, kiedy dane są poprawne. Zakładamy przy tym, że takt zegara to 2 ns (#). Ponadto chcemy aby dane było odczytywane co 12 taktów zegara. Liczba dwanaście wynika ze specyfiki maszyny stanów. Wysyłanie danych trwa 10 taktów zegara – 8 bitów danych + bit startu i stopu. Jeden takt trwa przejście przez stan początkowy (wtedy nic nie jest wysyłane). Pozostały jeden tak ustalamy dla bezpieczeństwa. Zatem przy odczytcie ustawiamy flagę *send* na '1', opóźniamy o jeden takt, ustawiamy na '0' i opóźniamy o pozostałe 11 taktów.
- Uwaga. Proszę zwrócić uwagę, że wewnątrz modułu posługujemy się rejestrami dla danych i flagi *send*, a do wyjść wartości przypisujemy z wykorzystaniem instrukcji *assign*.
- przy realizacji modułu do zapisu wzorujemy się na kodzie 4.2.2. Ustalamy, że zapisujemy co najmniej $16 * 12$ bitów. Pomiędzy kolejnymi zapisami wprowadzamy opóźnienie 2 ns. (takt zegara). Uwaga. Zapisujemy bity - tj. '0' lub '1'.
- następnym krokiem jest połączenie modułów w ramach "głównego" testbench'a oraz dodanie generacji sygnału zegarowego (takt 2 ns) – podobnie jak w poprzednich ćwiczeniach. Uwaga. Moduły łączy się za pomocą wire'ów, a nie rejestrów.
- uruchamiamy symulację. Sprawdzamy, czy "na oko" wszystko jest dobrze. Proszę pamiętać o odwrotnej kolejności w jakiej wysyłane są dane (a przynajmniej powinny). Proszę też sprawdzić, czy dane zapisują się do pliku.
- następnie piszemy model programowy. Jeśli korzystamy z Matlab'a to przypomnienie funkcji:
 - `fopen` – otwieranie pliku,
 - `fscanf` – czytanie z pliku (czytamy całe 16 bajtów),
 - `dec2bin` – zamiana liczby na postać binarną. Aby ze znaku otrzymać liczbę wystarczy wykorzystać składnię `double` (znak).

- `fliplr` – odwracanie wektora przydatne z uwagi na odwróconą kolejność danych,
- `fclose` – zamknięcie pliku.

Proszę nie zapomnieć o dodaniu bitu startu ('1') i bitu stopu ('0'). Proszę ewentualnie dodać bity '0', tak aby uzyskać zgodność z rezultatami z modułu sprzętowego (co najmniej dwa zera aby pojedyncza dana miała 12 bitów). Sprawdzenia można dokonać w "Notatniku". Obie sekwencje, "ustawione jedna pod drugą", powinny być identyczne.

6.2 Zadania do realizacji w domu

Zadanie 6.3 Proszę pobrać plik `or_gate.v` z modułem 10 wejściowej bramki OR. Proszę stworzyć nowy projekt w ISE DS, dodać do niego pobrany plik (umieścić w folderze) i utworzyć środowisko testowe, które w automatyczny sposób umożliwi sprawdzenie, czy dostarczona bramka działa prawidłowo. Koncepcja podobna do zadania 5.1.

W przypadku wykrycia błędów, proszę zaprojektować środowisko testowe, które w automatyczny sposób zapisze błędy w pliku (log). Proszę odnotować, które kombinacje wejść skutkują uzyskaniem niepoprawnych wyników. Czy jesteś w stanie zgadnąć dlaczego wyniki są błędne?

6.3 Zadania dodatkowe

Zadanie 6.4 Proszę samodzielnie zaprojektować maszynę stanową do obioru danych wysyłanych z wykorzystaniem maszyny z zadania 6.1. Moduł powinien mieć trzy wejścia: `clk`, `rst` i `rx` (dane) oraz dwa wyjścia `data` i `received` (flaga ustawiana po otrzymaniu paczki danych). Proszę również stworzyć moduł, który będzie zapisywał otrzymywane dane do pliku (w postaci ciągu znaków ASCII) i dodać go do testbench'a. Działanie całego modułu należy sprawdzić symulacyjnie.

Uwagi:

- maszyna stanów powinna być dość zbliżona (wręcz symetryczna) do realizującej wysyłanie,
- trzeba zaproponować mechanizm wykrywania bitu startu,
- flaga `received` powinna pojawić się po otrzymaniu 8 bitów danych na jeden takt zegara,
- w module do zapisu należy wykorzystać tę flagę,
- poprawne wykonanie – plik wejściowy i wyjściowy identyczne.



7 — Operacje arytmetyczne

Główną różnicą pomiędzy wykonywaniem obliczeń w systemach procesorowych, w porównaniu do układów FPGA jest to, że o ile w tych pierwszych istnieją z góry ustalone rozmiary i typy danych (8-bitowe, 16-bitowe, 32-bitowe oraz 64-bitowe – char, int, float, double), o tyle w układach FPGA, projektant ma możliwość wykorzystać elementy obliczeniowe pracujące na danych o dowolnym rozmiarze oraz określić czy będą to liczby stało czy zmiennoprzecinkowe. W przypadku procesora, dodanie dwóch wartości 17-bitowych wymaga zawsze wykorzystania typu 32-bitowego. Natomiast w układzie reprogramowalnym, wykorzystany zostanie sumator dokładnie 17-bitowy. Zaoszczędzone w ten sposób zasoby układu są wystarczające do realizacji np. 8-bitowego sumatora, które może być użyty w innym miejscu układu.

Zaczniemy zatem od przypomnienia bitowego formatu zapisu liczb całkowitych i stało-przecinkowych oraz przyjrzymy się jak wykonanie poszczególnych operacji wpływa na końcowy format uzyskanych wyników.

7.1 Format zapisu liczb

7.1.1 Całkowitoliczbowy bez znaku

Do zapisu dodatnich liczb całkowitych bez znaku wykorzystywany jest format, w którym na poszczególnych bitach od najstarszego do najmłodszego zapisuje się bezpośrednio zakodowaną wartość, zgodnie ze wzorem:

$$w = \sum_{i=0}^{N_c-1} c_i 2^i \quad (7.1)$$

Dwa przykładowe wektory, dla przypadku 7- i 5-bitowego słowa przedstawiono poniżej. Proszę zwrócić uwagę na maksymalny zakres wartości, który można opisać przy pomocy tych wektorów.

$$\mathbf{A} = \left\| \begin{array}{c|c|c|c|c|c|c|c} c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \end{array} \right\|_{N_c=7}$$

$$\mathbf{B} = \left\| \begin{array}{c|c|c|c|c|c} c_4 & c_3 & c_2 & c_1 & c_0 \end{array} \right\|_{N_c=5}$$

wartość minimalna: 0
wartość maksymalna: 127

wartość minimalna: 0
wartość maksymalna: 31

Wykonanie operacji arytmetycznych na wektorach A i B prowadzi do uzyskania wyników, których format jest inny niż wektorów A i B. W tabeli 7.1.1 pokazano, jakie maksymalne i minimalne wartości mogą przyjmować otrzymane wyniki. Przedstawiono również, jaki musi być format zapisu rezultatów, aby, bez straty żadnej informacji, udało się w nim przechowywać wynik operacji. Powyższa uwaga nie dotyczy operacji dzielenia. W tym przypadku trudno mówić o wyniki bez straty informacji, np. dla ułamka 1/3. Temat zapisu liczb ułamkowych zostanie przedstawiony w rozdziałach 7.1.3 i ??.

operacja	wartość		format
	min.	maks.	
Y=A+B	0	158	$c_7 \mid c_6 \mid c_5 \mid c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0$ $N_{Y_c} = \max(N_{Ac}, N_{Bc}) + 1$
Y=A-B	-31	127	$\ z \ c_6 \mid c_5 \mid c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0$ $N_{Y_z} = 1 \quad N_{Y_c} = \max(N_{Ac}, N_{Bc})$
Y=A*B	0	3937	$c_{11} \mid c_{10} \mid c_9 \mid c_8 \mid c_7 \mid c_6 \mid c_5 \mid c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0$ $N_{Y_c} = N_{Ac} + N_{Bc}$
Y=A/B	0	127	$c_6 \mid c_5 \mid c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0 \mid u_4 \mid u_3 \mid u_2 \mid u_1 \mid u_0$ $N_{Y_c} = N_{Ac} \quad N_{Y_u} = N_{Bc}$

Należy zauważyć, że wynik dodawania i mnożenia dwóch liczb całkowitych jest zawsze liczbą całkowitą nieujemną. Natomiast wynik odejmowania, może być ujemny, wymaga więc zastosowania formatu całkowitego ze znakiem (por. rozdział 7.1.2). Wynik dzielenia może być wartością ułamkową, do jego zapisu wymagane jest zastosowanie formatu stałoprzecinkowego (pewną liczbę bitów należy przeznaczyć na część całkowitą (N_{Y_c} , a pevną na ułamkową N_{Y_u})).

7.1.2 Całkowitoliczbowy ze znakiem

Do zapisu liczb całkowitych ujemnych, wykorzystywany jest format całkowitoliczbowy ze znakiem. Najstarszy bit w słowie jest bitem znaku, który określa czy dana liczba jest dodatnia czy ujemna. W kodzie uzupełnień do dwóch, wartość liczby jest zapisywana przy pomocy wzoru:

$$w = -z 2^{N_c} + \sum_{i=0}^{N_c-1} c_i 2^i \quad (7.2)$$

Dwa przykładowe wektory, dla przypadku 6- i 5-bitowego słowa przedstawiono poniżej. Proszę zwrócić uwagę na maksymalny zakres wartości, który można opisać przy pomocy tych wektorów.

$$A = \| z \| c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0 \|_{N_c=5}$$

wartość minimalna: -32
wartość maksymalna: 31

$$B = \| z \| c_3 \mid c_2 \mid c_1 \mid c_0 \|_{N_c=4}$$

wartość minimalna: -16
wartość maksymalna: 15

Wykonywanie operacji arytmetycznych na liczbach całkowitych ze znakiem jest nieco bardziej skomplikowane niż w przypadku bez znaku. Przykłady oraz skrajne wartości możliwych do uzyskania wyników, zostały przedstawione poniżej.

operacja	wartość		format
	min.	maks.	
Y=A+B	-48	46	$\begin{array}{ c c c c c c c c } \hline z & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline N_{Yz}=1 & N_{Yc} = \max(N_{Ac}, N_{Bc}) + 1 & & & & & \\ \end{array}$
Y=A-B	-47	47	$\begin{array}{ c c c c c c c c } \hline z & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline N_{Yz}=1 & N_{Yc} = \max(N_{Ac}, N_{Bc}) & & & & & \\ \end{array}$
Y=A*B	-496	512	$\begin{array}{ c c c c c c c c c c c } \hline z & c_{10} & c_9 & c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline N_{Yz}=1 & N_{Yc} = N_{Ac} + N_{Bc} + 1 & & & & & & & & & & \\ \end{array}$
Y=A/B	-32	32	$\begin{array}{ c c c c c c c c c c c c } \hline z & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & u_3 & u_2 & u_1 & u_0 \\ \hline N_{Yz}=1 & N_{Yc} = N_{Ac} + 1 & N_{Yu} = N_{Bc} & & & & & & & & & \\ \end{array}$

W tym przypadku jedynie wykonanie operacji dzielenia powoduje, że uzyskany wynik musi zostać zapisany w formacie stałoprzecinkowym ze znakiem.

7.1.3 Stałoprzecinkowy bez znaku

Format stałoprzecinkowy bez znaku umożliwia zapisanie dodatnich liczb ułamkowych (rzeczywistych). Słówko jest podzielone na dwie części i składa się z N_c bitów, które opisują część całkowitą oraz N_u bitów, na których zapisana jest część ułamkowa. Liczba jest opisana równaniem:

$$w = \frac{\sum_{i=0}^{N_c-1} c_i 2^{i+N_u} + \sum_{i=0}^{N_u-1} u_i 2^i}{2^{N_u}} \quad (7.3)$$

Przy czym im więcej bitów zostanie przeznaczonych na część ułamkową, tym większa jest rozdzielcość części ułamkowej (inaczej mówiąc dokładność reprezentacji).

$$A = \left\| \begin{array}{|c|c|c|c|c|c|} \hline c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline N_c=5 & & & & \\ \end{array} \right\| \left\| \begin{array}{|c|c|} \hline u_1 & u_0 \\ \hline N_u=2 & \\ \end{array} \right\| \quad B = \left\| \begin{array}{|c|c|c|c|c|c|} \hline c_3 & c_2 & c_1 & c_0 \\ \hline N_c=4 & & & \\ \end{array} \right\| \left\| \begin{array}{|c|c|c|} \hline u_2 & u_1 & u_0 \\ \hline N_u=3 & & \\ \end{array} \right\|$$

wartość minimalna: 0

wartość maksymalna: 31,75

rozdzielcość części ułamkowej: 0,25

wartość minimalna: 0

wartość maksymalna: 15,875

rozdzielcość części ułamkowej: 0,125

Na wektorach A i B można wykonywać operacje arytmetyczne. Poszczególne wyniki mają następujące zakresy:

operacja	wartość		format										
	min.	maks.											
Y=A+B	0	19,625	c_4	c_3	c_2	c_1	c_0	u_2	u_1	u_0			
			$N_{Yc} = \max(N_{Ac}, N_{Bc}) + 1$					$N_{Yu} = \max(N_{Au}, N_{Bu})$					
Y=A-B	-3,875	15,75	z	c_3	c_2	c_1	c_0	u_2	u_1	u_0			
			$N_{Yz} = 1$	$N_{Yc} = \max(N_{Ac}, N_{Bc}) + 1$				$N_{Yu} = \max(N_{Au}, N_{Bu})$					
Y=A*B	0	61,03125	c_5	c_4	c_3	c_2	c_1	c_0	u_4	u_3	u_2	u_1	u_0
			$N_{Yc} = N_{Ac} + N_{Bc}$					$N_{Yu} = N_{Au} + N_{Bu}$					
Y=A/B	0	126	c_6	c_5	c_4	c_3	c_2	c_1	c_0	u_3	u_2	u_1	u_0
			$N_{Yc} = N_{Ac} + N_{Bu} + 1$					$N_{Yu} = N_{Au} + N_{Bc}$					

Jedną z najważniejszych cech zapisu stałoprzecinkowego jest to, że do wykonywania obliczeń można wykorzystać standardowe elementy obliczeniowe pracujące na logice binarnej. To znaczy, mnożarka wykonująca mnożenie dwóch liczb całkowitych może zostać wykorzystana do wymnożenia dwóch liczb stałoprzecinkowych. Projektant jest natomiast odpowiedzialny za to, aby poprawnie ustalić szerokość części ułamkowej i odpowiednio zinterpretować wynik w formacie stałoprzecinkowym. Innymi słowy mówiąc, miejsce przecinka w tym formacie jest sprawą czysto umowną i nie ma wpływu na sposób prowadzenia obliczeń.

UWAGA! Podczas dodawania lub odejmowania dwóch liczb stałoprzecinkowych, należy zadbać o to, żeby szerokość części ułamkowej w obu wektorach była taka sama. W przeciwnym razie wynik nie będzie poprawny. Wyrównanie części ułamkowej można uzyskać poprzez dodanie bitów (zer) do wektora z mniejszą częścią ułamkową. Możliwe jest również odcięcie najmniej znaczących bitów części ułamkowej, co powoduje zmniejszenie rozdzielczości.

7.1.4 Stałoprzecinkowy ze znakiem

Format stałoprzecinkowy ze znakiem jest wykorzystywany do zapisywania dodatnich i ujemnych liczb ułamkowych i całkowitych. Słowo składa się ze znaku (najstarszy bit), N_c bitów części całkowitej oraz N_u bitów części ułamkowej. Wartość jest zakodowana jako:

$$w = \frac{-z2^{N_c+N_u} + \sum_{i=0}^{N_c-1} c_i 2^{i+N_u} + \sum_{i=0}^{N_u-1} u_i 2^i}{2^{N_u}} \quad (7.4)$$

Poniżej przedstawiono sposób kodowania dla dwóch wektorów:

$$A = \begin{array}{c} \parallel z \parallel c_3 \mid c_2 \mid c_1 \mid c_0 \parallel u_1 \mid u_0 \parallel \\ N_z=1 \quad N_c=4 \quad N_u=2 \end{array} \quad B = \begin{array}{c} \parallel z \parallel c_2 \mid c_1 \mid c_0 \parallel u_2 \mid u_1 \mid u_0 \parallel \\ N_z=1 \quad N_c=3 \quad N_u=3 \end{array}$$

wartość minimalna: -16,0

wartość maksymalna: 15,75

rozdzielcość części ułamkowej: 0,25

wartość minimalna: -8,0

wartość maksymalna: 7,875

rozdzielcość części ułamkowej: 0,125

Wynik podstawowych operacji na liczbach stałoprzecinkowych ze znakiem jest zawsze liczbą stałoprzecinkową ze znakiem. Formaty poszczególnych wyników są następujące:

operacja	wartość		format							
	min.	maks.	z	c_3	c_2	c_1	c_0	u_2	u_1	u_0
$Y=A+B$	-10,0	9.625	$N_{Yz}=1$	$N_{Yc} = \max(N_{Ac}, N_{Bc}) + 1$				$N_{Yu} = \max(N_{Au}, N_{Bu})$		
$Y=A-B$	-9,875	9,75	$N_{Yz}=1$	$N_{Yc} = \max(N_{Ac}, N_{Bc}) + 1$				$N_{Yu} = \max(N_{Au}, N_{Bu})$		
$Y=A*B$	-15,5	16,0	$N_{Yz}=1$	$N_{Yc} = N_{Ac} + N_{Bc} + 1$				$N_{Yu} = N_{Au} + N_{Bu}$		
$Y=A/B$	-64	64	$N_{Yz}=1$	$N_{Yc} = N_{Ac} + N_{Bu} + 1$				$N_{Yu} = N_{Au} + N_{Bc} + 1$		

7.2 Zmienna długość słowa

Czy nie lepiej dla uproszczenia przyjąć stałą maksymalną długość słowa i konsekwentnie jej używać, zamiast utrudniać sobie życie projektowaniem systemu? Tak właśnie postępuje się w procesorach ogólnego przeznaczenia, gdzie liczba typów jest praktycznie ograniczona do liczb 8, 16, 32 i 64 bitowych. Może i lepiej oraz wygodniej, ale jeśli prowadzimy obliczenia i wymagane jest 17 bitów, to nie ma możliwości wykorzystania 16 bitowego sumatora, a wykorzystanie sumatora 32 bitowego zużywa zasoby, które mogą być np. wykorzystane do realizacji sumacji 8 bitowej dla innej zmiennej.

Warto również w tym miejscu wspomnieć o metodologii przechodzenia z zapisu zmiennoprzecinkowego na stałoprzecinkowy. Założmy, że mamy dany prosty algorytm operujący na liczbach typu *double*. Oprócz bardzo specyficznych obliczeń, taka reprezentacja jest zwykle zupełnie satysfakcyjująca i dokładna. Chcemy teraz przejść na format stałoprzecinkowy. Musimy podjąć dwie decyzje. Po pierwsze trzeba ustalić ile bitów przeznaczmy na część całkowitą. Decyzja ta zwykle jest dość prosta, musimy tylko oszacować maksymalne i minimalne wartości jakie mogą wystąpić na każdym etapie obliczeń.

Po drugie, trzeba określić liczbę bitów przeznaczonych na część ułamkową. To już jest trudniejsze, gdyż precyzji *double* w ten sposób nigdy nie osiągniemy. Musimy się pogodzić z pewną utratą dokładności. Zatem zwykle tworzy się model stałoprzecinkowy algorytmu i empirycznie sprawdza, jak precyza wpływa na wynik. Albo inaczej, jak różni się wynik "dokładny" (precyza *double*) i analizowany. Oczywiście dla różnych danych wejściowych i różnej liczby bitów przeznaczonej na część ułamkową. Na podstawie wyników podejmuje się decyzję o precyzyji, która zwykle jest kompromisem pomiędzy dokładnością, a zasięgiem zasobów.

Zasygnalizowana metodologia zostanie zademonstrowana praktycznie w ramach niniejszego kursu.

7.3 Latencja

Oprócz formatu słowa, z którym są wykonywane obliczenia, drugim najważniejszym parametrem opisującym elementy wykonujące operacje arytmetyczne jest latencja. **Latencja** to liczba taktów zegara, która upływa od pojawienia się wartości na wejściu do ustalenia się prawidłowego rezultatu na wyjściu.

W większości przypadków projektowana logika składa się z części synchronicznej (tj. sterowanej sygnałem zegarowym – np. przerzutniki) oraz części asynchronicznej (np. element LUT, multiplekser). Schematycznie zostało to pokazane na rysunku 7.1.

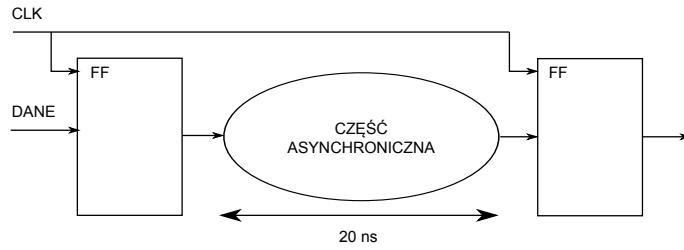


Figure 7.1: Logina synchroniczna i asynchroniczna

Jeśli opóźnienie części asynchronicznej na które składa się opóźnienie wprowadzane przez logikę (propagacja sygnału przez elementy LUT, multipleksery itp.) oraz wprowadzane przez połączenia (tj. ścieżki łączące odpowiednie elementy logiczne) wynosi 20 ns to maksymalna częstotliwość pracy wynosi 50 MHz. Latencja takiego rozwiązania równa jest 1. Zwiększenie częstotliwości można uzyskać poprzez podzielenie części asynchronicznej i wprowadzeniu dodatkowego przerzutnika. Przykład zaprezentowano na rysunku 7.2.

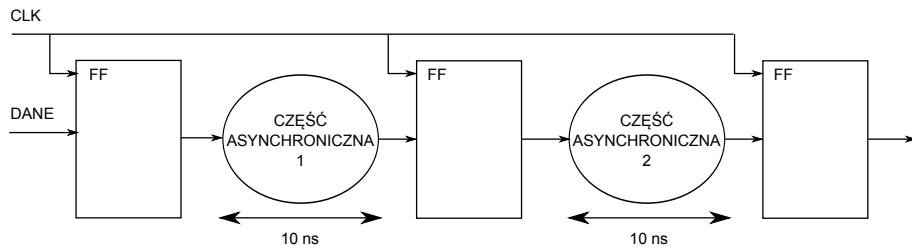


Figure 7.2: Logina synchroniczna i asynchroniczna – podział

Podział logiki pozwolił na redukcję opóźnienia i tym samym zwiększenie maksymalnej częstotliwości pracy. Odbyło się to kosztem zasobów logicznych (dodatkowy przerzutnik) oraz zwiększeniem latencji do 2.

W układach cyfrowych podzielenie poszczególnych operacji na etapy (podobnie jak w przypadku dodawania dwóch liczb metodą słupkową na kartce) umożliwia zaprojektowanie prostszych struktur. Wymagane jest jednak kilka taktów zegara na wykonanie tych obliczeń.

Na rysunku 7.3 pokazano jak wyglądają przebiegi czasowe dla sumatorów o latencji równej 0, 1 i 2 taktom zegara. W pierwszym przypadku, gdy latencja wynosi 0, zegar nie jest wykorzystywany. Zmiana wartości na wejściu bezpośrednio wpływa na zmianę wartości wejścia. Oczywiście nie oznacza to, że dzieje się to "w nieskończonym krótkim czasie". Logika asynchroniczna ma swoje czasy propagacji (czas przejścia sygnału przez element logiczny tj. np. LUT i zasoby połączeniowe). W przypadku, gdy latencja wynosi 1, wartość na wejściu zmienia się dopiero przy następnym narastającym zboczu zegara. W przypadku gdy latencja wynosi 2, wartość na wyjściu jest poprawna dopiero przy wystąpieniu drugiego narastającego zbocza zegara. Przy czym należy zauważać, że w przypadku latencji, opóźnione są wartości wejścia od wyjścia, natomiast poszczególne wyniki nie są od siebie opóźnione.

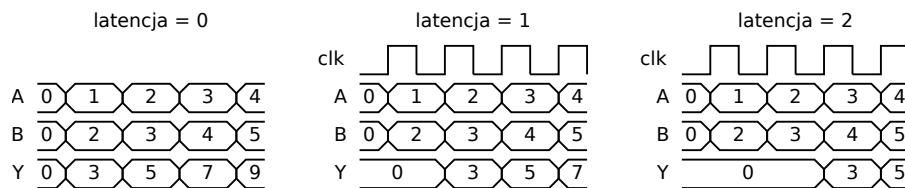


Figure 7.3: Wykonywanie obliczeń przez sumator z latencją równą 0, 1 i 2 taktom zegara

W przypadku wykonywania obliczeń, latencja wprowadza opóźnienie, co powoduje, że jeśli układ cyfrowy ma wykonać złożone obliczenia, to konieczne jest takie jego zaprojektowanie, aby poszczególne wyniki występuły w tej samej chwili czasu. Rozważmy przykład wykonywania operacji danej równaniem:

$$Y = (A + B) * C \quad (7.5)$$

Wartości z portów A, B i C podawane w kolejnych taktach zegara powinny zostać odpowiednio zsumowane i wymnożone.

Jeśli wszystkie elementy obliczeniowe pracują z latencją równą 0, to do wykonania obliczeń można wykorzystać schemat przedstawiony na rysunku 7.4 a). Gdyby zastosować ten schemat z elementami o latencji większej od zera, wynik sumowania ($A+B$) zostałby pomnożony przez wartość C podaną w kolejnym taktie zegara. Zatem rezultat operacji byłby niepoprawny. W związku z tym, konieczne jest zastosowanie schematu z rysunku 7.4 b), dodatkowy blok oznaczony jako "D" stanowi opóźnienie (ang. *delay*). Aby możliwe było poprawne wykonanie obliczeń, powinien on opóźnić wartość z portu C o liczbę taktów zegara równą latencji sumatora.

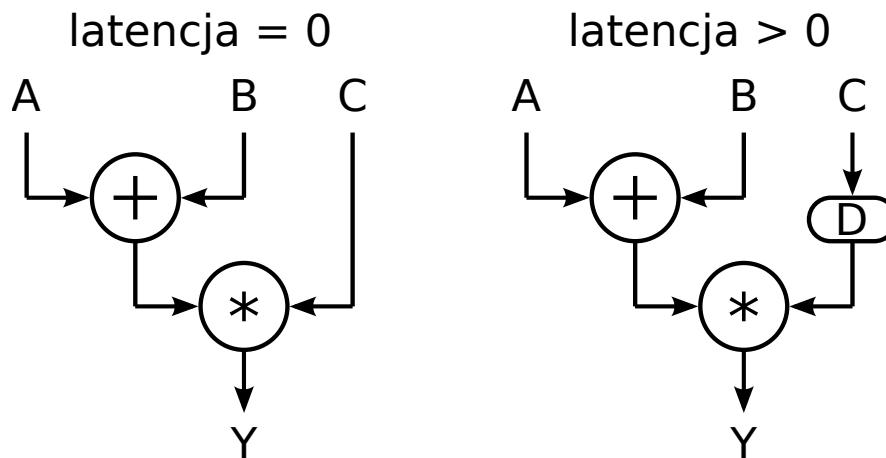


Figure 7.4: Wykonanie operacji $Y=(A+B)*C$ przy elementach o różnych latencjach

Można się więc zastanowić dlaczego do wykonywania wszystkich obliczeń nie wykorzystywać elementów o latencji równej 0. Głównym argumentem jest to, że w przypadku zerowej latencji, wartość wyjścia nie ustala się tak naprawdę w tym samym czasie, ale w ciągu ułamka nanosekund. W tym czasie na wyjściu występuje stan nieustalony. Ponadto, gdy połącz się dużo elementów o zerowej latencji, czas tych opóźnień się sumuje. Ponieważ systemy powinny działać z dużą częstotliwością, a w przypadku stanów nieustalonych nie ma możliwości sprawdzenia czy wyjście jest już poprawne, to możliwe jest, że zostanie odczytany niepoprawny stan wyjścia. Aby tego uniknąć stosuje się elementy obliczeniowe pracujące synchronicznie (z zegarem) – w tym przypadku latencja wynosi 1. Rozbiecie operacji na prostsze operacje (podobnie jak w przypadku dodawania dwóch liczb metodą słupkową na kartce) zwiększa latencję, umożliwia jednak dalsze zwiększanie częstotliwości zegara, jak zostało zademonstrowane wcześniej. W tabeli 7.1 podano maksymalne częstotliwości pracy dla sumatorów 32-bitowych. Można wyraźnie zauważyć, że elementy o większej latencji umożliwiają taktowanie z większą częstotliwością. Jest to jednak okupione większym użyciem zasobów układu FPGA. Uwaga. W układzie FPGA operacje arytmetyczne mogą być realizowane w prost w logice (LUT) lub za pomocą dedykowanych zasobów – mnożarek (DSP).

W praktyce projektowania systemów wykonujących operacje arytmetyczne w układach FPGA, przyjmuje się że jeśli tylko jest to możliwe to należy wykorzystywać elementy obliczeniowe

Latencja	oparte o LUT			oparte o DSP		
	0	1	2	0	1	2
FF	96	128	145	96	96	96
LUT 6	53	38	80	18	28	24
SLICE	35	46	47	27	32	34
DSP48	0	0	0	1	1	1
Zegar max.	318 MHz	369 MHz	409 MHz	257 MHz	360 MHz	539 MHz

Table 7.1: Różne konfiguracje sumatorów 32-bitowych zrealizowanych w układzie Virtex 7 firmy Xilinx

o maksymalnej dostępnej latencji. Pozwala to na osiągnięcie układów pracujących z największą częstotliwością.

7.4 Pisanie a generowanie

W języku Verilog, istnieją operatory umożliwiające wykonywanie operacji arytmetycznych, są to konstrukcje syntezowalne¹, należą do nich "+", "-" i "*". Operator "/" jest syntezowalny jedynie w przypadku, gdy wartość jest dzielona przez potęgę liczby dwa (wtedy jest to de facto proste przesunięcie bitowe). Przy pomocy tych operatorów można uzyskać elementy o latencji 0 lub 1 w zależności od tego czy wyniki są zapisywane do rejestrów (*reg*) czy do ścieżki/portu wyjściowego (*wire*).

Kod 7.4.1 — Sumator o zerowej latencji:

```
module adder_latency0
(
    //input ports
    input [7:0]a,
    input [7:0]b,
    //output ports
    output [7:0]y
)
assign y=a+b;
endmodule;
```

Kod 7.4.2 — Sumator o latencji równej jednemu cyklowi zegara:

```
module adder_latency1
(
    //input ports
    input [7:0]a,
    input [7:0]b,
    //output ports
    output [7:0]y
)
reg [7:0]r_y;
always @ (posedge clk)
begin
    r_y<=a+b;
end
assign y=r_y;
endmodule;
```

Uzyskanie elementów obliczeniowych o większej latencji, wymaga wykorzystania predefiniowanych bloków logicznych (aplikacja *CORE generator*) i wygenerowania ich jako bloków *IP Core* (czarna skrzynka, w której nie da się nic zmienić). Uwaga. Narzędzie *CORE generator* zostanie omówione w ramach jednego z zadań. *CORE generator* pozwala również na wygenerowanie sprzętowych dzielarek, które mogą dokonywać dzielenia dowolnych dwóch liczb.

¹syntezowalne – takie, które się syntezują. Nie wszystkie konstrukcje języka Verilog da się przenieść do sprzętu. Przykładem są typowe dla symulacji operacje na plikach.

7.5 Pierwiastkowanie, funkcje trygonometryczne, logarytmy

Wykonanie operacji takich jak:

- pierwiastkowanie,
- funkcje trygonometryczne,
- logarytmy
- itd.

nie jest możliwe za pomocą instrukcji w języku Verilog. Należy zauważyć, że podobnie jest z wykonywaniem tych operacji na procesorach CPU. Procesor nie ma sprzętowej instrukcji obliczania logarytmu. Funkcja ta jest obliczana przez bibliotekę programową, która rozwija w szereg odpowiednie przybliżenie numeryczne. W układach FPGA są dostępne wyspecjalizowane moduły, które mogą wykonywać te operacje, jedyną możliwością ich wykorzystania jest natomiast ich wygenerowanie jako elementu typu IP Core w narzędziu CORE generator lub ich samodzielne zaprojektowanie.

7.5.1 Tablicowanie wartości funkcji

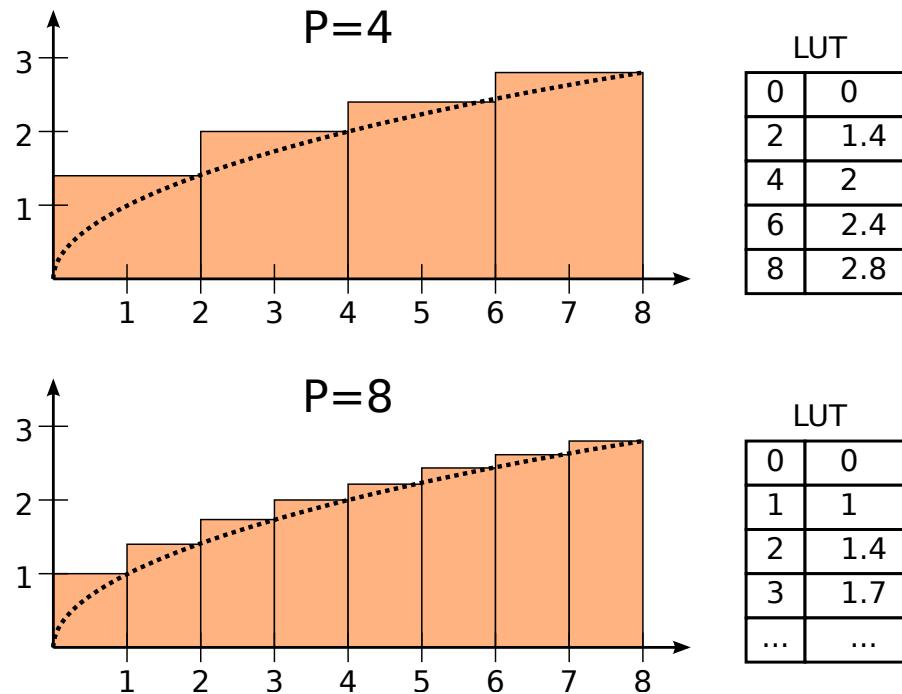
Bezpośrednie obliczanie wartości skomplikowanych funkcji na platformie rekonfigurowalnej jest utrudnione, wymaga bowiem zastosowania dużej liczby elementów obliczeniowych i wielu iteracji metody przybliżonej. W związku z tym, często stosowanym podejściem, umożliwiającym realizację tych obliczeń w prosty sposób jest metoda wykorzystująca tablicowanie wartości funkcji.

W metodzie tej, dla pewnych argumentów są obliczane wartości funkcji i umieszczane w tablicy LUT (rysunek 7.5). Następnie podczas pracy systemu, wartość funkcji jest obliczana poprzez znalezienie dla danego argumentu, najbliższego, który został stablicowany i zwrócenie odpowiadającej mu wartości z tablicy. Takie podejście generuje błędy oraz skutkuje powstaniem charakterystycznych obszarów (schodków), dla których wartość funkcji jest taka sama.

Tablica jest najczęściej realizowana jako blok pamięci BRAM, skonfigurowany do pracy w trybie read-only. Rozmiar pamięci koniecznej do przechowywania wartości jest zależny od kilku czynników:

- zakresu argumentów i liczby przedziałów tablicowania,
- przyjętej dokładności tablicowania wartości funkcji.

Na rysunku 7.5 przedstawiono przykład dla tablicowania funkcji $y = \sqrt{x}$ dla dwóch przypadków. W pierwszym z nich, funkcja jest tablicowana dla 4 przedziałów. Tablica jest niewielka, jednak skutkuje to dużym błędem oddania wyników w dla wartości bliskich 0. W drugim przypadku funkcja jest tablicowana na 8 przedziałach, rozmiar pamięci, która jest konieczna do ich przechowania jest dwukrotnie większy. Zmniejszył się jednak błąd powstający dla wartości bliskich 0.

Figure 7.5: Tablicowanie wartości funkcji $y = \sqrt{x}$

Okazuje się, że metoda ta chociaż mało dokładna, jest chętnie stosowana w praktyce. W zależności od typu funkcji, wymaga jedynie dobrania odpowiedniej liczby przedziałów i reprezentacji przechowywanych liczb. W bardziej zaawansowanych aplikacjach można również wykorzystać dodatkową aproksymację pomiędzy przedziałami histogramu, co pozwala poprawić dokładność.

7.6 Zadania do wykonania na laboratorium

Zadanie 7.1 Zaprojektuj architekturę obliczeniową, która zrealizuje równanie: $Y = (A + B) * C$.

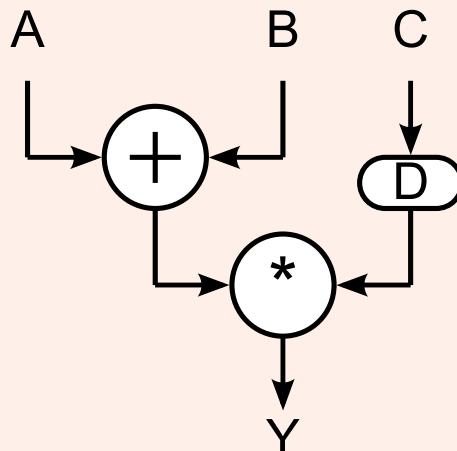


Figure 7.6: Prosta operacja arytmetyczna

Załącz, że argumenty A , B i C to liczby rzeczywiste z przedziału $[-1; 1]$. W pierwszym kroku, w pakiecie Matlab zaimplementuj zadane równanie na liczbach double, a potem na stałoprzecinkowych z "wybieraną" precyzją. Przyjmij wartości testowe liczb jako:

$$A = 0.32345;$$

$$B = -0.78743;$$

$$C = 0.56532;$$

Wyrysuj wykres, który ilustruje popełniany błąd w zależności od wybranej precyzji obliczeń. Wybierz "odpowiednią" precyzję i wykonaj moduł sprzętowy. Jego działanie zweryfikuj.

Realizacja ćwiczenia – podpowiedzi i uwagi:

- W ramach tego, dość prostego zadania, zademonstrowanych zostanie szereg aspektów związanych z realizacją operacji arytmetycznych w układach FPGA: tworzenie modelu programowego, jego analiza, wybór precyzji, korzystanie z narzędzia *CORE generator* i implementacja obliczeń w sprzęcie.
- W pierwszym kroku należy uruchomić pakiet Matlab, stworzyć m-plik i zaimplementować zadane równanie z podanymi argumentami.
- Założyliśmy, że liczby A , B i C są z przedziału $[-1 : 1]$ zatem: trzeba uwzględnić 1 bit na znak, na część całkowitą przeznaczyć 1 bit, a na część ułamkową dowolnie. Zwykle, w pierwszym przybliżeniu stosuje się ograniczenie do 18 bitów (na całość reprezentacji), gdyż jest to maksymalna szerokość słowa obsługiwana przez element DSP w układzie FPGA Spartan 6.
- Do realizacji obliczeń stałoprzecinkowych wykorzystamy "Fixed-Point Designer" Toolbox z pakietu Matlab. W pierwszym kroku należy stworzyć obiekt, który pozwoli na konwersję liczby zmiennoprzecinkowej na format stałoprzecinkowy zadaną precyzją. W tym celu wykorzystuje się polecenie `fi`:

Kod 7.6.1 — Konwersja z liczby zmiennoprzecinkowej na stałoprzecinkową:

```
value=0.32345;
sign=1; %0-unsigned value, 1-signed value
prec_i=8; %number of integer part bits (Nc)
prec_f=8; %number of fractional part bits (Nu)
word = 1 + prec_i + prec_f;

o_fix = fi(value,sign,word,prec_f)

o_fix =
0.3242
DataTypeMode: Fixed-point: binary point scaling
Signedness: Signed
WordLength: 8
FractionLength: 8
```

Posiada ono cztery argumenty: liczbę która ma zostać przekształcona (format *double – value*), informację o znaku (*sign*), liczbę bitów przeznaczoną na całą liczbę (*word*), liczbę bitów przeznaczoną na część ułamkową (*prec_f*). Zwraca liczbę w zadanym formacie stałoprzecinkowym (*o_fix*).

- Dokonaj konwersji liczb A , B i C na format stałoprzecinkowy. Proszę sprawdzić, jak wygląda zapis liczby w formacie stałoprzecinkowym. Do analizy reprezentacji binarnej może być przydatna funkcja `bin(o_fix)` lub `hex(o_fix)`.

- Następnie potrzebujemy funkcję odwrotną, czyli zamianę z formatu stałoprzecinkowego na zmiennoprzecinkowy. Przykładowy kod zamieszczony jest poniżej.

Kod 7.6.2 — Konwersja z liczby stałoprzecinkowej na zmiennoprzecinkową:

```
o=double(o_fix);
```

Zwraca liczbę formacie zmiennoprzecinkowym (o).

- Dokonaj konwersji trzech uzyskanych poprzednio liczb stałoprzecinkowych na format *double*. Porównaj tak uzyskane wartości z wejściowymi.
- Kolejny krok to realizacja dodawania liczb A i B . Dzięki wykorzystaniu toolbox'a, operacja ta jest niezwykle prosta:

Kod 7.6.3 — Dodawanie dwóch liczb stałoprzecinkowych ze znakiem:

```
x = fi(0.32,1,8,4);
y = fi(1.45,1,8,4);
z=x+y

z =
1.7500
 DataTypeMode: Fixed-point: binary point scaling
 Signedness: Signed
 WordLength: 9
 FractionLength: 4
```

Zrealizuj dodawanie liczb A i B w dwóch wariantach: zmiennie i stałoprzecinkowym. Porównaj uzyskane wyniki – np. wypisz obok siebie na konsoli lub podglądnij w *Workspace*.

- Sprawdź cztery możliwości zmieniając znak przy argumentach A i B . Sprawdź co się stanie jak wyniki przekroczy zakres – zmień jeden z argumentów. Czy zwiększenie liczby bitów przeznaczonych na część całkowitą rozwiąże problem?
- Następny krok to dodanie mnożenia, które realizuje się podobnie jak dodawanie. Zaimplementuj równanie $Y = (A + B) * C$. Porównaj wyniki modelu dokładnego i stałoprzecinkowego.
- Przekształć swój kod, tak aby można było w pętli zmienić parametr *prec_f* – od 0 do 16. Dla każdej precyzji oblicz błąd reprezentacji tj. moduł (abs) z różnicą pomiędzy wynikiem dokładnym (format *double*) i stałoprzecinkowym. Zapisz go w tablicy i wyświetl.

Przypomnienie składni Matlab'a:

- definicja bufora na wyniki: `res = zeros(1,17);`
- pętla `for`: `for prec_f=0:16 ... end;`
- zapis wyniku: `res(prec_f+1) = ...` – indeksowanie w Matlab'ie od '1'.
- wyświetlanie wyniku: `plot(res);`

Na podstawie uzyskanych wyników wybierz precyzję obliczeń.

- Przechodzimy teraz do realizacji obliczeń w układzie FPGA. Utwórz nowy projekt w ISE DS. Dodaj moduł nadrzędny w postaci modułu Verilog. Powinien on mieć 5 wejść (*clk*, *ce* i argumenty operacji A, B, C) i jedno wyjście. Szerokości zgodnie z ustaloną uprzednio precyzją. Uwaga. Należy się zastanowić nad najgorszym przypadkiem tj. np. wartość A i $B = 1$.
- W języku Verilog występuje typ `signed` tj. można np. zadeklarować połączenie ze znakiem – `wire signed [2:0] x;`. W przypadku operacji na liczbach ze znakiem należy stosować ten typ, gdyż wtedy np. inaczej realizowane są operatory porównania itp. Uwaga. W trakcie implementacji stosuje się zwykle połączenia (`wire`). Należy

je **bezwzględnie** definiować przed użyciem. Jeśli tego nie zrobimy to narzędzie samo wygeneruje odpowiednie połączenia, ale tylko 1-bitowe. To w większości przypadków prowadzi do błędów! Połączeń nie należy inicjalizować (w odróżnieniu od rejestrów). Przypisanie do *wire'a* wartości 0 powoduje stałe podłączenie tego sygnału do masy.

- Zaczniemy od realizacji operacji dodawania. Do projektu dodaj nowy moduł. Jako typ wybierz *IP(CORE Generator & Architecture Wizard)*. Nazwij go. Otworzy się okno, z listą różnych komponentów. Nas interesuje *Adder Subtractor* (w folderze *Math Functions*). Po jego wybraniu (Next, Finish) otworzy się konfigurator. Pracujemy na liczbach ze znakiem, szerokość ustawiamy zgodnie z użytą precyzją. Ustawiamy latencję na tryb *Automatic*. Zapamiętujemy ile ona wynosi. Generujemy moduł (trwa chwilkę). Moduł pojawi się w hierarchii projektu. Jeśli go wybierzemy to zyskamy dostęp do opcji *CORE Generator -> View HDL Instantiation Template*. Po jej uruchomieniu otworzy się plik z "szablonem" instancji modułu sumatora. Należy go wstawić do projektu. Dobrą praktyką jest pisanie nad modułem jego latencji.
- Zgodnie z omówieniem z rozdziału 7.3 argument *C* należy odpowiednio opóźnić. Można do tego celu wykorzystać zrealizowany w ramach zadania 5.4 moduł. Należy go oczywiście odpowiednio skonfigurować (szerokość i latencję). Uwaga. Tutaj też warto skorzystać z opcji: *Design Utilites-> View HDL Instantiation Template*. Niestety, sekcję odpowiedzialną za parametry należy dodać "ręcznie".
- Trzeci potrzebny moduł to mnożarka. Ponownie dodajemy moduł *CORE Generator*, tylko tym razem – *Multiplier*. Konfigurujemy go. Na stronie 1 ustalamy format. Na stronie 2 możemy wybrać czy używamy elementów LUT, czy DSP (Mult). Ustawiamy *Use Mults*. W tym miejscu można także ew. ustawić czy moduł ma być bardziej wydajny czy kompaktowy. Na stronie 3 ustalamy latencję. Wybieramy optymalną i zapamiętujemy jej wartość. Moduł generujemy i wstawiamy do projektu.
- Na końcu należy wypisać na wyjście modułu "odpowiednie" bity z wyniku mnożenia.
- Następnie tworzymy testbench. Dodajemy generowanie zegara. Ustalamy początkowe wartości *A*, *B* i *C* na podstawie modelu programowego. Sprawdzamy czy uzyskujemy poprawny wynik. Dodatkowo sprawdzamy, czy wyniki pojawia się po tylu taktach zegara, po ilu się go spodziewamy. Uwaga. Do eliminacji błędów bardzo przydaje się model programowy. Dzięki niemu mamy wszystkie wyniki pośrednie i możemy precyzyjnie zlokalizować na którym etapie występuje nieprawidłowość.

7.7 Zadania do wykonania w domu

Zadanie 7.2 Proszę zaprojektować moduł pokazany na rysunku 7.7.

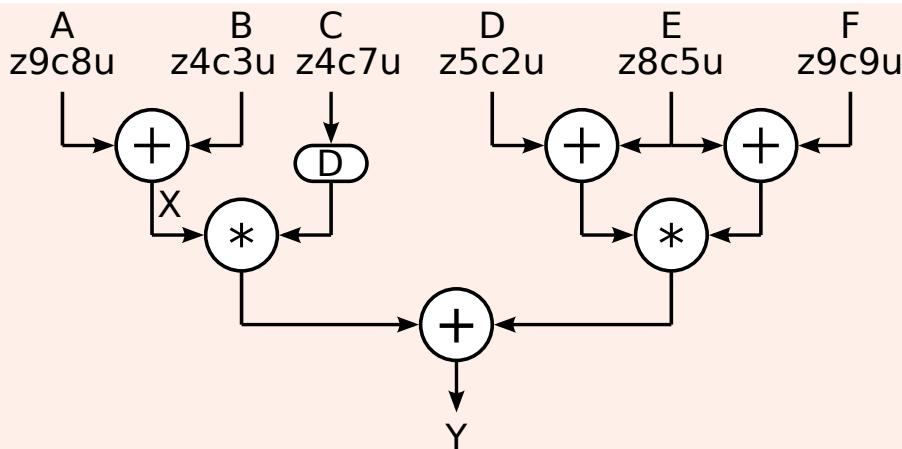


Figure 7.7: Złożony moduł arytmetyczny

Realizacja ćwiczenia – podpowiedź i uwagi:

- Proszę przrysować schemat modułu na kartkę i oznaczyć jakie będą formaty na wejściach i wyjściach poszczególnych sumatorów i mnożarek. Należy wykorzystać wiedzę z podrozdziału 7.1.4.
- Proszę w pierwszej kolejności zaimplementować sumator $X=A+B$ (tj. moduł IP CORE o wejściach zgodnych z pokazanymi na rysunku) i wykonać dodawanie (symulacja) dla liczb $A=1,7$ $B=2,5$. Warto również stworzyć model w Matlab’ie – ułatwi generowanie danych do symulacji. Czy wynik jest poprawny tj. zgodny z modelem? Dlaczego?
- Aby dodawać dwie liczby stałoprzecinkowe, wymagane jest, aby szerokość ich części ułamkowej była identyczna. Jeśli ten warunek nie jest spełniony, możliwe jest rozszerzenie krótszej z liczb poprzez dopisanie zer na najmniej znaczące pozycje części ułamkowej. W celu poszerzenia wektora, stosuje się nawiasy klamrowe:

Kod 7.7.1 — Przykład poszerzania wektora:

```
wire [7:0] x;
wire [9:0] y;

assign y={x, 2'b0};
```

Jak w związku z tym należy zmodyfikować formaty na wejściach i wyjściach do poszczególnych sumatorów (na kartce)? Zmodyfikuj sumator (IP CORE). Sprawdź czy teraz wynik jest poprawny.

- Proszę sprawdzić (symulacyjnie) czy konieczne jest takie rozszerzanie wektorów w przypadku mnożarek? Należy postąpić podobnie jak w przypadku sumatora tj. model Matlab + symulacja ISim.
- Proszę odpowiednio zmodyfikować formaty na wejściach i wyjściach poszczególnych mnożarek (na kartce).
- Proszę wygenerować odpowiednie mnożarki i zapisać latencje poszczególnych modułów.
- Czy latencja na wszystkich poziomach jest taka sama? Jeśli nie, to gdzie trzeba dodać moduły opóźniające? Proszę zmodyfikować schemat modułu (na kartce).
- Proszę opisać cały moduł z rysunku 7.7.
- Proszę stworzyć pełny model programowy w pakiecie Matlab. Przyjmijmy, że wartości

poszczególnych portów wynoszą: A=-100,34 B=7,367 C=-4,92 D=9,111
 E=-99,99 F=134,56

Jaki jest błąd spowodowany realizacją tych obliczeń na liczbach stałoprzecinkowych?

- Proszę zapisać poszczególne liczby A-F binarnie (polecamie *bin* w programie matlab).
- Proszę stworzyć środowisko testowe (testbench) umożliwiające weryfikację poprawności działania modułu.
- Proszę zaproponować jeszcze trzy zestawy wartości portów A-F i podać je w testbenchu na odpowiednie wejścia modułu arytmetycznego, w kolejnych taktach zegara.

Zadanie 7.3 Zaprojektowane podczas laboratoriów moduły działały potokowo i umożliwiały przeprowadzanie obliczeń arytmetycznych na liczbach, które były podawane na poszczególne porty w sposób równoległy. W niniejszym zadaniu zaprojektowana zostanie architektura, która umożliwia zsumowanie wartości pojawiających się na jednym porcie w kolejnych taktach zegara (tzw. akumulację wartości).

Schemat modułu został przedstawiony na rysunku 7.8

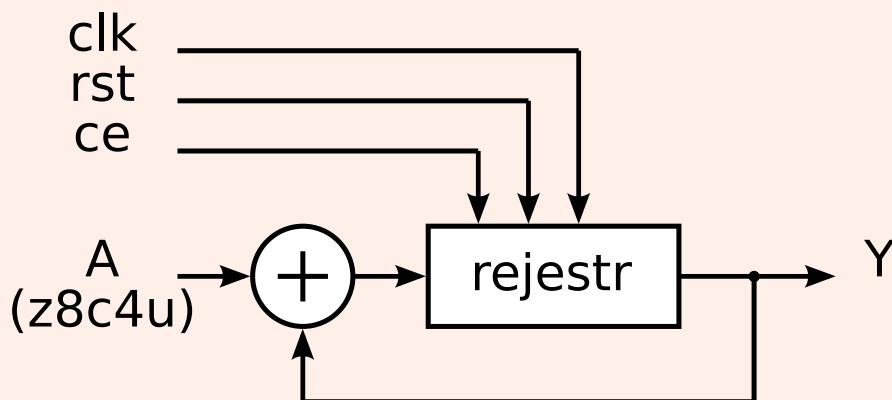


Figure 7.8: Moduł akumulujący

Jeśli na narastającym zboczu zegara, na wejściu *ce* znajduje się 1, to wartość z portu A powinna zostać dodana do poprzedniej wartości z rejestrze. Wystąpienie sygnału *rst* powinno umożliwić wyzerowanie wartości w rejestrze. Przyjmijmy założenie, że rozmiar rejestrów akumulacyjnych musi umożliwić dodanie maksymalnie 256 wartości z portu A.

Realizacja ćwiczenia – podpowiedzi i uwagi:

- Proszę zastanowić się, jakie powinny być formaty wejściowe i wyjściowe w wykorzystywanym sumatorze. Jak szeroki musi być rejestr?
- Proszę wygenerować odpowiedni sumator. Jaka musi być latencja sumatora, aby możliwa była akumulacja wartości z portu przychodzących w następujących po sobie taktach zegara?
- Proszę zaimplementować moduł z rysunku 7.8.
- Proszę napisać model programowy. W tym celu konieczne jest wykorzystanie funkcji *accumpos()*. Proszę zapoznać się z przykładem jej użycia, ze strony <http://www.mathworks.com/products/fixed-point-designer/> (Code Examples – > Perform Fixed-Point Arithmetic – > Modeling Accumulators).
- Proszę wygenerować ciąg 10 liczb w formacie z8c4u i wyznaczyć ich sumę przy pomocy modelu programowego.

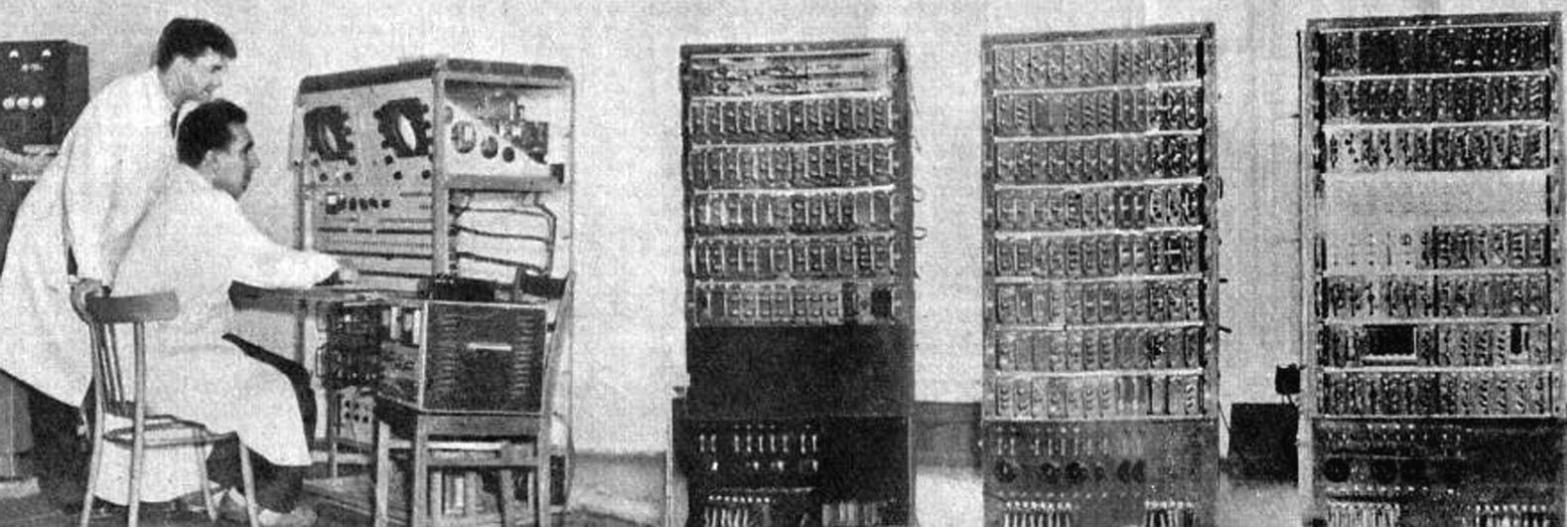
- Proszę napisać środowisko testowe (testbench) umożliwiające sprawdzenie, czy opisany moduł sprzętowy działa poprawnie.

7.8 Zadania dodatkowe

Zadanie 7.4 Proszę stworzyć moduł, który umożliwia wykonanie mnożenia macierzowego. Proszę przyjąć, że porty A i B mają precyzję z8c4u.

$$\begin{pmatrix} Y \\ Z \end{pmatrix} = \begin{pmatrix} -0.11 & 2.3 \\ 3.14 & -11.25 \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} \quad (7.6)$$

Proszę opisać również środowisko testowe (testbench + model programowy w Matlabie), który sprawdzi, czy moduł działa poprawnie dla co najmniej 8 wartości A i B (w tym skrajne wartości na tych portach).



8 — Potokowe przetwarzanie i analiza obrazów

8.1 Wstęp teoretyczny

Typowy system wizyjny składa się z kamery, elementu realizującego obliczenia oraz ew. urządzenia do wizualizacji wyników lub ich transmisji. Moduł obliczeniowy realizuje wiele pojedynczych operacji przetwarzania wstępnego, analizy i rozpoznawania obrazów. Całość określa się jako potok przetwarzania i analizy obrazów. Ilustruje to schemat przedstawiony na rysunku 8.1



Figure 8.1: Schemat typowego systemu wizyjnego

Cechą charakterystyczną przetwarzania potokowego, dla systemu wizyjnego z układem FPGA, jest dokonywanie operacji bezpośrednio na strumieniu pikseli odbieranych z kamery. Obliczenia odbywają na wszystkich pikselach, informacja nie jest tracona. Cały tor wizyjny wprowadza jedynie pewną, zwykle niewielką, latencję.

Szczegółowe omówienie poszczególnych etapów wykracza poza ramy niniejszego skryptu i kursu. Dla zainteresowanych osób polecana jest następująca literatura [13], [2], [1], [12]. W tym miejscu warto nadmienić, że przykładem operacji przetwarzania wstępnego są: korekcja gamma, konwersja pomiędzy przestrzeniami barw (np. RGB → HSV, RGB → YCbCr), filtracja Gaussa (uśredniająca, dolnoprzepustowa), filtracja medianowa, różne operacje morfologiczne (erozja, dylatacja). Posiadają one jedną wspólną cechę – informacja na wyjściu i wejściu jest zasadniczo taka sama (pewnym wyjątkiem jest konwersja obrazu kolorowego do odcieni szarości).

Natomiast analiza obrazu to proces wydobywania z niego istotnej informacji. "Istotność" ta jest ściśle powiązana z docelową aplikacją. Przykładami są: segmentacja obiektów (podział sceny na poszczególne obiekty: ludzi, samochody itp.), indeksacja (przypisanie do poszczególnych pikseli etykiet) oraz wyliczanie współczynników kształtu lub innych deskryptorów cech (HOG, SIFT, SURF, LBP, GLCM i inne). Charakterystyczne jest to, że na wejściu mamy dany obraz (kolorowy, w odcieniach szarości, binarny), a na wyjściu najczęściej opis w postaci wektorów cech dla poszczególnych obiektów (przykładowo ich pola).

Ostatni etap to rozpoznawanie, w którym podstawie cech obiektów (zebranych zazwyczaj w tzw. wektor cech) dokonuje się klasyfikacji obiektów np. określa czy klocek na scenie ma kształt prostokątny, trójkątny czy okrągły. W tym celu wykorzystuje się różnego rodzaju klasyfikatory od prostych opartych o progowanie współczynników po złożone: sieci neuronowe, maszyny wektorów nośnych SVM, drzewa decyzyjne czy sieci bayesowskie. Rezultatem tego etapu jest tzw. semantyczny opis sceny czyli nazwanie poszczególnych obiektów. W ogólnym, idealnym, przypadku wszystkich, w realnych tylko wybranych. Przykładem może być zagadnienie detekcji ludzi na scenie.

Implementacja sprzętowa algorytmów każdego z etapów jest możliwa w układzie FPGA. Oczywiście najlepsze do implementacji równoległej i potokowej są metody, w których występuje duża liczba stosunkowo prostych i powtarzalnych operacji, a dostęp do danych jest uporządkowany. Znaczenie mają również wykorzystywane w algorytmie operacje arytmetyczne. Poza tym, zawsze warto pamiętać o możliwości realizacji "niewygodnych" etapów w ramach architektury opartej o procesor Microblaze lub ARM (dla układów Zynq).

8.2 Typowy cyfrowy interfejs wizyjny

Typowy interfejs modułu do potokowego (szeregowego) przetwarzania cyfrowego strumienia wizyjnego zaprezentowano na rysunku 8.2.

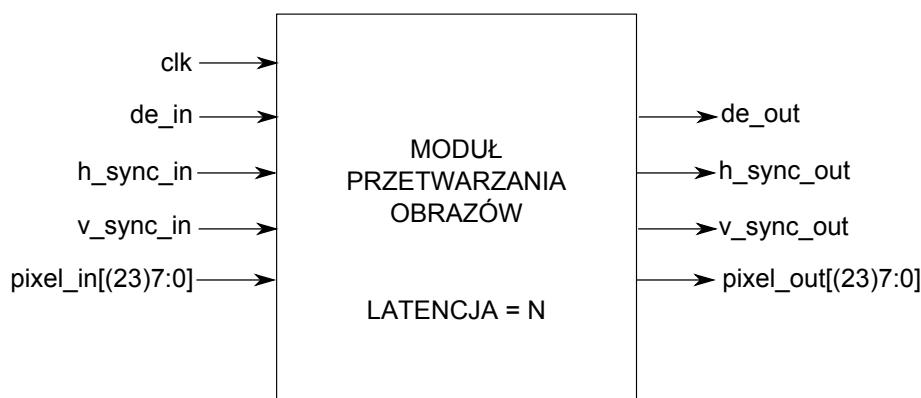


Figure 8.2: Schemat typowego modułu do przetwarzania obrazów

Jest to interfejs minimalny tj. występują w nim tylko niezbędne sygnały. Może on zostać, w zależności od aplikacji, poszerzony: po stronie wejść o różne parametry (np. maska filtracji, próg binaryzacji itp.), a po stronie wyjść wynikiem niekoniecznie musi być wyłącznie piksel, a np. przepływ optyczny (dwie liczby), maska binarna lub opis w postaci cech.

Opis sygnałów:

- *clk* – zegar. W tym przypadku jest to tzw. zegar piksela, czyli zegar, który taktuje poszczególne piksele. Przebieg zegara, dla strumienia RGB pokazano na rysunku 8.3. Warto zaznaczyć, że transmisja cyfrowego sygnału video odbywa się właśnie w taki szeregowy sposób.
- *de_in* (data enable) – flaga oznaczająca, że dany piksel jest "ważny" tj. zawiera poprawne dane wizyjne, piksele.
- *hsync_in* – flaga oznaczająca synchronizację poziomą.
- *vsync_in* – flaga oznaczająca synchronizację pionową.
- *pixel_in* – dane dla obrazu w odcieniach szarości (wektor 8-bitowy) lub kolorowego (wektor 24-bitowy).
- *de_out* – opóźniona flaga *de*.

- *hsync_out* – opóźniona flaga *hsync*.
- *vsync_out* – opóźniona flaga *vsync*.
- *pixel_out* – przetworzony (i opóźniony) piksel (wektor 8 lub 24-bitowy).

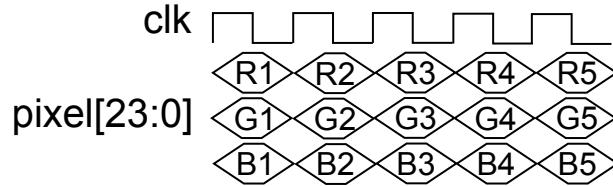


Figure 8.3: Przykładowy przebieg zegara piksela

Sygnały sterujące tj. *de*, *hsync*, *vsync* wymagane są do poprawnego wyświetlania obrazu na ekranie monitora, a generowane są przez urządzenie dokonujące akwizycji tj. kamerę. Synchronizacja pionowa i pozioma początek swój wzięła z pierwszych telewizorów analogowych (CRT – ang. *cathode-ray tube*), które działały na zasadzie "ostrzeliwania" ekranu wiązką elektronów. Aby wyświetlić pełny obraz "działko" poruszało się do lewej do prawej (skanowanie poziome) oraz od góry do dołu (skanowanie pionowe). Sygnały synchronizacji sterowały tym procesem. Przerwa w wyświetleniu obrazu pozwalała na powrót działka do początkowego położenia. Na podstawie sygnałów synchronizacji można wyznaczyć moment wystąpienia nowej linii lub nowej ramki, z czego będziemy korzystać.

Na rysunku 8.4 pokazano sygnały dla obrazu o rozdzielcości 640×480 .

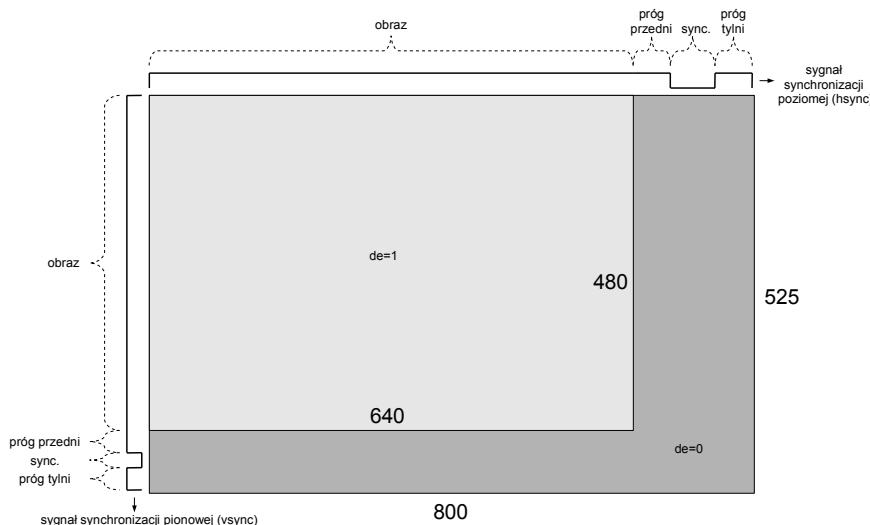


Figure 8.4: Synchronizacja dla obrazu o rozdzielcości 640×480

Można zauważyć, że rzeczywista rozdzielcość obrazu jest większa od wyświetlonej (800×525 vs. 640×480). Flaga *de* ma wartość 1 tylko w obszarze obrazu, a *hsync* i *vsync* w danej linii i ramce.

Moduł posiada jeszcze jeden parametr – latencję. Latencja ma duże znaczenie w potokowym przetwarzaniu danych, gdyż określa opóźnienie pomiędzy danymi wejściowymi, a wyjściowymi jakie wprowadza dany moduł lub system. Jak zostało to już zademonstrowane wcześniej, należy ją uwzględniać przy projektowaniu systemu, aby zapewnić odpowiednią synchronizację działania. Natomiast z punktu widzenia funkcjonalności aplikacji, dla strumienia wideo o 60 ramkach

na sekundę, wprowadzenie kilku linii opóźnienia (a nawet całej ramki) jest praktycznie niezauważalne i zwykle nieistotne (wyjątkiem są aplikacje o bardzo rygorystycznych wymaganiach czasowych).

Czy sygnały synchronizacji są potrzebne/wykorzystywane w przetwarzaniu obrazu ?

To zależy od implementowanej operacji. Np. w dodawaniu dwóch obrazów czy operacji LUT nie mają one znaczenia – moduł może również realizować funkcjonalność przy $de=0$, a jedynie "niepoprawny" wynik powinien być pomijany przy wyświetlaniu.

Natomiast przy operacjach kontekstowych de ma już duże znaczenie, gdyż flaga pozawala określić poprawność kontekstu – zagadnienie zostanie omówione szerzej w ramach zagadnień związanych z operacjami kontekstowymi 12.

Na podstawie sygnałów synchronizacji możliwe jest również wyznaczenie położenia piksela na obrazie – przydatne np. przy wyświetlaniu.

Czy sygnały synchronizacji są potrzebne do wyświetlania obrazu ?

Zdecydowanie tak. Nieprawidłowe sygnały synchronizacji powodują albo "pływanie" obrazu albo uniemożliwiają współpracę z monitorem. Dlatego, przy realizacji wszystkich operacji należy zadbać o to, aby sygnały synchronizacji "nadążały" za pikselem (właściwym). Metoda najprostsza to "doklejenie" ich do piksela, a trudniejsza to odpowiednie generowanie (podejście takie pozawala zaoszczędzić zasoby logiczne).

Podsumowując. Każdy moduł realizujący przetwarzanie lub analizę obrazów powinien, oprócz obliczeń na danych, zapewniać opóźnienie sygnałów de , $hsync$, $vsync$ dokładnie o wartość latencji, jaką wprowadzają te obliczenia.

8.3 Model programowy przetwarzania obrazów

W tym i następnych ćwiczeniach będziemy wykorzystywać model toru wizyjnego do symulacyjnego testowania modułów przetwarzania obrazów. Pozwala on zrealizować przetwarzanie pojedynczego obrazka wczytanego z pliku w formacie ppm (ang. *portable pixmap format*) – chyba najprostszym z możliwych (bardzo prosty nagłówek i dane w postaci nieskompresowanej). Wynik przetwarzania również zostanie zapisany do pliku ppm. W modelu, oprócz danych wejściowych, generowane są również sygnały synchronizacji – de , $hsync$ i $vsync$.

Zadanie 8.1 Uruchom symulację toru wizyjnego. Jest ona zawarta w archiwum dostępnym w repozytorium kursu (*hdmi.zip*). ■

Model składa się z trzech plików:

- *hdmi_in.v* – wczytywanie pliku i generacja sygnałów synchronizacji,
- *hdmi_out.v* – zapis do pliku,
- *tb_hdmi.v* – pusty plik testowy (bezpośrednie połączenie modułów *hdmi_in* z *hdmi_out*).

Uruchom symulację (behavioralną) modułu *tb_hdmi*. Upewnij się, że w folderze projektu znajduje się plik *geirangerfjord_64.ppm* (w paczce). Uwaga. Aby oszczędzić czas operujemy na obrazie o rozdzielcości 64×64 . Symulacja nawet prostych operacji przetwarzania i analizy obrazu jest dość czasochłonna.

Sprawdź, czy w wyniku symulacji otrzymano poprawny obraz. Zwróć uwagę na odpowiedni czas trwania symulacji (**min. 20 us**). Zwróć również uwagę na przebieg sygnałów de , $hsync$ i $vsync$.

Uwaga. Zarówno podczas tworzenia potoku przetwarzania do symulacji lub później do implementacji należy zadbać o **dobre nazewnictwo sygnałów**. Dobrze jest do nazwy dodać element związanego z modelem źródłowym np. ".in" dla źródła sygnału. Praktyka ta pozwala uniknąć błędów w przypadku konieczności połączenia większej liczby modułów.

8.4 Uruchomienie toru wizyjnego na karcie Atlys

Zadanie 8.2 Uruchom tor wizyjny na karcie Atlys. Potrzebne pliki zawarte są w archiwum *hdmi.zip*. Uwaga. W zależności od oznaczania na kablu HDMI, konieczne jest zdefiniowanie lub zakomentowanie linii `'define SPLITTER` w pliku *hdmi_main.v*.

- kabel zielony – odkomentowane,
- kabel czerwony – zakomentowane.

Sposób podłączenia karty Atlys przedstawiono na rysunku 8.4. Uwagi:

- na karcie Atlys używamy portów HDMI IN i HDMI OUT umieszczonych na górnjej krawędzi,
- do HDMI IN podpinamy sygnał wizyjny z kamery,
- do HDMI OUT kabel łączący z monitorem LCD,
- wyniki oglądamy po wybraniu odpowiedniego wejścia na monitorze,
- proszę nie zapomnieć, że projekt trzeba zaimplementować oraz wygenerować plik konfiguracyjny *bit*, który następnie, poprzez port USB, należy wgrać na kartę.

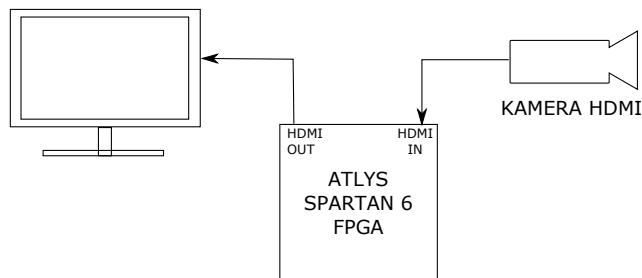


Figure 8.5: Schemat podłączenia kamery i monitora do karty Atlys

8.5 Realizacja operacji LUT

Operacja LUT (ang. look-up table) to jedna z najprostszych operacji punktowych w przetwarzaniu obrazów. Polega na przekształceniu wartości piksela zgodnie z uprzednio zdefiniowaną tablicą przekodowania. W FPGA realizowana jest zwykle z wykorzystaniem modułu ROM (ang. read-only memory – pamięci rozproszonej lub blokowej). W tym przypadku operacja LUT działa tak, że jako adres (ADDR) podaje się wartość piksela, a w wyniku zwracana jest wartość zapisana w pamięci pod tym adresem (VAL), która staje się nową wartością piksela. Schematycznie zostało to przedstawione na rysunku 8.5.

Zadanie 8.3 Zrealizuj moduł LUT oparty o pamięć rozproszoną układu FPGA. Moduł ma przetwarzać liczby z zakresu [0:255] tj. 8-bitowe. Przetestuj go symulacyjnie w modelu toru wizjengo, a następnie zweryfikuj jego działanie na karcie Atlys.

Uwagi do realizacji:

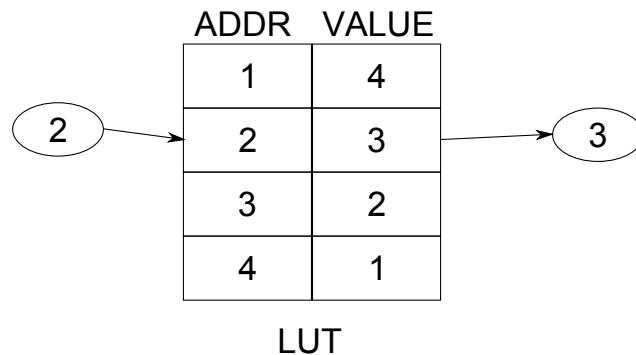


Figure 8.6: Operacja LUT

- Tworzymy nowy IP Core np. LUT. W narzędziu CORE Generator odszukujemy element *Distributed Memory Generator*. Nie używamy pamięci blokowej (BRAM), gdyż jest ona zbyt duża dla naszych potrzeb.
- W oknie konfiguradora trzeba wybrać kilka opcji:
 - *Depth* – liczbę komórek pamięci (określ samodzielnie),
 - *Data Width* – rozmiar pojedynczej komórki pamięci w bitach (określ samodzielnie),
 - *Memory Type* – typ pamięci. Nas interesuje pamięć tylko do odczytu tj. ROM.
- Na drugiej karcie ustawiamy rejestrację wyników (*Output Options -> Registered*) oraz latencję (Pipeline Stages) na 0 (taka się wyświetli).
- Na trzeciej zakładce należy załadować zawartość pamięci – podać odpowiedni plik *.coe. W pliku tym zapisana są wartości, które zostaną załadowane do modułu ROM. Przykładowy plik zamieszczony jest w głównym folderze projektu. Podglądaj wczytaną zawartość. Wygeneruj moduł.
- Instancję LUT dodaj do modelu symulacyjnego toru wizyjnego i odpowiednio połącz. De facto potrzebujemy trzech instancji – przetwarzamy każdą składową barwną niezależnie. Zauważ, że założyliśmy latencję modułu 1. Zatem trzeba opóźnić sygnały synchronizacji o '1'. Najprościej to zrobić wykorzystując instrukcję `always` i trzy rejesty.
- Sprawdź symulacyjnie poprawność rozwiązania.
- Dodaj moduły do toru wizyjnego przeznaczonego dla karty Altys i uruchom go. Zwróć uwagę na odpowiednie nazewnictwo sygnałów. Jako sygnału zegarowego użyj `rx_pclk`, reszta nazw jest "intuicyjna". Nie zapomnij wyjść modułów LUT połączyć z wyjściem w sekcji "*HDMI output port signal assignments*".

8.6 Zadania do wykonania w domu

Zadanie 8.4 Za pomocą elementów i operacji logicznej LUT zrealizuj binaryzację dla strumienia RGB.

Podpowiedź:

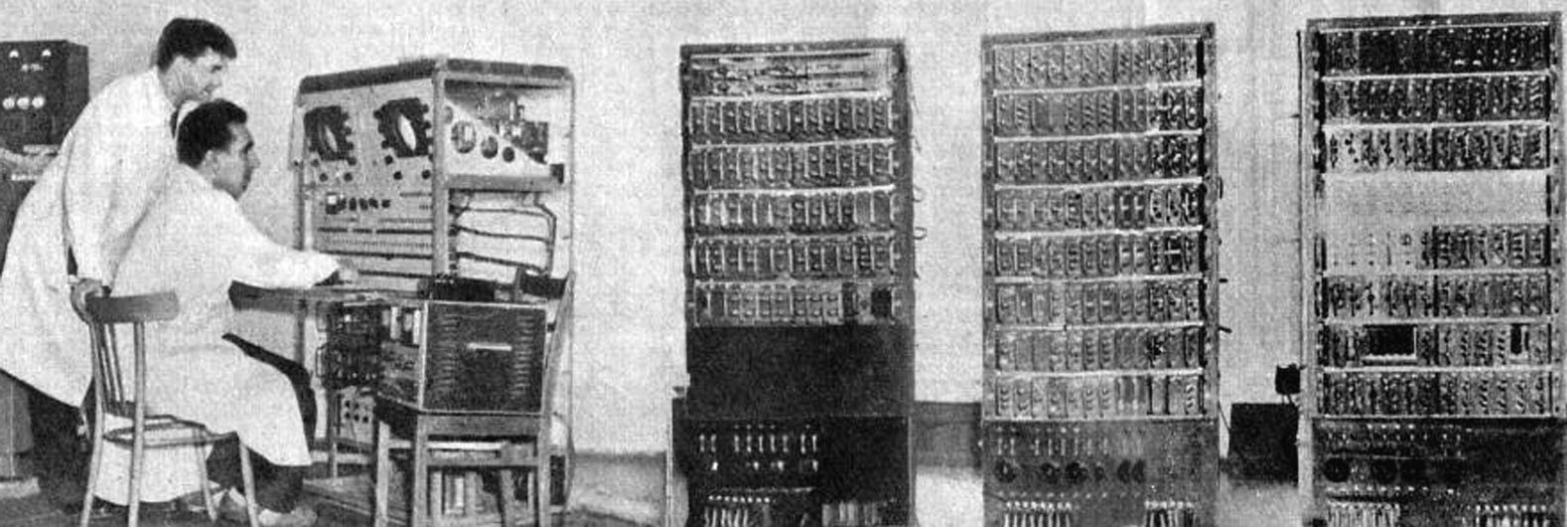
- klucz to postać pliku *.coe. Należy wykonać analizę istniejącego lub przeglądając dokumentację modułu *Distributed Memory*. Dostęp poprzez narzędzie IP CORE Generator (przycisk *Datasheet*). Następnie trzeba utworzyć nowy plik *.coe i podmienić w module LUT.
- Wyjścia z trzech modułów LUT należy połączyć operatorem AND (iloczyn logiczny). Można to zrobić przy przypisaniu do końcowych wartości. Uwaga. Na każdy kanał wyjściowy powinien trafić ten sam rezultat. Inaczej wyświetlanie nie będzie poprawne.

- nowy moduł należy przetestować symulacyjnie oraz na karcie Atlys.

Zadanie 8.5 Wykonaj testy symulacyjne modułu LUT dla różnych modeli tj. behawioralnego, po fazach translacji, mapowania oraz łączenia i rozmieszczania. Zaobserwuj różnice.

Przebieg zadania:

- utwórz nowy plik Verilog – *mainLUT.v*. Jako wejście ustaw: *clk* i *a* (8 bitów), a wyjście *qspo* (8 bitów). Wewnątrz modułu utwórz instancję LUT. Uwaga. Bezpośrednie symulowanie modułu IP Core w ISE, na poziomie innym niż behawioralny, nie jest możliwe, stąd potrzeba "opakowania" go w dodatkowy plik (wrapper). Uwaga. Upewnij się, że moduł *mainLUT.v* oznaczony jest jak *Top Module* dla projektu.
- utwórz nowy plik testowy Verilog, wybierz moduł *mainLUT*, dodaj generowanie sygnału zegarowego (uwaga **okres 10 ns**) – zegar powinien być generowany w osobnej sekcji *initial*, ustal też jakąś przykładową wartość wejściową.
- uruchom symulację. Zwróć uwagę, aby na liście wyboru zaznaczona była pozycja *Behavorial*.
- na przebiegach symulacji odszukaj miejsce, gdzie ustawiana jest wartość sygnału *a*. Upewnij się, że następuje to przed narastającym zboczem zegara (tj. w momencie gdy zbocze się pojawia sygnał ma już ustaloną wartość).
- następnie zaobserwuj, gdzie zmienia się wartość *qspo* (wyjście z modułu LUT). Zmierz opóźnienie pomiędzy zboczem zegara, a pojawiением się wartości na wyjściu *qspo* (pierwszym poprzedzającym). W tym celu:
 - powiększ wykres,
 - zaznacz narastające zbocze zegara,
 - trzymając lewy przycisk myszy przeciągnij kurSOR do miejsca, gdzie zmienia się wartość *qspo*,
 - zmierzone opóźnienie zanotuj.
- powtarzaj opisane wyżej czynności dla kolejnych modeli symulacyjnych: *post translate*, *post map*, *post route*.
- zastanów się/doczytaj z czego wynikają różnice w uzyskiwanych opóźnieniach.



9 — Segmentacja obszarów o kolorze skóry

9.1 Wprowadzenie

W ramach kilku kolejnych laboratoriów zajmiemy się tworzeniem systemu wizyjnego opartego o układ FPGA w celu zademonstrowania możliwości sprzętowej realizacji złożonych algorytmów wizyjnych. Zadaniem systemu będzie wykrycie na obrazie dostarczonym z kamery HDMI fragmentów, na którym znajduje się twarz lub ręka obserwowanej przez kamerę osoby. W tym celu zostanie wykorzystany algorytm, opierający się na segmentacji obszarów o kolorze zbliżonym do koloru skóry. Dodatkowo, wyznaczone zostanie przewidywane położenie środka ciężkości poszukiwanego obiektu. System ten może znaleźć zastosowanie np. w aparatach fotograficznych, do ustawiania punktu ostrości na obszarze, który z dużym prawdopodobieństwem jest twarzą fotografowanej osoby.

Warto w tym miejscu zaznaczyć, że nie jest to podstawowe podejście do problemu detekcji twarzy na obrazie. Najbardziej rozpowszechniona wydaje się być metoda zaproponowana przez Paula Viola i Michela Jonesa oparta o cechy Haara i algorytm uczący AdaBoost [14]. Dostępna jest ona np. w popularnej bibliotece do przetwarzania obrazów OpenCV (opencv.org) Tym niemniej, kolor może być informacją wspomagającą, która pozwala wykluczać tzw. fałszywe detekcje.

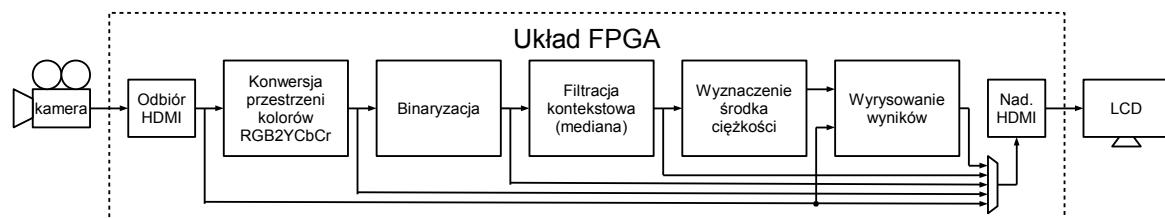


Figure 9.1: Projektowany system wizyjny

Schemat zaproponowanego rozwiązania zostało przedstawiony na rysunku 9.1. W pierwszej kolejności, obraz z kamery jest odbierany po złączu HDMI i zamieniany na postać równoległą jak omówiono w rozdziale 8. Kolejny blok ma za zadanie dokonać konwersji przestrzeni barw z formatu RGB na YCbCr, w którym łatwiej wykrywać kolor skóry. Skóra ma specyfczną

chrominancję – wpadającą w czerwień. Właściwość ta jest zachowana (oczywiście w pewnym zakresie) dla przedstawicieli różnych ras. Dlatego też użycie przestrzeni, w której luminancja i chrominancja są odseparowane (np. YCbCr) pozwala na uzyskanie lepszych rezultatów, niż analiza w przestrzeni RGB.

Kolejny blok jest wykorzystywany do binaryzacji obszarów, w których wykryto odcień odpowiadający skórze. Otrzymana maska binarna poddawana jest filtracji kontekstowej: medianowej lub morfologicznej (erozja, dylatacja, otwarcie, zamknięcie). Wygładzona maska binarna przesyłana jest do kolejnego bloku, który wyznacza środek ciężkości obszaru uznanego za kolor skóry. Następnie punkt ten jest bezpośrednio oznaczany na przesyłanym obrazie. Wykorzystywany zostanie również multiplekser, który będzie umożliwiał przełączanie danych przesyłanych do monitora LCD na wyjście dowolnego z zaprojektowanych bloków.

Przybliżony plan pracy będzie zatem następujący:

- implementacja modelu programowego algorytmu w pakiecie Matlab – zadanie domowe,
- projekt i implementacja konwersji przestrzeni barw z RGB na YCbCr – jedno laboratorium,
- projekt i implementacja modułu binaryzacji, realizacja multipleksera, uruchomienie systemu – jedno laboratorium,
- projekt i implementacja modułu wyznaczania środka ciężkości – uruchomienie systemu – jedno laboratorium
- projekt i implementacja modułu filtracji kontekstowej oraz uruchomienie całości systemu – dwa laboratoria.

9.2 Konwersja RGB do YCbCr — podstawy

Najczęściej wykorzystywany formatem zapisywania obrazów kolorowych jest format RGB. Jest on stosowany w kamerach cyfrowych (mozaika Bayera), w monitorach komputerowych i wielu innych urządzeniach. Wykorzystuje on zwykle trzy wartości, które są liczbami całkowitymi z przedziału od 0 do 255, do zapisania składowej czerwonej (R), zielonej (G) i niebieskiej (B) (tj. 24 bity na piksel). Okazuje się jednak, że w tej przestrzeni barw, wydzielenie grup kolorów, które są podobne (dla człowieka) poprzez proste progowanie wartości każdego z kanałów jest problematyczne. Z tego powodu, zaproponowano wiele innych przestrzeni barw (HSV, CIE Lab). Jedną z nich jest format YCbCr, w którym obraz kolorowy reprezentowany jest przez składową luminancji (Y) oraz dwie składowe chrominancji (Cb, Cr). Wykorzystuje się go między innymi podczas kodowania obrazów w formacie JPEG. Zaletą przestrzeni jest także dość prosta konwersja RGB-> YCbCr, w odróżnieniu od np. konwersji RGB-> CIE Lab, RGB->HSV.

Obraz w przestrzeni RGB i YCbCr oraz poszczególne składowe w skali szarości zostały pokazane na rysunku 9.2.

W celu konwersji obrazów z przestrzeni barw RGB do formatu YCbCr, stosowane jest następujące równanie:

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \quad (9.1)$$

gdzie: R,G,B to odpowiednio składowa czerwona, zielona i niebieska (wartości od 0 do 255).

9.3 Binaryzacja

Aby wykryć te piksele, które mają kolor zbliżony do koloru skóry, konieczne jest wykonanie operacji progowania obrazu. W rezultacie uzyskamy maskę binarną. Dla każdego piksela określa ona, czy należy on do poszukiwanego obiektu (w tym przypadku twarzy lub ręki), czy nie.



Figure 9.2: Oryginalny obraz "lena" oraz poszczególne jego kanały w przestrzeni RGB oraz obraz w przestrzeni YCbCr i poszczególne kanały w skali szarości

Aby wyznaczyć te piksele, które mają kolor podobny do koloru skóry, konieczne jest wykonanie porównań wartości z kanałów opisujących chrominancję (Cb,Cr) zgodnie z równaniem:

$$\text{maska} = \begin{cases} 1 & \text{jeśli } Ta < Cb < Tb \text{ i } Tc < Cr < Td \\ 0 & \text{w pozostałych przypadkach} \end{cases} \quad (9.2)$$

Wartości progów Ta , Tb , Tc , Td ustala się w sposób eksperymentalny. Dzięki analizie chrominancji, a pominięciu luminancji system jest teoretycznie niezależny od oświetlenia i karnacji osoby, której twarz/ręka ma być wykrywana.

9.4 Filtracja

Uzyskana maska binarna zwykle zawiera szумy, czy to w postaci niewielkich grup pikseli wykrytych jako obszar skóry, czy też dziur wewnętrznych "teoretycznie" jednolitych obszarów. Zatem zwykle pożądane jest przeprowadzenie filtracji. Do wyboru są dwa podejścia: filtracja medianowa dla obrazów binarnych lub wybrana operacja morfologiczna. W rozważanym systemie zastosujemy medianę z oknem o rozmiarze 7×7 .

Warto zwrócić uwagę, że filtracja medianowa dla obrazów binarnych sprowadza się do określenia, czy w otoczeniu rozważanego piksela przeważają piksele białe czy czarne. Na podstawie tej informacji przypisuje się nową wartość piksela.

Z uwagi na specyfikę późniejszej implementacji sprzętowej warto przypomnieć sobie o problemie obsługi pikseli brzegowych. Dla potrzeb niniejszego systemu (i wielu innych praktycznych rozwiązań) założymy, że jeśli nie dysponujemy pełnym kontekstem tj. rozpatrywany piksel leży blisko krawędzi obrazu to wyniki naszej filtracji wynosi '0'.

9.5 Wyznaczanie środka ciężkości

Na podstawie maski binarnej, system wizyjny nie ma możliwości stwierdzenia ile obiektów znajduje się na obrazie, albo jakie mają rozmiary. Informacja ta musi zostać wydobыта, poprzez

przetworzenie maski przy pomocy algorytmów indeksacji i analizy grup pikseli. W rozważanym przypadku zdecydowano się na jedną z najprostszych operacji analizy – wyznaczenie średniego położenia wszystkich pikseli maski należących do obiektu.

Warto w tym momencie powiedzieć, że implementacja indeksacji w potokowym systemie wizyjnym nie jest sprawą trywialną. W przypadku tzw. indeksacji jednoprzebiegowej trzeba zrealizować moduł analizujący parametry poszczególnych obiektów i ich łączenie. Natomiast typowy algorytm dwuprzebiegowy wymaga użycia pamięci RAM (zwykle zewnętrznej) i wprowadza do systemu opóźnienie (latencję) powyżej jednej ramki.

Do wyznaczenia wypadkowego położenia pikseli należących do poszukiwanego obiektu, wykorzystywane są wzory na środek ciężkości wyznaczone na podstawie momentów geometrycznych:

$$m_{00} = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_{ij} \quad (9.3)$$

$$m_{10} = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} i \cdot x_{ij} \quad (9.4)$$

oraz

$$m_{01} = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} j \cdot x_{ij} \quad (9.5)$$

gdzie: N i M to rozmiary obrazu, a x_{ij} to piksel (binarny) z maski. Na tej podstawie oblicza się środek ciężkości jako:

$$x_{sc} = \frac{m_{10}}{m_{00}} \quad (9.6)$$

$$y_{sc} = \frac{m_{01}}{m_{00}} \quad (9.7)$$

Wyznaczony w ten sposób punkt określa oczywiście środek ciężkości wszystkich pikseli oznaczonych jako elementy skóry na obrazie. Jednakże przyjmujemy optymistyczne założenie, że w kadrze znajduje się dokładnie jeden obiekt, a ewentualne błędy wyeliminuje filtracja medianowa.

9.6 Przykład działania

Przykład działania systemu zaprezentowano na rysunku 9.6. Zdjęcie ręki jest poddawane konwersji do YCbCr, progowaniu, filtracji medianowej. Ostatecznie wyznaczany jest środek ciężkości otrzymanego obszaru.



9.7 Zadania do wykonania w domu

Uwaga Wykonanie niniejszych zadań jest **warunkiem dopuszczenia** do realizacji laboratorium.

9.7.1 Model programowy

Zadanie 9.1 Proszę wykonać cyfrowe zdjęcie twarzy lub ręki. Obiekt powinien znajdować się w środku kadru i zajmować min 20% powierzchni. Dla ułatwienia, proszę zadbać o to, aby w kadrze nie było obiektów o kolorze zbliżonym do koloru skóry. Zdjęcie proszę przeskalać do rozdzielczości 720×560 .

Proszę napisać skrypt w programie Matlab/Octave, który:

1. wczyta zdjęcie z pliku,
2. dokona konwersji zdjęcia z formatu RGB na format YCbCr zgodnie z metodą opisaną w rozdziale 9.2. UWAGA. Proszę nie wykorzystywać do tego celu funkcji wbudowanej w środowisko Matlab/Octave. Proszę zrealizować odpowiednie równania bezpośrednio w środowisku Matlab/Octave.
3. dokona segmentacji obszaru twarzy (koloru skóry) – 9.2.
4. dokona filtracji maski – kontekstowa filtracja medianowa dla obrazów binarnych (funkcja `medfilt2`),
5. wyznaczy środek ciężkości położenia pikseli oznaczonych jako rejon twarzy – 9.5,
6. narysuje dwie linie (pionową i poziomą) przebiegające przez całą szerokość obrazu, których środek oznaczy wyznaczony punkt ciężkości położenia obszaru twarzy (polece-
nie `line`).

Zasadniczo efekt końcowy powinien być zbliżony do zaprezentowanego na rysunku 9.6.

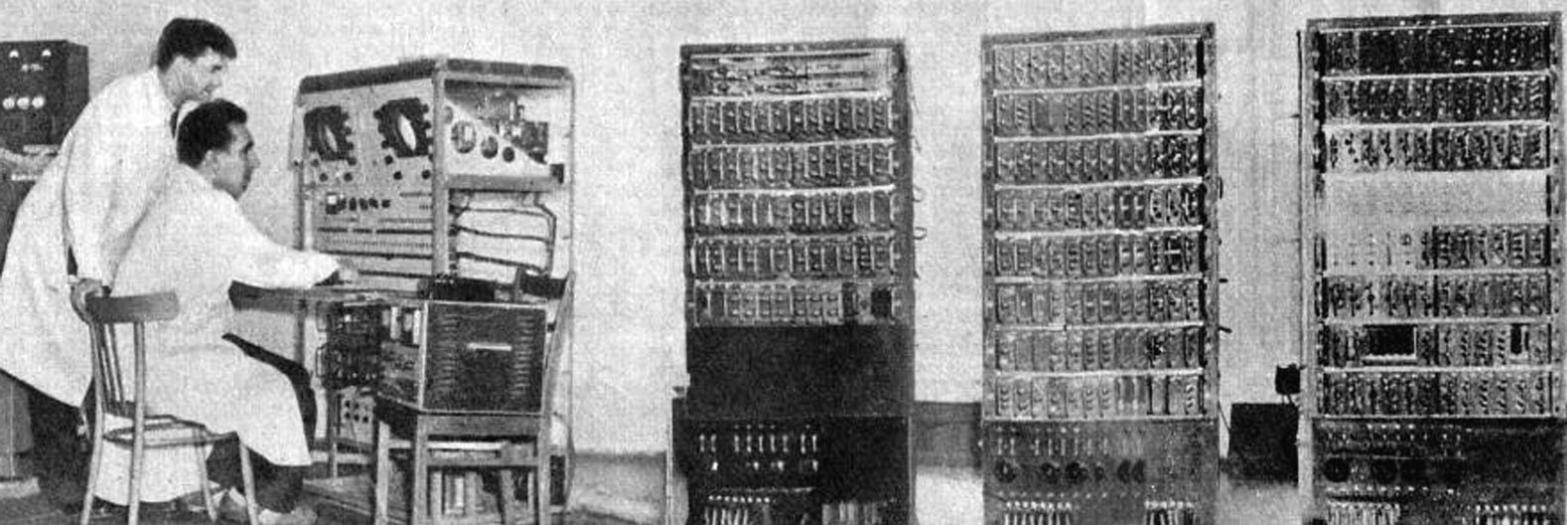
Proszę też przygotować wersję obrazu o rozdzielczości 64×64 w formacie ppm. ■

Uwaga. W systemie Windows otrzymanie poprawnego pliku *ppm* może być problematyczne. Można to zrobić używając Matlab'a. Wtedy trzeba dokonać "ręcznej" korekty pliku. Najlepiej w programie Notepad++ lub podobnym. Nagłówek powinien mieć postać:

P6LF
64 64LF
255LF

Przy czym kodowanie znaku końca linii UNIX, a nie WINDOWS (LF, a nie CRLF). W Notepad++ opcja *Edycja->Konwersja znaku końca linii*. Poprawność można sprawdzić wyświetlając wszystkie znaki (*Widok->Pokaż Niewidoczne Znaki*).

Alternatywnie można wykorzystać program IrfanView (zarówno do skalowania, jak i zapisu do ppm). Wtedy trzeba tylko "ręcznie" usunąć komentarz "# Created by IrfanView".



10 — Konwersja RGB do YCbCr

10.1 Model programowy

Zadanie 10.1 Proszę zaprojektować schemat architektury potokowej, do realizacji konwersji RGB na YCbCr. Proszę przeanalizować zakresy liczbowe, które muszą zostać wykorzystane na każdym etapie przetwarzania i oznaczyć je na schemacie, przy pomocy formatu X.Y – gdzie X część całkowita, Y – część ułamkowa.

Punktem wyjścia jest równanie 9.1, które należy "rozrysowywać". Model (rysunek) należy skonsultować z prowadzącym.

Zadanie 10.2 Wykonaj model programowy konwersji RGB -> YCbCr na liczbach stałoprzecinkowych.

Wykorzystując metodologię wprowadzoną w zadaniu 7.6 stwórz stałoprzecinkowy model programowy. Ustal precyzję (dla współczynników do mnożenia na 17+1 bitów), dla reszty (8+1 bitów – zakres [0:255]). Uwaga. Precyzja 18 bitów ze znakiem dla współczynników wynika z maksymalnego rozmiaru słowa obsługiwanej przez moduł DSP48 w układzie Spartan 6.

Uwaga. Zakładamy, że po operacji mnożenia **dokonujemy pominięcia części ułamkowej**. W Matlab'ie należy zastosować polecenie `quantize`. Kwantyzację należy przeprowadzić osobno na każdym wyniku mnożenia. Można to zrealizować np. za pomocą pętli `for`.

Wybierz przykładowy wektor testowy RGB. Zrealizuj konwersję RGB->YCbCr na liczbach `double`, a następnie stałoprzecinkowych. Wyniki porównaj. Z czego wynikają różnice?

10.2 Implementacja sprzętowa

Zadanie 10.3 Zaimplementuj sprzętowo schemat obliczeń zaproponowany w zadaniu 10.1. Przetestuj jego działanie symulacyjnie – zarówno sam moduł, jak i w ramach modelu toru wizyjnego.

Wskazówki:

1. Rozwijamy moduł zrealizowany na poprzednim ćwiczeniu tj. tor wizyjny HDMI (por. rozdział 8).
2. Zaczynamy od tworzenia modułu nadzawanego np. `rgb2ycbcr.v`. Jego wejścia to zegar,

sygnały synchronizacji i piksel (RGB), a wyjścia to odpowiednio opóźnione sygnały synchronizacji oraz piksel YCbCr.

3. Następnie trzeba zaimplementować mnożenia. Realizujemy je z użyciem modułów DSP48 dostępnych w układzie Spartan 6. Należy wygenerować IP CORE typu *Multiplier*. W konfiguratorze ustawiamy:

- strona 1 – rozmiar na 18 bitów i typ *Signed*,
- strona 2 – opcja *Use Mults*,
- strona 3 – *Pipeline Stages* – tak jak pokazuje *Optimum Pipeline Stages*.

Kilka słów wyjaśnienia. Po pierwsze realizujemy mnożenie na dedykowanych mnożarkach sprzętowych, a nie w logice (LUT). Wykorzystujemy pełną precyzję mnożarki (18 bitów w przypadku liczb ze znakiem). Staramy się zapewnić maksymalną częstotliwość pracy ustalając optymalną wartość latencji. Ostatecznie jedno mnożenie powinno być realizowane z wykorzystaniem jednego elementu DSP48.

Wstawiamy 9 instancji mnożarki. Ułatwienie – *View HDL Instantiation Template*. Do każdej z nich trzeba doprowadzić wartość R,G lub B. Pomocna może być składnia $\{10'b0, R\}$, czyli konkatenacja dwóch wektorów bitowych. Przypomnienie. Oba argumenty mnożenia są 18-bitowe. Drugi argument to stała. Najprościej ją uzyskać z modelu w pakiecie Matlab używając polecenia `b1n(X)`, gdzie X to liczba stałoprzecinkowa (w odpowiedniej precyzyji). Wyjście z modułu należy zapisać do uprzednio stworzonych połączeń. Zaleca się zastosować składnię `wire signed`.

4. Kolejny krok to realizacja dodawania. Ponieważ docelowe wartości liczb są z zakresu [0:255], nie zachodzi konieczność zwiększenia zakresu ponad 9 bitów (8 + znak). Wykorzystaj moduł IP CORE. Ustaw ich latencję na *Automatic* (tj. 2). Argumentami operacji dodawania są wyniki mnożenia. Aby pominąć część ułamkową należy wybrać odpowiednie bity (dokładnie 9 odpowiednich bitów). Dla składowej Y potrzebujemy dwa sumatory i jeden moduł opóźniający (należy wykorzystać ten stworzony w ramach zadania 5.4). Dla składowych *Cb*, *Cr* potrzebujemy po 3 sumatory.

Uwaga. Należy stworzyć wszystkie potrzebne sygnały (`wire`).

Uwaga. Struktura, która powstanie będzie dość złożona. Warto zwrócić szczególną uwagę na poprawne nazewnictwo połączeń. Pozwoli to uniknąć błędów. Ponadto trzeba uważać stosując metodę kopijuj – wklej. Jej wykorzystanie jest częstą przyczyną błędów, gdyż dość łatwo zapomnieć zmienić jakąś nazwę. Dodatkowo, tego typu błąd zazwyczaj nie jest prosty do wykrycia.

5. W ostatnim kroku trzeba dodać opóźnienie sygnałów synchronizacji. Można ponownie wykorzystać moduł opóźniający. Trzeba tylko wyznaczyć globalną latencję modułu konwersji. Warto zastosować konkatenację sygnałów (wejściowych i wyjściowych) – `{de,hsync,vsync}`. Powoli to zastosować jeden, a nie trzy moduły i zaoszczędzić nieco pisania (kopowania) kodu.
 6. Prosty tesbench. Po opisaniu modułu warto zrobić szybki test, dla tej samej wartości co w modelu programowym. Na tym etapie należy wyeliminować wszystkie błędy – wynik musi być identyczny.
 7. Symulacja toru wizyjnego. Wykonany moduł należy wstawić do modelu toru wizyjnego. Należy także dodać do folderu własny plik *ppm* i ustawić jego nazwę w module *file_input*.
 8. Ostatnim etapem będzie porównanie wyników modelu programowego i symulacji dla całego obrazka. W tym celu należy zmodyfikować model stałoprzecinkowy (Matlabowy) w taki sposób, aby można było przetworzyć obraz. Oczywiście ma to być własny plik *ppm* ze zdjęciem ręki lub twarzy.
- Zasadniczo, zrealizowane przetwarzanie należy obudować pętlą po wszystkich pikselach na obrazie. Następnie należy wczytać wyniki symulacji – plik *out_*. Do porównania można

użyć funkcji `imabsdiff`. Oczywiście, oczekujemy, że nie będzie żadnych różnic. Uwaga. Obliczenia, nawet dla tak niewielkiego obrazka (64×64) mogą trwać chwilę. Jeśli będziemy mieli pewność, że wyniki modelu i symulacji ISim są zgodne, to warto zapisać obraz w przestrzeni YCbCr z Matlab'a (funkcja `imwrite`). Wykorzystamy go później.

10.3 Uruchomienie na karcie ATLYS

Sprawdzamy działania zaprojektowanego modułu na karcie ATLYS tj. dokonujemy weryfikacji na platformie sprzętowej.

W tym celu, proponowany jest system zaprezentowany na rysunku 10.1. Dane przychodzące z kamery, są odbierane przy pomocy portu HDMI. Układ FPGA jest odpowiedzialny za ich odbiór i deserializację. Następnie strumień wideo jest kierowany do modułu `rgb2ycbcr`, w którym dokonywana jest konwersja przestrzeni barw. Aby umożliwić przełączanie wejścia pomiędzy oryginalnym i przekształconym strumieniem wideo, wykorzystywany jest multiplekser sterowany przy pomocy przełączników SW0-SW2, znajdujących się na karcie ATLYS. Wybrany strumień jest następnie serializowany w układzie FPGA i przesyłany na wyjście HDMI, które powinno być podłączone do monitora aby umożliwić oglądanie wyników przetwarzania.

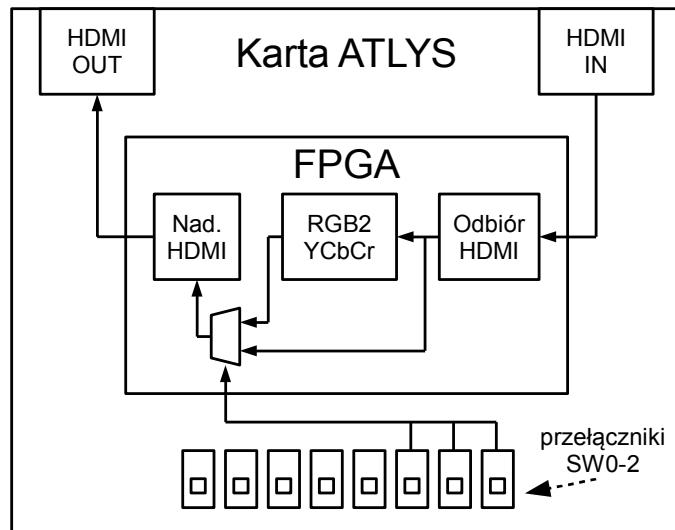


Figure 10.1: Proponowany system wizyjny

Zadanie 10.4 Uruchom zaprojektowany moduł na karcie Atlys. Dodaj multiplekser. ■

Podpowiedź:

- wstawienie modułu należy przeprowadzić analogicznie jak w przypadku LUT (por. rozdział 8).
- multiplekser jest trochę bardziej złożony. Na początku należy zdefiniować tablicę dla sygnałów R,G,B oraz synchronizacji:

```
wire [7:0] r_mux[7:0];
wire [7:0] g_mux[7:0];
wire [7:0] b_mux[7:0];
wire de_mux[7:0];
```

```
wire hsync_mux[7:0];
wire vsync_mux[7:0];
```

W ten sposób możemy np. do *r_mux* przypisać 8 różnych wartości.

Następnie do wybranych pozycji w tablicy przypisujemy odpowiednie sygnały (składowe i synchronizację). W naszym przypadku mamy dwa (RGB i YCbCr). W ramach dalszych prac multiplekser rozbudujemy.

W sekcji "*HDMI output port signal assigments*" wykorzystujemy stworzone tablice, przy czym wybór elementu uzależniamy od układu przełączników. Ponieważ maksymalnie zakładamy 8 możliwości to wystarczą 3 przełączniki. W nagłówku modułu należy odkomentować i ew. przedefiniować rozmiar sygnału *SW* – ma być [2:0]. Należy również sprawdzić plik *ucf*.

- projekt należy uruchomić i sprawdzić na karcie Atlys.

10.4 Implementacja progowania

Następnym etapem jest progowanie obrazu w przestrzeni YCbCr – z wykorzystaniem tylko chrominancji. Moduł, który jest proponowany do realizacji powyższej operacji, został przedstawiony na rysunku 10.2. Składa się on z dwóch identycznych bloków, odpowiadających za porównanie odpowiednio kanałów Cb i Cr. Każdy z nich jest zbudowany z dwóch komparatorów oraz bramki AND. Wynik porównań dla obu kanałów jest również łączony przy pomocy bramki AND. Rezultat jest wykorzystywany jako sygnał sterujący do multipleksera. W zależności od tego, czy piksel należy do wykrywanego obiektu czy nie, wybierany jest kolor biały (RGB = 255, 255, 255) lub czarny (RGB = 0, 0, 0). Dodatkowo, wejściowe sygnały synchronizujące (*de*, *hsync*, *vsync*) są opóźniane o tyle cykli zegara, ile wynosi łączna latencja wprowadzana przez bloki dokonujące binaryzacji. Dzięki temu, porty wyjściowe modułu mogą zostać bezpośrednio podłączone do wejść nadajnika HDMI, co umożliwia ocenę działania zaprojektowanego modułu na podłączonym monitorze LCD.

Zadanie 10.5 Zrealizuj moduł progowania. Przetestuj go symulacyjnie i na karcie Atlys. ■

Podpowiedzi:

- w modelu symulacyjnym należy wykorzystać zapisany poprzednio (rozdział 10.2) obraz YCbCr. Tylko w ten sposób będziemy mieli gwarancję, że uzyskana maska będzie identyczna zarówno w modelu symulacyjnym, jak i w sprzęcie. Jest to istotne w dalszych pracach, przy wyliczaniu środka ciężkości.
- nowy moduł(*thresholding*) ma mieć następujący interfejs: wejścia – zegar, *cb* i *cr*, sygnały synchronizacji oraz parametry (*Ta*, *Tb*, *Tc*, *Td*), wyjścia – nowy piksel (*rgb*) oraz synchronizacja,
- parametry (*Ta*, *Tb*, *Tc*, *Td*) należy ustawić w instancji modułu. Można sobie wyobrazić rozbudowę aplikacji o zmienianie tych parametrów podczas działania systemu (tj. on-line) przykładowo poprzez UART.
- przygotowany moduł należy przetestować symulacyjnie. Warto sobie stworzyć "multiplekser z komentarzy" w module *tb_hdmi*, czyli zakomentować, a nie kasować poprzednie przypisania (YCbCr) (sekcja *Output assigment*).
- następnie moduł należy wstawić w tor przetwarzania obrazów (*hdmi_main*). Jeśli dobrze dobraliśmy nazwy, to wystarczy skopiować instancję z modułu testowego i zmienić nazwę zegara. Należy też podłączyć nowe wyjście do multipleksera.
- system należy zaimplementować i zweryfikować na karcie Atlys.

10.5 Zadania dodatkowe

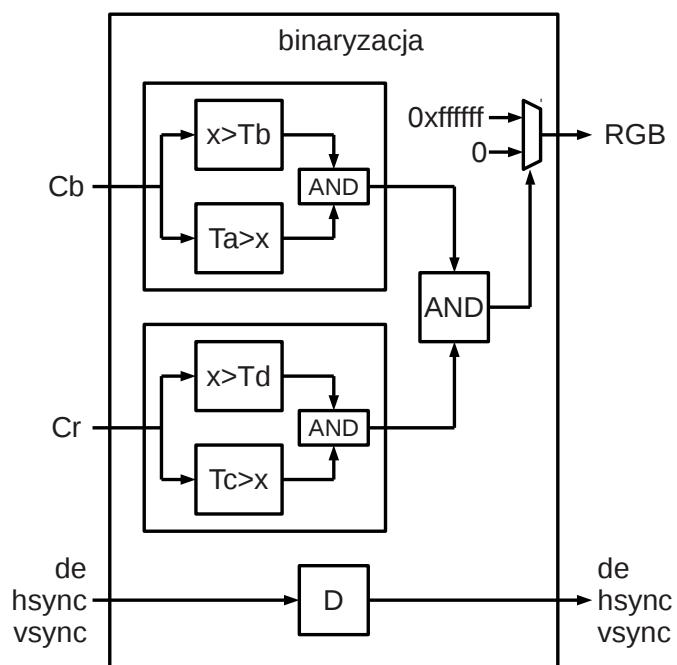


Figure 10.2: Schemat modułu dokonującego binaryzacji

Zadanie 10.6 Zaimplementuj konwersję z przestrzeni barw RGB do HSV. Wykorzystaj następujące przekształcenie (takie samo występuje w bibliotece OpenCV i pakiecie Matlabie):

1. Wejściowe składowe R,G,B z zakresu [0;255] konwertowane są na liczby z zakresu [0:1].

2. Następnie składowe H,S,V obliczane są zgodnie z zależnościami:

$$V = \max(R, G, B) \quad (10.1)$$

$$S = \begin{cases} \frac{V - \min(R, G, B)}{V} & \text{if } V \neq 0 \\ 0 & \text{else} \end{cases} \quad (10.2)$$

$$H = \begin{cases} 0 & \text{if } V - \min(R, G, B) = 0 \\ 60(G - B)/(V - \min(R, G, B)) & \text{if } V = R \\ 60(B - R)/(V - \min(R, G, B)) + 120 & \text{if } V = G \\ 60(R - G)/(V - \min(R, G, B)) + 240 & \text{if } V = B \end{cases} \quad (10.3)$$

Jeśli $H < 0$ to $H = H + 360$. Następnie $H = H/360$.

W efekcie wszystkie liczby są z zakresu [0;1]. W ostatnim etapie należy dokonać ich przeskalowania do zakresu [0;255] (ten krok nie jest już zgodny z OpenCV lub Matlab'em ale umożliwia poprawne wyświetlanie na ekranie).

Uwaga wstępna. Zadanie jest trudne i wymaga sporo pracy. Jednakże pozwala bardzo dokładnie zapoznać się z realizacją bardziej złożonych operacji arytmetycznych, w tym z dzieleniem. Uczy także ścisłej współpracy z modelem programowym.

Uwaga. Sugerowane rozwiążanie należy traktować jako przykładowe i jedno z możliwych.

Wskazówki i przebieg implementacji:

1. W pierwszym kroku należy przeanalizować wzory na konwersję RGB -> HSV i wypisać potrzebne moduły: Zatem potrzebujemy:

- 3 elementy konwersji z przedziału [0;255] do [0:1] – dzielarki przez 255,
- moduł do obliczania maksimum i minimum ze składowych R,G,B
- moduł do obliczania różnicy pomiędzy maksimum, a minimum ($C = V - \min(R, G, B)$),
- moduł do obliczania różnic: $G - B, B - R, R - G$,
- moduł mnożący przez 60,
- moduł dodający 0, 120 lub 240,
- moduł dodający 360,
- moduł dzielący przez 360,
- moduł realizujący dzielenie C/V – obliczanie wartości składowej S,
- 3 moduły mnożące przez 255,
- szereg multiplekserów oraz linii opóźniających.

2. Obliczenie składowych S i V jest dość proste. Najwięcej problemów sprawia składowa H.

Jej ostateczna wartość wybierana jest jako jedna z czterech możliwości (por. równanie (10.3)). Przy czym wymagane operacje tj.: odejmowanie, dzielenie, mnożenie przez 60 i dodawanie są identyczne. Zatem istnieją dwa rozwiązania: równolegle prowadzenie wszystkich obliczeń lub multipleksacja argumentów poszczególnych operacji. W tym przypadku rekomenduje się drugie podejście, gdyż pozwala ono na ewidentną oszczędność w zasobach (dzielenie jest bardzo niewdzięczne do realizacji w FPGA).

3. Pracę zaczynamy od implementacji programowego modelu referencyjnego. Możemy go zrealizować w Matlabie lub ew. innym języku programowania z biblioteką do obliczeń stałoprzecinkowych. W pierwszym kroku realizujemy dzielenie składowych R,G,B przez 255. Niestety pakiet Fixed-Point z programu Matlab nie wspiera operacji dzielenia liczb stałoprzecinkowych. Trzeba zatem znaleźć obejście. Przykładowo można wykorzystać następującą składnię:

```
R = R / 255;
R_FIX = fi(R,1,word,prec_f,'RoundingMethod','Floor');
```

Tutaj dzielenie realizujemy na liczbach zmienoprzecinkowych, a wynik konwertujemy na stały przecinek. Ważne jest ustalenie metody obsługi kwantyzacji na obcięcie tj. parametr '*Floor*'.

Następnie obliczamy maksimum i minimum oraz ich różnicę (parametr C). W tym miejscu następuje rozdzielenie obliczeń dla składowych H,S,V.

4. Najprostsza jest składowa V – maksimum z kanałów R,G,B wyznaczone w poprzednim etapie. Jedyną dodatkową operacją jest mnożenie przez 255:

```
V_255 = floor(double(M_FIX) * 255);
```

Warto zwrócić uwagę, że trzeba zastosować stałoprzecinkową wersję maksimum – dla kompatybilności z późniejszą implementacją FPGA.

5. Obliczenie składowej S jest również dość proste. Uwaga. Podobnie jak wcześniej wymagane dzielenie należy zrealizować na liczbach zmienoprzecinkowych (*double*), a następnie wyniki przekonwertować na format stałoprzecinkowy.

6. W przypadku składowej H warto od razu przygotować model, który będzie działał podobnie jak planowany moduł sprzętowy. Zatem mając dane V (maksimum z R,G,B) realizujemy multipleksację do dwóch argumentów modułu odejmującego zgodnie z warunkami z równania (10.3). Wartości odejmujemy. Następnie realizujemy dzielenie przez wartość C. Ponownie konieczna jest konwersja na format zmienoprzecinkowy, dzielenie i powrót do formatu stałooprzecinkowego:

```
A1_A2_c = double(A1_A2_FIX)/C;
A1_A2_c_FIX = fi(A1_A2_c,1,word,prec_f,'RoundingMethod','Floor');
```

Następnie mnożymy przez 60 (wcześniej należy liczbę 60 zamienić na format stałooprzecinkowy). Dalej realizujemy drugi multiplekser – zgodnie z równaniem (10.3). Jeśli $C = 0$ to dalej ustalamy $H = 0$. W innym przypadku zgodnie z warunkami logicznym dodajemy odpowiednio 0, 120 lub 240. Jeśli otrzymana wartość H jest mniejsza od zera to dodajemy do niej 360. Dalej realizujemy dzielenie przez 360 – ponownie trzeba zastosować "patent" z obliczeniami zmienoprzecinkowymi i konwersją. Ostatni etap to mnożenie przez 255.

7. Stworzony model należy sprawdzić. Na początku warto skupić się na jednym lub kilku wektorach testowych. Referencję stanowić będą wyniki zwracane przez funkcję `rgb2HSV` pomnożone przez 255. Różnica pomiędzy wynikami zwracanymi przez funkcję Matlaba i modelem nie powinna być większa niż 1-2 (różnica na poszczególnych składowych pikseli). Drugi test przeprowadzamy na całym obrazie. Przykładowo na używanym w poprzednich ćwiczeniach `geirangerfjord_64.ppm`. Wizualnie obrazy nie powinny się różnić.

Uwaga. Etap poprawnej implementacji modelu programowego jest bardzo ważny. Jeśli tutaj popełnione zostaną błędy, to później przeniesione zostaną one do realizacji sprzętowej. I wtedy konieczna będzie analiza zgodności: implementacji bazowej z modelem, modelu z wersją FPGA, a także wersji FPGA z bazową. Przy czym doświadczenie wskazuje, że najtrudniej jest się zorientować, że popełniony został błąd. Dlatego implementację sprzętową warto rozpocząć dopiero wtedy gdy mamy pewność co do poprawności naszego modelu programowego.

8. Realizowany moduł sprzętowy jest dość złożony, zatem sugeruje się jego tworzenie etap po etapie i weryfikację poprawności z modelem programowym. Inaczej wyszukiwanie błędów może okazać się dość czasochłonne. Poniżej zmieszczone są istotne wskazówki związane z implementacją:

- (a) Pierwszy potrzebny moduł to dzielarka przez 255. Wykorzystujemy IP Core *Divider*. Jako *Remainder Type* ustawiamy *Fractional* – interesuje nas część ułamkowa wyniku, a nie reszta z dzielenia. Zakładamy, że reprezentacja 8-bitowa części ułamkowej nas satysfakcjonuje. Na tym etapie jeszcze nie wprowadzamy znaku. Należy zapamiętać/zapisać latencję.

Uwaga. Bardzo cennym przy tego typu złożonych projektach jest rysowanie schematu modułów wraz z latencją. Pozawala to "panować" nad systemem.

Wynik dzielenia trzeba odpowiednio użyć. Po pierwsze dodajemy znak (od tej pory prowadzimy obliczenia na liczbach ze znakiem), po drugie obsługujemy przypadek gdy dzielna równa się 255 (tj. wyniki dzielenia wynosi 1):

```
wire signed [9:0] r_01;
assign r_01[9] = 1'b0; // bit znaku -- tu na 0
assign r_01[8] = q_r[0]; // czesc calkowita wyniku
assign r_01[7:0] = f_r; // czesc ułamkowa wyniku
```

- (b) Drugi etap to wyszukiwanie maksimum i minimum oraz obliczanie ich różnicy. Moduły do obliczania dwóch pierwszych wartości należy opisać w Verilogu. Powinny one, oprócz wartości maksymalnej/minimalnej, zwracać "pozycję" tego ekstremum (tj. 0,1,2). Ułatwi to realizację późniejszych multiplekserów. Najprostsza implementacja to seria `if else` w procesie.
- Do obliczania różnicy należy wykorzystać odpowiedni IP Core.
- (c) Mając obliczone maksimum, mamy już wartość V. Trzeba będzie ją tylko odpowiednio opóźnić i pomnożyć przez 255 (IP Core). Dobór opóźnienia trzeba zostawić jednak na koniec (po implementacji obliczania H).
- (d) Obliczanie wartości S również nie jest bardzo skomplikowane. Argumentami operacji dzielenia są C i V. Trzeba zapewnić tylko ich odpowiednią synchronizację tj. opóźnić V. Następnie trzeba zrealizować dzielenie – moduł IP Core. Należy też zapewnić mechanizm zwracania zera w przypadku gdy V = 0. Przykładowo można to zrealizować za pomocą instrukcji `assign`:

```
assign S_01 = (V_del_S > 0) ? {S_01_q[0], S_01_f[8:1]} : 10'b0;
```

Jeśli warunek jest prawdziwy to przypisuje się wynik dzielenia, a jeśli nie to 0. Podobnie jak w przypadku V, uzyskaną wartość S należy opóźnić i pomnożyć przez 255 (kolejność dowolna). Długość opóźnienia dobrana zostanie później.

- (e) Najbardziej złożone jest obliczanie składowej H. Na początku trzeba zrealizować multiplekser, który pozwoli wybrać argumenty do operacji odejmowania. Można wykorzystać koncepcję podobną do zaprezentowanej w zadaniu 10.4. Następnie tworzymy moduł odejmujący (IP Core) i przypisujemy argumenty. Multiplekserem steruje pozycja maksimum. Warto zwrócić uwagę, że przed multiplekserem trzeba wprowadzić opóźnienie dla R,G,B, tak aby sygnał o pozycji maksimum był zsynchronizowany z danymi.

Kolejny etap to realizacja dzielenia przez C. Należy wykorzystać odpowiedni IP Core. Problematyyczna jest interpretacja wyniku. Dzielarka dla argumentów ze znakiem zwraca część całkowitą i ułamkową ze znakiem (osobno). Taką liczbę trzeba "zespolić". Przykładowe rozwiązanie:

```
wire signed [10:0] res_A1_A2_c;
assign res_A1_A2_c[10] = (res_A1_A2_c_q[10] | res_A1_A2_c_f[9]);
                           // bit znaku
assign res_A1_A2_c[9] = (res_A1_A2_c_f[9] == 1'b0 & res_A1_A2_c_q
                           [10] == 1'b0) ? 1'b0 : 1'b1;           // bit całkowity
assign res_A1_A2_c[8] = (res_A1_A2_c_f[9] == 1'b0 & res_A1_A2_c_q
                           [10] == 1'b0) ? res_A1_A2_c_q[0] : 1'b1; // bit całkowity
assign res_A1_A2_c[7:0] = res_A1_A2_c_f[8:1];
                           // bity ułamkowe
```

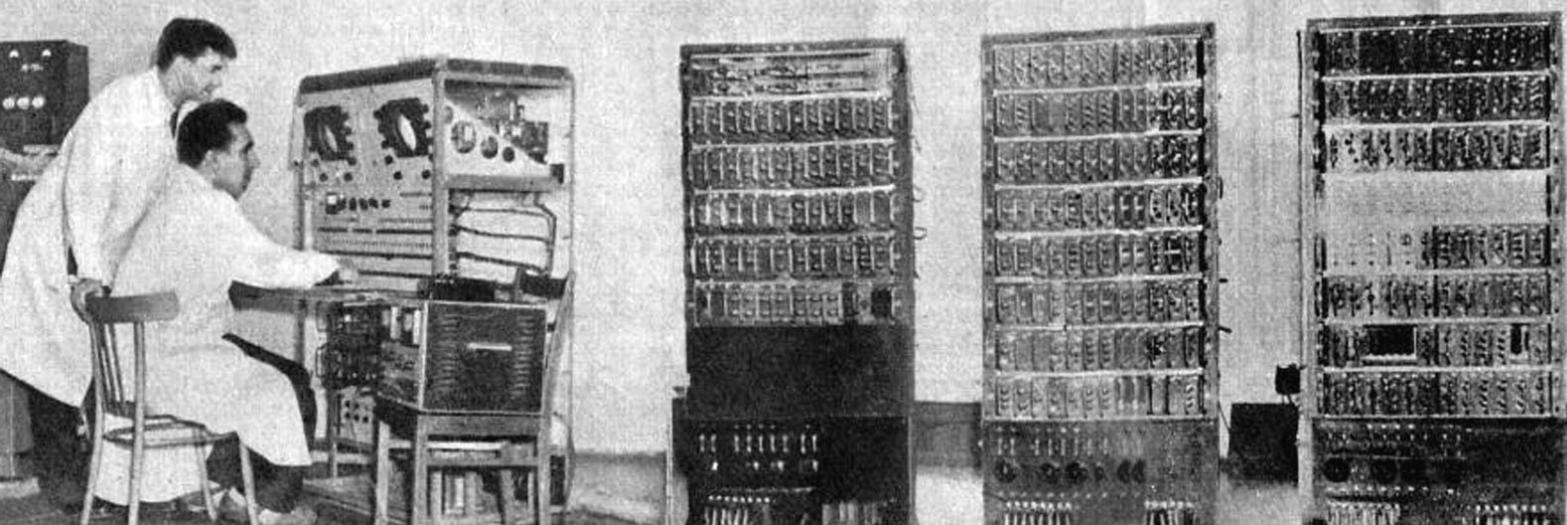
Uwaga. Prostsze rozwiązania mile widziane.

Dalej mnożenie przez 60 (IP Core) oraz kolejny multiplekser. Tutaj trzeba do wyniku mnożenia dodać 0, 120 lub 240. Wykorzystujemy podejście podobne jak wcześniej. Działaniem multipleksera steruje również pozycja maksimum – przy czym trzeba ją odpowiednio opóźnić.

Następnie trzeba wykonać warunkowe dodawanie 360. Równolegle realizujemy dodawanie i opóźnienie wartości niezmienionej. Ostatecznego wyboru dokonujemy na podstawie znaku wyniku. Na tym etapie należy także sprawdzić czy nie zaszło dzielenie przez 0 tj. warunek (C=0) – jeśli tak to zwrócić wartość 0.

Ostatnie dwie operacje to dzielenie przez 360 i mnożenie przez 255.

- (f) Po realizacji toru obliczeń dla składowej H możemy wyznaczyć latencję całego przetwarzania oraz potrzebne do opóźnienia sygnałów S,V wartości. Warto sumiennie uzupełniać schemat blokowy modułu. Należy również zapewnić opóźnienie sygnałów synchronizacji: *de*, *hsync*, *vsync*.
 - (g) Gotowy moduł trzeba przetestować. Uwaga. Bardzo istotne są testy po poszczególnych etapach. Dobra rada to robić je na "bieżąco" – po dodaniu kolejnego modułu. Opisany tutaj test finalny powinien tylko potwierdzić poprawność implementacji. Testujemy na trzech poziomach: konwersja jednej trójki R,G,B, konwersja serii trójek (pozwala wykryć błędy z synchronizacją), test na obrazie – podobnie jak we wcześniejszych zadaniach. Różnica pomiędzy przetwarzaniem FPGA, a modelem nie powinna być większa niż 1.
9. Ostatni etap to uruchomienie modułu w torze przetwarzania sygnału HDMI na karcie Atlys. Uwaga implementacja modułu może trwać dość długo – ma to związek z użyciem kilku mnożarek.



11 — Wyznaczanie środka ciężkości oraz wizualizacja

11.1 Wyznaczanie środka ciężkości

Wyznaczenie środka ciężkości zostanie zrealizowane w oparciu o wzory (9.3) – (9.7).

Implementacja rozwiązania w architekturze sprzętowej wymaga stworzenia modułu, którego schemat został przedstawiony na rysunku 11.1.

Wykorzystywane są trzy liczniki: kolumn, rzędów i końca ramki, sterowane na podstawie sygnałów synchronizacji (*de*, *hsync*, *vsync*). Pozwalają one określić współrzędne każdego piksela na obrazie (*x*, *y*) oraz wyznaczyć moment, w którym przetworzono ostatni piksel aktualnej ramki obrazu (sygnał *e* powinien zostać ustawiony na 1 na jeden cykl zegara).

Wartości *x*, *y*, *eof* oraz maska binarna są wykorzystywane jako sygnały sterujące modułami wyznaczającymi współrzędne środka ciężkości. Wartość maski binarnej jest podpięta do wejścia *ce* w liczniku i rejestrze akumulującym. Dzięki temu, będą one pracowały jedynie w przypadku, gdy maska binarna ma wartość 1 (piksel należy do poszukiwanego obiektu). Sygnał końca ramki jest wykorzystywany jako sygnał *rst* i umożliwia wyzerowanie wartości licznika i rejestru akumulującego przed przystąpieniem do przetwarzania kolejnej ramki obrazu. Do wyznaczenia liczby wszystkich pikseli należących do obiektu 9.3 stosowany jest wspólny licznik.

Natomiast do wyznaczenia momentów *m01* i *m10* (równania: (9.5), (9.4)) wykorzystywany jest sumator o latencji równej 0 i rejestr (porównaj zadanie 7.3). Po wyznaczeniu wartości momentów, należy wykonać dzielenie. Wykorzystana zostanie tutaj dzielarka iteracyjna (moduł dostępny na stronie kursu). Modułu IP Core skutkowałbym w tym przypadku długim czasem implementacji projektu, a w pełni potokowe dzielenie nie jest tutaj potrzebne. Dzielarka uruchamiana poprzez podanie 1 logicznej na jeden takt zegara na wejście *s* (start). W naszym przypadku będzie to sygnał końca obrazu *e*.

Na obrazie wynikowym wyznaczony środek ciężkości zostanie zwizualizowany za pomocą dwóch kresek – pionowej i poziomej.

Implementacja tego modułu (*visualize*) jest bardzo prosta. Należy wykorzystać liczniki kolumn i rzędów – identyczne jak przy obliczaniu środka ciężkości, a następnie sprawdzić czy współrzędne aktualnie przetwarzanego piksela pokrywają się ze współrzędnymi środka ciężkości. Ponieważ chcemy otrzymać linii to tylko jedna ze współrzędnych musi się zgadzać (warunek OR). Jeśli warunek jest spełniony to modyfikujemy zawartość piksela np. ustawiamy jego kolor na czerwony. Jeśli nie to pozostawiamy piksel bez zmian.

Warto się jeszcze zastanowić jak należy połączyć moduły wyznaczania środka ciężkości

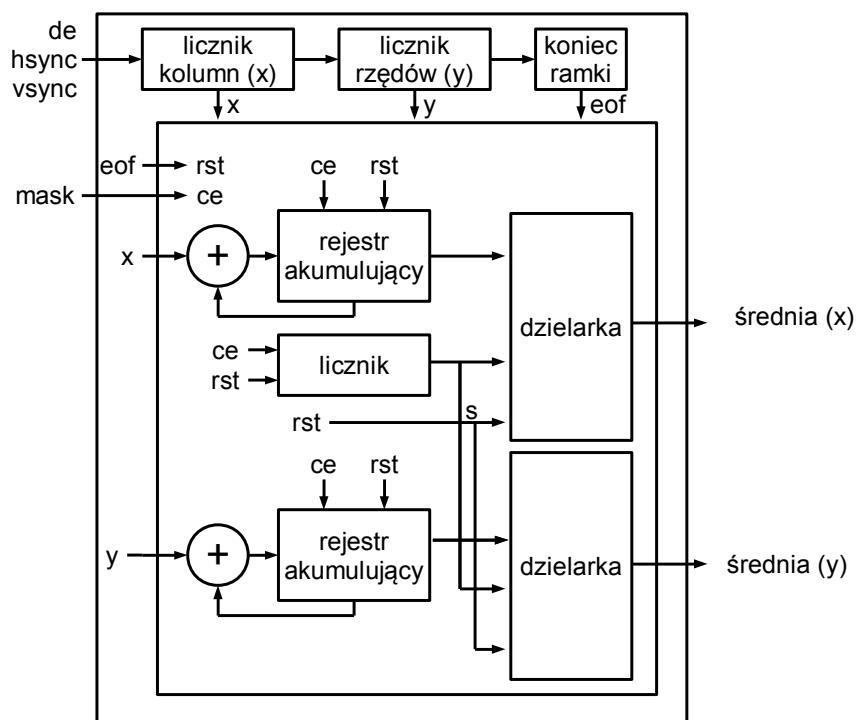


Figure 11.1: Schemat modułu do wyznaczania środka ciężkości

i wizualizacji. Prosta realizacja koncepcji przedstawionej na rysunku 10.1 ma pewną wadę. Wyznaczanie środka ciężkości trwa w zasadzie całą ramkę – konieczne jest przetworzenie wszystkich pikseli. Zachowanie potokowości wymagałoby opóźniania maski oraz sygnałów synchronizacji o niemal jedną klatkę. Jest to oczywiście możliwe, ale niekonieczne sensowne i ekonomiczne.

Można za to zastosować pewne "oszustwo" polegające na tym, że środek ciężkości wyznaczony dla poprzedniej ramki będzie wyświetlany na obrazie bieżącej. Przy częstotliwości 25/30 ramek na sekundę raczej nie będziemy w stanie zauważać tej drobnej rozbieżności. Koncepcję ilustruje rysunek 11.2.

11.2 Zadania do wykonania na laboratorium

Zadanie 11.1 Proszę zaimplementować moduły do wyznaczania środka ciężkości oraz wizualizacji opisane w rozdziale 11.1 oraz przeprowadzić ich weryfikację symulacyjną oraz w sprzętce.

Wskazówki:

- Pracę zaczynamy od stworzenia modułu wyznaczania środka ciężkości (*centroid.v*). Wejścia: *clk*, *ce*, *rst*, *de*, *hsync*, *vsync*, *mask* (bitowe). Wyjścia: współrzędna x i y środka ciężkości. Zakładamy, że maksymalna rozdzielcość przetwarzanego obrazu będzie 720×576 . Wykorzystując tę informację należy dobrać rozmiar wektora. Ponadto potrzebne są dwa parametry: *IMG_H* i *IMG_W* – odpowiednio wysokość i szerokość obrazu.
 - Na początku stworzymy liczniki, które pozwolą określić aktualną pozycję na obrazie. Wewnatrz procesu (*always*) sprawdzamy czy sygnał *vsync* wynosi 0 – jeśli tak to liczniki

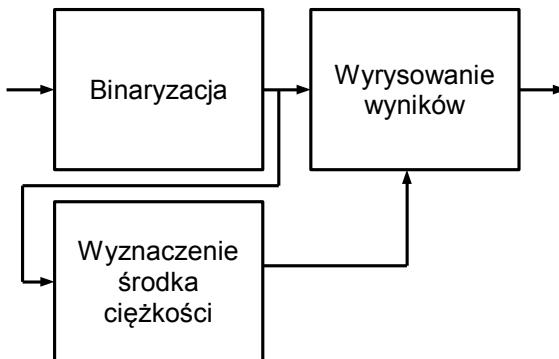


Figure 11.2: Współpraca modułu obliczania środka ciężkości i wizualizacji

zerujemy, jeśli nie to sprawdzamy czy sygnał *de* (tj. poprawność piksela) jest równy 1. Jeśli tak to inkrementujemy licznik horyzontalny. Sprawdzamy czy nie osiągnął on wartości szerokości obrazu (*IMG_W - 1* z uwagi na specyfikę działania przypisania \leq). Jeśli tak to go zerujemy i inkrementujemy licznik wertykalny. Jeśli licznik wertykalny osiągnął wartość wysokości obrazu (*IMG_H - 1* z uwagi na specyfikę działania przypisania \leq) to go zerujemy. Oczywiście liczniki należy wcześniej zdefiniować jako rejestrzy o odpowiedniej szerokości (wyzerować).

Uwaga. Zaprezentowana została tylko jedna przykładowa możliwość implementacji. Równie dobrze można oprzeć się tylko i wyłącznie na przebiegach sygnałów synchronizacji – wtedy nie jest potrzebne określanie rozmiarów obrazu.

- Wykrycie końca ramki (*eof – end of frame*) można zrealizować albo na podstawie powyższych liczników, albo na postawie sygnału *vsync*. W tym drugim przypadku należy wykryć jego opadające zbocze np. rejestrując poprzednią wartość. Ponieważ potrzebujemy impulsu (sygnał trwający jeden takt) to możemy wykorzystać następującą składnię:
`assign eof = (prev_vsync == 1'b1 & vsync == 1'b0) ? 1'b1 : 1'b0;`
- Następny element to obliczanie wartości *m00* – do wykonania za pomocą prostego licznika zerowanego sygnałem *eof* (*rst = eof*). Rozmiar rejestru należy określić na podstawie maksymalnej rozdzielczości obrazu.
- Implementacja obliczania momentów *m01* i *m10* opiera się o moduł akumulatora z zadania 7.3. Składa się on z sumatora o latencji 0 oraz procesu synchronizującego dodawanie. Szczegóły w opisie zadania. Jedyną modyfikacją jest rozmiar akumulatora. Należy się zastanowić jaki on powinien być. Można zastosować taki sam moduł dla obu momentów, gdyż oszczędność zasobów w przypadku realizacji różnych będzie nieznaczna, a pracy więcej. Przy wstawianiu instancji warto wrócić uwagę na potrzebę opóźnienia sygnałów *eof* i *mask*, tak aby były one zgodne z licznikami.
- Do dzielenia należy wykorzystać dostępny na stronie kursu moduł *divider_28_20*. Jest to implementacja dzielenia za pomocą kolejnych aproksymacji. Moduł składa się z mnożarki i maszyny stanowej. W skrócie jego działanie opiera się na następującej zasadzie. Na początku wynik (*sar*) jest zerowany. Następnie, w kolejnych iteracjach (ich liczba zależy od długości dzielnika), na '1' ustawiany jest odpowiedni bit rejestru *sar* – zaczynając od najstarszego. Jest on mnożony przez dzielną (*divisor*), a rezultat operacji porównywany z dzielnikiem. Jeśli jest od niego mniejszy to '1' na tej pozycji zostaje. Jeśli nie, co oznacza, że proponowana liczba jest za duża, analizowany bit jest zerowany. Przechodząc,

w ten sposób przez cały wektor *sar* uzyskuje się aproksymację dzielenia.

Niewątpliwą zaletą metody jest jej prostota i stosunkowo niewielkie zużycie zasobów logicznych. Wada to iteracyjność – jedno dzielenie wymaga liczba bitów wyniku $\times (2 + 8)$ taktów zegara, gdzie 8 oznacza latencję użytej mnożarki. Przy czym w naszym przypadku, kiedy dzielnie potrzebne jest raz na ramkę, takie podejście wydaje się być najlepszym wyborem.

Dostarczony moduł składa się z pliku *divider_28_20.v* oraz IP Core *mult_28_20.lm*. Oba należy dodać do projektu. Przy czym IP Core (wszystkie pliki o nazwie *mult_28_20.lm*) dobrze umieścić w folderze *ipcore_dir*. Później dodajemy plik *mult_28_20.lm.xco*). Dzielarka domyślnie ma ustawioną szerokość dzielnej i wyniku na 28 bitów, a dzielnika na 20 bitów. Oczywiście można to zmienić, ew. trzeba tylko przegenerować IP Core. Jeśli nowy miałby inną latencję niż 8, trzeba ustawić odpowiedni parametr.

Moduł zwraca, oprócz wyniku dzielenia, flagę *qv* – informacja o poprawności wyniku. Wykorzystamy ją do "zatrzaśnięcia" wyniku – środka ciężkości. Definiujemy dwa rejesty na współrzędne środka ciężkości. Następnie wewnątrz procesu, w momencie gdy odpowiednia flaga wynosi '1', przepisujemy wynik dzielenia do stworzonego rejestru. Natomiast wartości z rejestru przypisujemy do wyjść modułu (*assign*).

Dzielarkę należy uruchomić sygnałem *eof*. Moduł rejestruje argumenty wewnętrz, także to, że w następnym taktie zostaną wyzerowane nie będzie miało znaczenia.

- Testowanie symulacyjne modułu jest dość proste. Po dodaniu go do pliku *tb_hdmi* i odpowiednim podłączeniu, uruchamiamy symulację i monitorujemy sygnały *m00*, *m01*, *m10* oraz wyniki dzielenia. Powinny one być w 100 % zgodne z modelem programowym. Oczywiście przy założeniu, że mamy uzgodniony wynik po etapie binaryzacji.
- Realizacja modułu wizualizacji jest, wbrew pozorom, dość prosta. Podstawą są liczniki określające aktualną pozycję na ekranie. Należy wykorzystać stworzony wcześniej kod. Następnie sprawdzamy, czy położenie piksela odpowiada położeniu środka ciężkości i jeśli tak to "podmieniamy" kolor rozważanego piksela. Uwaga. Proszę pamiętać, że rysujemy dwie prostopadłe linie, a nie punkt – pojedynczy piksel byłby trudno widoczny. Przykładowy kod:

```
assign o_red = ((x_cnt[9:0]==x || y_cnt[9:0]==y)?8'hff:i_red);
```
- Moduł wizualizacji należy przetestować symulacyjnie. Proszę pamiętać o sposobie jego podłączenia, w stosunku do modułu obliczania środka ciężkości.
- W ostatnim kroku należy oba moduły dodać do pliku *hdmi_main* i przetestować na karcie ATLYS. Proszę pamiętać o ustawieniu parametrów *IMG_W* i *IMG_H* na odpowiednio 720 i 576.

11.3 Zadania do wykonania w domu

Zadanie 11.2 Proszę zaproponować moduł wizualizacji pozycji środka ciężkości, w którym zamiast dwóch przecinających się linii, wyświetlane będzie koło o promieniu kilku pikseli. Moduł należy przetestować symulacyjnie i w sprzecie. ■

11.4 Zadania dodatkowe

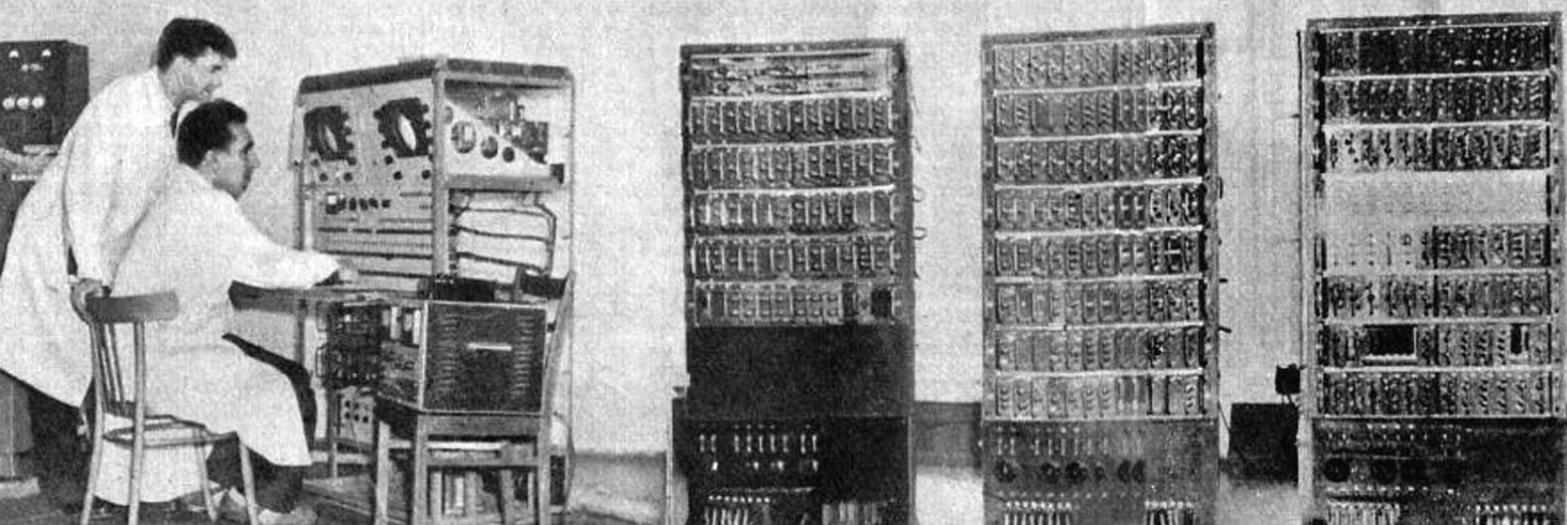
Zadanie 11.3 Proszę zrealizować moduły, które pozwolą na wyznaczenie i wyrysowanie prostokąta otaczającego (ang. *bounding box*) dla wykrytego obiektu. Dla uproszczenia zakładamy, że wierzchołki prostokąta wyznaczają maksymalne i minimalne współrzędne pikseli obiektów w obu osiach (nie wykonujemy indeksacji), zatem musimy traktować wszystkie "białe" piksele jako należące do jednego obiektu). Moduł należy przetestować symulacyjnie

i w sprzęcie.



Wskazówki:

- zadanie jest dość proste. Trzeba wykorzystać licznik identyczny jak w module wyznaczania środka ciężkości (można nawet oba moduły zintegrować). Następnie trzeba stworzyć rejestrysty na wierzchołki prostokąta oraz dodać odpowiednią logikę.
- moduł używamy podobnie jak *centroid* tj. "obok" głównego toru wizyjnego.
- wyświetlanie prostokąta również jest względnie proste. Trzeba tylko uzupełnić moduł wizualizacji o nowe warunki logiczne. Dla ambitnych – można spróbować wyświetlać prostokąt i środek ciężkości w różnych kolorach.



12 — Potokowa realizacja operacji kontekstowych

12.1 Koncepcja realizacji operacji kontekstowych w potokowym systemie wizyjnym

Na początku drobne przypomnienie/wyjaśnienie – co to są operacje kontekstowe? Zatem operacje kontekstowe, to takie w których nowa wartość piksela nie zależy tylko od aktualnej wartości (jak w przypadku operacji LUT), ale również od otoczenia danego piksela. Przykłady:

- filtracja dolnoprzepustowa (uśredniająca, Gaussa itp.),
- filtracja górnoprzepustowa (Sobel),
- filtracja medianowa,
- operacje morfologiczne (erozja, dylatacja i inne),
- wiele innych, bardziej złożonych operacji analizy obrazu (temat ważny).

W ramach ćwiczenia będziemy realizować binarną filtrację medianową z oknem o rozmiarze 5×5 . Wyznaczenie mediany dla obrazu binarnego jest bardzo proste i wymaga tylko zliczenia pikseli o wartości '1' występujących w aktualnie rozpatrywanym oknie. Jeśli suma jest większa od połowy rozmiaru okna to jako wyniki uznajemy '1', jeśli mniejsza to '0'. Koncepcję modułu przedstawiono na rysunku 12.1.

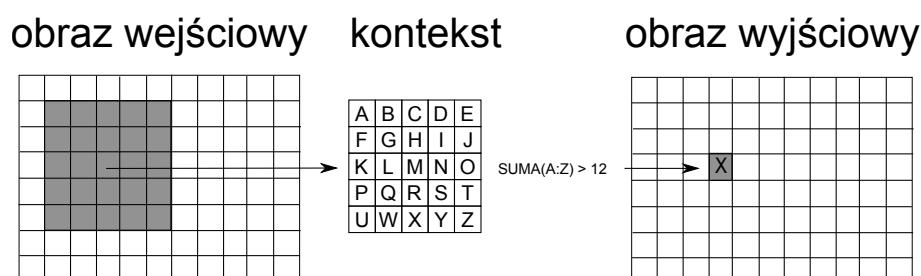


Figure 12.1: Koncepcja realizacji operacji medianowej z oknem 5×5

Mamy zatem obraz wejściowy, z którego wycinamy kolejne konteksty o danym rozmiarze (w tym przypadku 5×5). Warto zauważyć, że dla pikseli brzegowych nie da się wyznaczyć pełnego kontekstu (dokładnie dwóch pikseli z każdego brzegu). W literaturze spotyka się wiele możliwości "obsługi" tego przypadku:

- zwiększenie rozmiaru obrazu wejściowego i uzupełnienie brakujących pikseli poprzez

- powielenie pikseli brzegowych,
- zawijanie obrazu (brakujące piksele z lewej uzupełniane są tymi z prawej itd.),
- założenie, że "brakujące" piksele mają ustaloną wartość np. 0, 127 lub 255,
- założenie, że pikseli z ramki się nie przetwarzają (wtedy rezultat filtracji ma mniejszy rozmiar niż oryginał)
- przekopiowanie oryginalnych pikseli "ramki" do obrazu wynikowego.
- uznanie, że na obrazie wynikowym piksele brzegowe będą mieć wartość 0.

Należy zauważyć, że o ile implementacja każdej z tych metod programowo (Matlab, C++, Java) jest dość prosta, o tyle ich implementacja sprzętowa (i w dodatku potokowa) nastręcza pewnych trudności. Dla ułatwienia, w rozważanym projekcie wykorzystana zostanie metoda ostatnia, czyli piksele, które nie mają pełnego kontekstu, uzyskają na obrazie wynikowym wartość 0. Warto także zwrócić uwagę, że z praktycznego punktu widzenia obsługa pikseli brzegowych nie jest zbyt istotna, gdyż pominięcie filtracji ramki o szerokości 1 lub 2 piksele (odpowiednio filtr 3×3 i 5×5) zwykle ma marginalny wpływ na działanie całego systemu wizyjnego o rozdzielcości np. 720×576 .

Wybrany kontekst (A-Z) trzeba następnie zsumować. Należy tu wykorzystać tzw. drzewo sumacyjne, którego koncepcja pokazana została na rysunku 12.2.

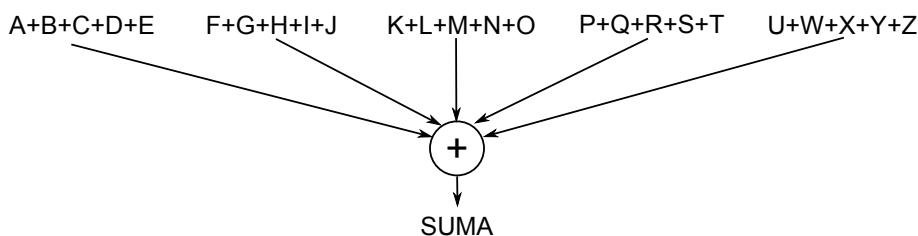


Figure 12.2: Koncepcja realizacji drzewa sumacyjnego

Na pierwszym poziomie drzewa sumujemy wartości w wierszach kontekstu, a na drugim uzyskujemy ostateczną sumę. Warto zwrócić uwagę, że w przypadku obrazów binarnych możemy (a wręcz powinniśmy) dodawać wartości bitowe (1 lub 0), co umożliwia realizację wielu dodawań na jednym poziomie drzewa bez specjalnego wypływu na maksymalną częstotliwość pracy modułu (sumowanie wielu liczb o znacznych szerokościach wprowadza znaczne opóźniania na tzw. ścieżce krytycznej logiki asynchronicznej).

Otrzymaną sumę należy porównać z połową rozmiaru maski (12) i zwrócić wynik filtracji.

Na końcu trzeba się zastanowić jak przekształcić sekwencję pikseli przychodzącej z kamery na kontekst. Wykorzystuje się tutaj tzw. schemat linii opóźniających, których schemat zaprezentowano na rysunku 12.3.

Moduł składa się z 25 pojedynczych opóźnień (D – o jeden takt zegara – przerzutnik) oraz czterech długich linii opóźniających (w których pamiętana jest cała linia obrazu – z pominięciem pięciu pikseli, które pamiętane są w elementach pojedynczych). Kontekst to wyjścia z pojedynczych przerzutników (D). Istotny jest rozmiar H_SIZE . Na podstawie informacji dotyczących postaci sygnału HDMI, wartość H_SIZE to nie po prostu szerokość obrazu w pikselach. Trzeba też uwzględnić próg przedni, synchronizację i próg tylny (por. rozdział 8). Zatem dla przypadku z rysunku nie byłoby to 640, a 800 pikseli!

Zaleca się "wyobrażenie" sobie sposobu, w jaki przetwarzany jest pojedynczy piksel i dlaczego taki mechanizm ma prawo działać.

12.2 Zadania do wykonania na laboratorium

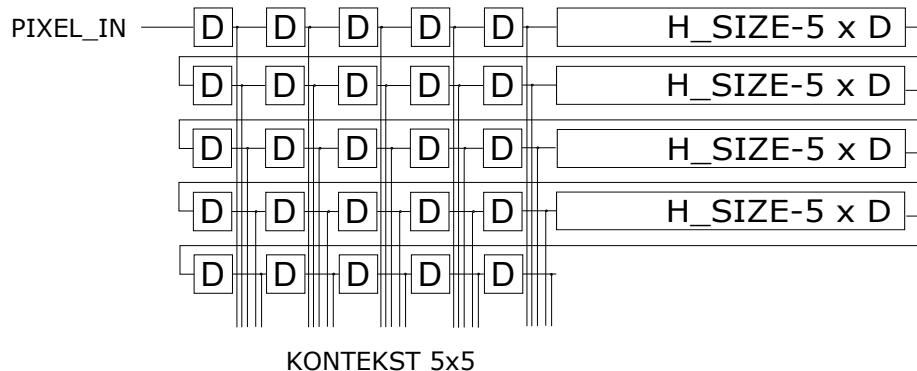


Figure 12.3: Schemat linii opóźniających dla kontekstu 5×5

Zadanie 12.1 Proszę zaimplementować i przetestować symulacyjne binarną filtrację medianową z maską o rozmiarze 5×5 .

Wskazówki:

- Tradycyjnie zaczynamy od stworzenia nowego modułu – *median5x5.v*. Mam mieć on typowy interfejs wizyjny, przy czym dla uproszczenia obliczeń zakładamy, że na wejściu podajemy maskę bitową (w praktyce najmłodszy bit z maski uzyskanej z poprzedniego modułu), a na wyjściu wypisujemy już wektor 8-bitowy (z zakresu 0-255). Moduł powinien posiadać jeden parametr – *H_SIZE*, który opisuje szerokość obrazu (sekcja *parameter*). Jako domyślną wartość należy ustawić 83 – odpowiada to przetwarzaniu obrazu o rozdzielczości 64×64 .
 - W pierwszym kroku tworzymy moduł długiej linii opóźniającej. Dodaj nowy IP Core, nazwij go *delayLineBRAM* (**dokładnie tak**) i wybierz *Memories & Storage Elements -> RAMs & ROMs -> Block Memory Generator*. Uwaga. Do realizacji linii opóźniających idealnie nadaje się pamięć blokowa (Block RAM) dostępna w układzie FPGA – zapewnia ona odpowiednią długość i szerokość oraz wspiera operacje FIFO (ang. *First In First Out*). Ustaw następujące opcje:
 - Interface Type -> Native,
 - Memory Type -> Single Port RAM,
 - Write Width -> 17 (wartość maksymalna dla pojedynczego modułu),
 - Write Depth -> 1024 (to gwarantuje przetwarzanie obrazu 720×576 z dostępnej kamery HDMI),
 - Operating mode -> Read first,
 - zauważ ile bloków pamięci wykorzystujemy.Dodaj do projektu moduł *delayLineBRAM_WP* (dostępny na stronie kursu). Przeanalizuj jego implementację. Czy zgadzisz się ze stwierdzeniem, że działa on jak FIFO zbudowane na tablicy (tu module Block RAM)?
 - Jak wiemy należy również zapewnić poprawne opóźnienie sygnałów synchronizacji: *de*, *hsync*, *vsync*. Najprostszą możliwością jest ich "doklejenie" do przetwarzanego piksela. Wykorzystamy to w realizowanym module. Zatem pojedyncza porcja danych będzie składać się z 4 bitów: *{mask, de, hsync, vsync}*.
 - Trzeba teraz przygotować i opisać strukturę zgodną z rysunkiem 12.3: 25 krótkich linii opóźniających (w postaci 25 rejestrów o odpowiedniej długości) i cztery długie linie opóźniające. Wszystkie przypisania należy zrealizować w ramach jednego procesu. Uwagi:
 - skoro pojedyncza porcja danych składa się z 4-bitów to wymagane cztery linie opóźniające możemy zrealizować z wykorzystaniem pojedynczego modułu BRAM. Trzeba

tylko zadbać o odpowiednie przypisanie sygnałów wejściowych i wyjściowych. Użyteczny będzie operator konkatenacji – np. {x,y,z}.

- sygnał *rst* i *ce* w długiej linii opóźniającej należy ustawić na odpowiednio '0' i '1'.
- port *h_size* modułu długiej linii opóźniającej należy ustawić na *H_SIZE* - 5 – z uwagi na opóźnienie w pojedynczych rejestrach.
- szerokość danych dla modułu *delayLineBRAM_WP* została ustawiona na 16 tj. 4 porcje danych o szerokości 4 bitów (odpowiada za to parametr *WIDTH*).
- do pierwszego z 25 rejestrów należy przypisać sygnały *mask* oraz *de*, *hsync*, *vsync*.
- należy stworzyć sygnał na dane wyjściowe z długiej linii opóźniającej i wykorzystać go do przypisania wewnątrz procesu (jak pamiętamy rejestr nie może być argumentem wyjściowym).
- kluczem do sukcesu jest dobre nazewnictwo rejestrów – sugeruje się numerowanie wierszami i kolumnami np. *D11* itp.
- Realizacja drzewa sumacyjnego jest prosta. Należy tylko zdefiniować rejesty na sumy cząstkowe i ostateczną oraz dobrać ich rozmiary. Samo dodawanie można wykonać w ramach procesu (tego samego co opóźnienie).
- Żeby całość działała poprawnie należy jeszcze odpowiednio opóżnić sygnały synchronizacji dla piksela centralnego, oraz flagę *context_valid* (iloczyn sygnału *de* wszystkich pikseli kontekstu – np. *D11[2]* & *D12[2]* *D55[3]*, która określa, czy rozpatrywany kontekst jest poprawny). W przypadku dwupoziomowego drzewa sumacyjnego (latencja=2) opóźniamy dokładnie o 2. Uwaga. Dla operacji kontekstowych zwykle działamy w odniesieniu do piksela centralnego tj. środkowego dla danej maski. Do opóźnienia można zastosować używaną wcześniej linię opóźniającą zbudowaną na przerutnikach (w odróżnieniu od długich linii opóźniających opartych o pamięć blokową).
- Na samym końcu należy wypisać wynik operacji tj. wartość piksela i trzy sygnały synchronizacji. Wykorzystujemy instrukcję *assing*. Uwaga. Na tym etapie należy także sprawdzić, czy uzyskana suma jest większa, czy mniejsza od połowy liczby pikseli w masce. Można to np. zrealizować na pomocą następującej instrukcji:
`assign mask_new = sum > 5'd12 ? 255 : 0;`. Należy też sprawdzić poprawność kontekstu – składnia zbliżona. Uwaga. Należy zadbać, aby sygnały synchronizacji były zsynchronizowane z pikselem.
- Testowanie symulacyjne. Tradycyjnie jest to ważny element realizacji projektu. W przypadku tak subtelnej operacji jak mediana, ocena poprawności implementacji "na oko" niekoniecznie jest najlepszym sposobem. Zatem wstawiamy stworzony moduł do pliku *tb_hdmi*. Jeśli mamy pewność, że nasz model jest 100 % zgodny z FPGA to po prostu sprawdzamy wyniki. Powinny one być zgodne z funkcją *medfilt2* dokładnością do krawędzi. Można je "wyzerować":
`m(1:2,:) = 0; m(63:64,:) = 0; m(:,1:2) = 0; m(:,63:64) = 0;` Wtedy powinniśmytrzymać pełną zgodność. Inną opcją jest zapisanie wyniku binaryzacji z modelu symulacyjnego. Stosując go jako wejście do modelu programowego możemy sprawdzić samą medianę.
- Testowanie w sprzęcie. Stworzony moduł należy dodać do plik *hdmi_main* i odpowiednio podłączyć. Bardzo ważne jest ustawienie parametru ***H_SIZE* na 864**. Wartość wynika oczywiście ze specyfiki sygnału HDMI i rozdzielczości sygnału z kamery: 720×576 .

12.3 Zadania do wykonania w domu

Zadanie 12.2 Wykorzystując zaimplementowany moduł zrealizuj moduł morfologicznego otwarcia lub zamknięcia (czyli kombinacji erozji i dylatacji). Rozwiążanie

przetestuj symulacyjnie i w sprzęcie.

12.4 Zadania dodatkowe

Zadanie 12.3 Wykorzystując koncepcję operacji kontekstowych zrealizuj moduł filtracji uśredniającej z oknem 3×3 . Idea przedstawiona została na rysunku 12.4. Po implementacji moduł należy przetestować symulacyjnie (zgodność z modelem programowym) oraz w sprzęcie.

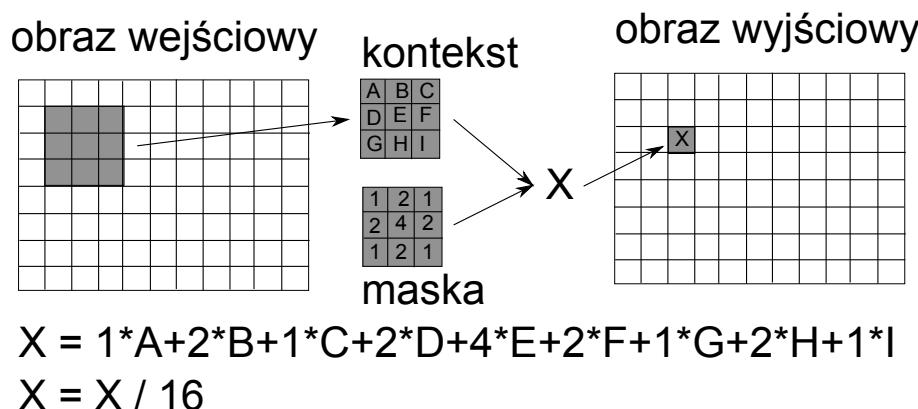


Figure 12.4: Koncepcja realizacji filtracji uśredniającej 3×3

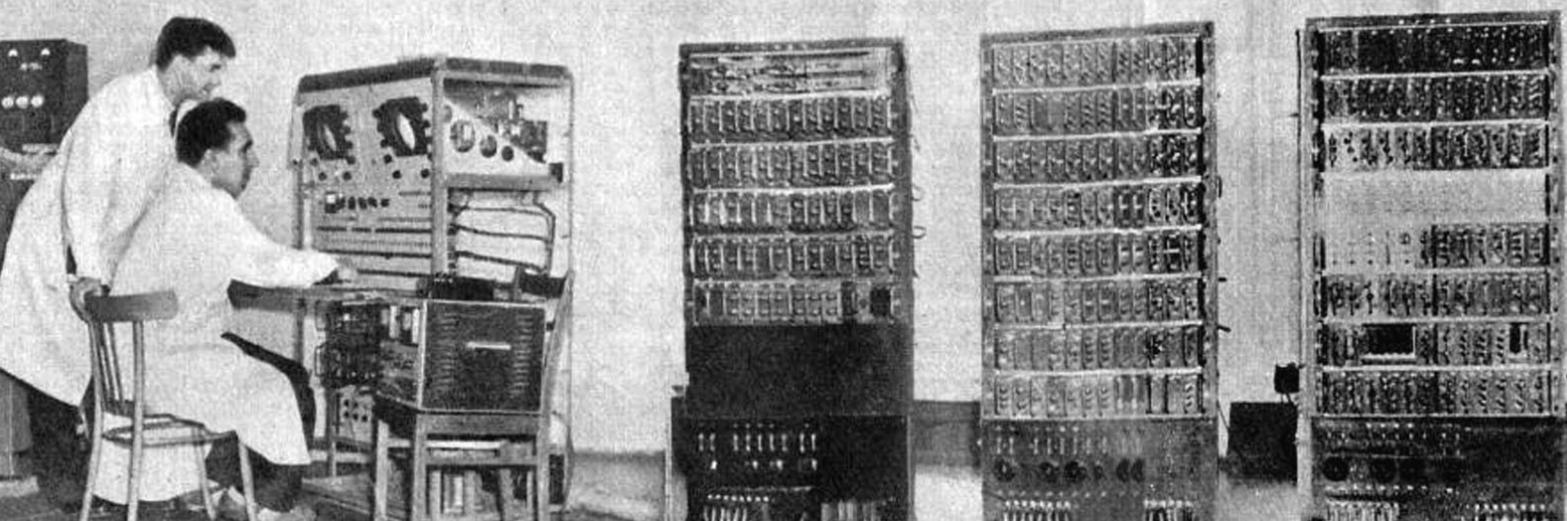
Wskazówki:

- zakładamy, że przetwarzamy tylko składową Y – jasność, a chrominancję pomijamy,
- generowanie kontekstu należy wykonać analogicznie jak w przypadku mediany. Trzeba tylko wykorzystać dwa moduły BRAM, gdyż dla 8 bitów danych i 3 bitów synchronizacji jeden moduł jest niewystarczający.
- mnożenie zastępujemy operacjami bitowymi, gdyż współczynniki to potęgi liczby 2. Stosujemy konkatenację np. $b = a, 1'b0$ – mnożenie b przez 2. Odbywa się to poza procesem. Wykorzystujemy instrukcję `assign`. Trzeba zapewnić odpowiedni rozmiar sygnałów wyników.
- wyniki mnożenia sumujemy – drzewo sumacyjne, a następnie dzielenie przez 16. Ponownie stosujemy przesunięcie bitowe. Dla sumowania trzeba zwrócić uwagę na szerokości rejestrów wynikowych. Ponadto sumowane wartości należy uzupełnić z lewej strony "zerami", tak aby wszystkie argumenty sumowania miały taki sam rozmiar – przykład: $1'b0, s1 + s2 + 1'b0, s3$;
- przy obsłudze krawędzi do wyboru: przepisanie oryginalnej wartości (niezmodyfikowanej) lub ustalenie na wartość 0.
- model symulacyjny łatwo stworzyć w oparciu o dotychczas wykorzystywany w pakiecie Matlab.

Zadanie 12.4 Wykorzystując koncepcję operacji kontekstowych zrealizuj moduł detekcji krawędzi – kombinowany filtr Sobela. Powinien się on składać z:

- dwóch modułów Sobela – pionowego i poziomego (będą one bardzo zbliżone do filtracji, przy czym trzeba uwzględnić operacje na liczbach ze znakiem),
- połączenia wyników operatorem `abs` tj. $S = |S_x + S_y|$.
- skalowania wyniku lub binaryzacji – tak aby jakoś zaprezentować liczby, które nie mieszczą się w zakresie 0-255.

Po implementacji moduł należy przetestować symulacyjnie (zgodność z modelem programowym) oraz w sprzęcie.



Bibliography

- [1] R.C. Gonzalez and R.E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
- [2] R.C. Gonzalez, R.E. Woods, and Eddins S.L. *Digital Image Processing Using MATLAB (2nd Edition)*. Gatesmark Publishing, 2009.
- [3] Xilinx Inc. Spartan-6 FPGA Configurable Logic Block - User Guide. 2010.
- [4] Xilinx Inc. Spartan-6 FPGA DSP48A1 Slice User Guide. 2010.
- [5] Xilinx Inc. Spartan-6 FPGA GTP Transceivers User Guide. 2010.
- [6] Xilinx Inc. Spartan-6 FPGA Integrated Endpoint Block for PCI Express User Guide. 2010.
- [7] Xilinx Inc. Spartan-6 FPGA Integrated Endpoint Block for PCI Express User Guide (AXI). 2010.
- [8] Xilinx Inc. Spartan-6 FPGA Memory Controller User Guide. 2010.
- [9] Xilinx Inc. Spartan-6 FPGA Block RAM Resources User Guide. 2011.
- [10] Xilinx Inc. Spartan-6 FPGA Clocking Resources User Guide. 2011.
- [11] Xilinx Inc. Spartan-6 FPGA SelectIO Resources User Guide. 2014.
- [12] J. C. Russ. *Image Processing Handbook, Fourth Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2002.
- [13] R. Tadeusiewicz. *Komputerowa analiza i przetwarzanie obrazów*. Wydawnictwo Fundacji Postępu Telekomunikacji, 1997.
- [14] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.