Systemy Rekonfigurowalne

skrypt do ćwiczeń laboratoryjnych

Mateusz Komorkiewicz,
Tomasz Kryjak

PUBLISHED BY AGH

First printing, March 2014

Spis treści

1 — Wprowadzenie

1.1 Słowo wstępne

Umiejętność programowania architektur równoległych jest w dzisiejszych czasach bardzo pożądana. Z przyczyn technologicznych maksymalna częstotliwość taktowania procesorów sekwencyjnych zatrzymała się na ok. 5 GHz, przy czym praktycznie stosuje się rozwiązania o taktowaniu mniejszym od 4 GHz. Zatem jedyną drogą akceleracji staje się szeroko rozumiane zrównoleglanie obliczeń. Przykładem są procesory wielordzeniowe ogólnego przeznaczenia (CPU – ang. *Central Processor Unit*) lub (GPP – ang. *General Purpose Processor*, GPP – ang. *General Purpose Processor*), programowalne karty graficzne (GPGPU –ang. *General-purpose computing on Graphics Processing Units*), 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 nieosiągalny dla systemów technicznych), to myśleć wolimy w sposób sekwencyjny. Programowanie w klasycznych językach C/C++/C#/Java/Python 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 można dość łatwo 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" przez które "przepływa" strumień danych. Ponadto, aby wykorzystać możliwości jakie oferuje równoległość należy dość dobrze znać wykorzystywaną platformę sprzętową (np. GPGPU, instrukcje wektorowe w CPU). W przypadku programowania na CPU nie ma to tak dużego znaczenia, bo dostępne kompilatory tworzą bardzo dobry kod maszynowy w sposób w pełni automatyczny. Można wręcz zaryzykować stwierdzenie, że możliwe jest tworzenie programów np. w Javie zupełnie nie wiedząc jak działa system komputerowy, procesor, pamięć instrukcji i danych itp.

Programowanie równoległe jest niewątpliwie trudniejsze od sekwencyjnego. Jednak wydaje się być na chwilę obecną koniecznością, gdyż nie pojawiała się technologia umożliwiająca dalszą akcelerację operacji sekwencyjnych. 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.

Warto w tym miejscu przedstawić klasyfikację architektur komputerowych zaproponowaną

przez prof. Michaela Flynna w 1966r. (wciąż aktualną). Wyróżnia się w niej cztery klasy ze względu na liczbę równoległych strumieni danych i instrukcji.

- SISD (ang. *Single instruction stream, single data stream*) – pojedynczy strumień instrukcji, pojedynczy strumień danych. Tego typu architektura nie wykorzystuje żadnego zrównoleglenia obliczeń. Jednostka obliczeniowa (PU – ang. *Processing Unit*) wykonuje pojedynczą instrukcję na pobranych z pamięci danych. Przykładem takiego rozwiązania są standardowe procesory ogólnego przeznaczenia (jednordzeniowe, bez wielowątkowości).
- SIMD (ang. *Single instruction stream, multiple data stream*) – pojedynczy strumień instrukcji, wiele strumieni danych. W tym przypadku wiele jednostek obliczeniowych wykonuje te same operacje na różnych danych. Jest to najbardziej intuicyjna forma zrównoleglenia obliczeń. Przykładem może być generowanie grafiki, gdzie te same operacje są wykonywane dla różnych fragmentów obrazu.
- MISD (ang. *Multiple instruction stream, single data stream*) – wiele strumienie instrukcji, pojedynczy strumieni danych. W takim rozwiązaniu na tych samych danych wykonywany jest ciąg instrukcji. Pierwszy przykład to system zwiększający niezawodność – trzy niezależne procesory (najlepiej różne) wykonują te same obliczenia na tych samych danych. Zgodny wynik uznawany jest za wartość poprawną. Ta nazwą można również określić tzw. przetwarzanie potokowe, gdzie pojedyczny zestaw danych (np. obraz) poddawany jest różnym przekształceniom w sposób sekwencyjny (np. korekcji kolorów, binaryzacji, indeksacji).
- MIMD (ang. *Multiple instruction stream, multiple data stream*) – wiele strumienie instrukcji, wiele strumieni danych. W tym przypadku wiele niezależnych elementów obliczeniowych realizuje różne instrukcje na różnych danych. Jest to typowa architektura dla superkomputerów

W ramach niniejszego kursu zajmować się będziemy układami rekonfigurowalnymi (reprogramowalnymi) FPGA (ang. *Field Programmable Gate Array*) – rozdział 1.3. Oprócz tego, że są one wręcz idealną platformą do realizacji obliczeń równoległych, to nie mają one zdefiniowanej na etapie produkcji funkcjonalności. Określenie jak będzie działał dany układ należy do projektanta logiki. Zatem "programowanie" [1] układów FPGA różni się dość zasadniczo od programowania CPU/GPU. W pierwszym przypadku budujemy logikę (elementy obliczeniowe oraz pamiętające) z podstawowych elementów logicznych (bramek, elementów LUT (ang. *Look-Up Table*), przerzutników, zasobów połączeniowych). Musimy też określić sposób przepływu danych pomiędzy poszczególnymi modułami. W drugim, architekturę mamy ściśle określoną. Nasza rola ogranicza się tylko do utworzenia odpowiedniego zbioru instrukcji.

Nauka umiejętności projektowania logiki realizującej konkretne 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 – por. rozdział 8. Zadanie jakie stoi przed nami nie jest najłatwiejsze, ale pozwala w odmienny sposób spojrzeć na "programowanie", co jest na pewno bardzo rozwijające.

Na koniec warto podkreślić, że na kurs należy patrzeć raczej jako wyzwanie intelektualne i możliwość zapoznania się z nietypową metodologią programowania, niż nabycie *stricte* praktycznych umiejętności projektowania w językach opisu sprzętu (Verilog).

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

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 rejestry (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 ulegały miniaturyzacji – od dużych obudów, które zawierały kilka bramek logicznych, po nowoczesne procesory wykorzystywane w aplikacjach i urządzeniach mobilnych, które charakteryzują się niskim zużyciem energii i dużą wydajnością obliczeniową (przykład to porównanie możliwości przeciętnego smartfona i komputera PC z początku lat 90 XX w.).

Ten spektakularny rozwój nie byłby możliwy, gdyby nie istniały odpowiednie narzędzia, które umożliwiały projektowanie coraz bardziej skomplikowanych architektur. W latach 80 powstał problem opisu struktury i zachowania układów scalonych, w szczególności dokumentowania działania. Ciągła miniaturyzacja wymuszała tworzenie coraz bardziej skomplikowanych struktur, natomiast wykorzystywane metody projektowania układów przy pomocy schematu ideowego (graficznego) nie pozwalały na osiągniecie tego celu w prosty sposób. W związku z tym, powstało zapotrzebowanie na tzw. języki opisu sprzętu (HDL ang. *Hardware Description Languages*). Jednym z lepiej znanych jest opracowany w latach 80 przez Departament Obrony Stanów Zjednoczonych język VHDL (ang. *Very High Speed Integrated Circuits Hardware Description Language*).

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 dość 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

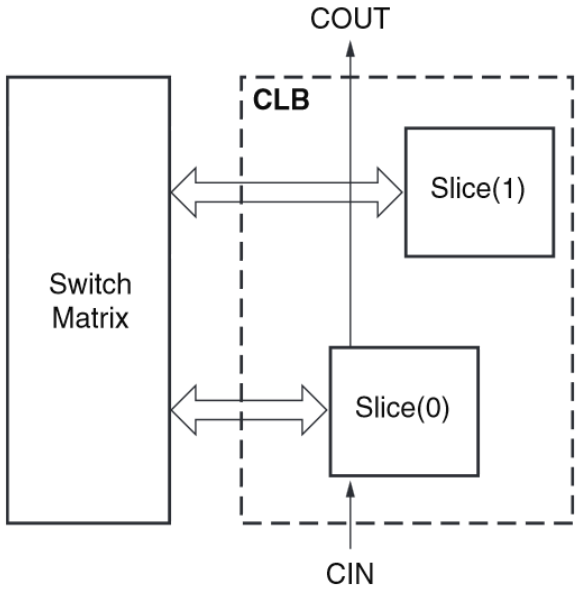
Jak już zostało wspominane, aby dobrze projektować logikę dla układów FPGA należy poznać podstawy ich budowy. W niniejszym rozdziale zostaną zatem omówione podstawowe zasoby logiczne dostępne w układach Spartan 6 firmy Xilinx. Ogólny opis rodziny Spartan 6 można odnaleźć w dokumencie [3].

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 przełączeń (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.

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:



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

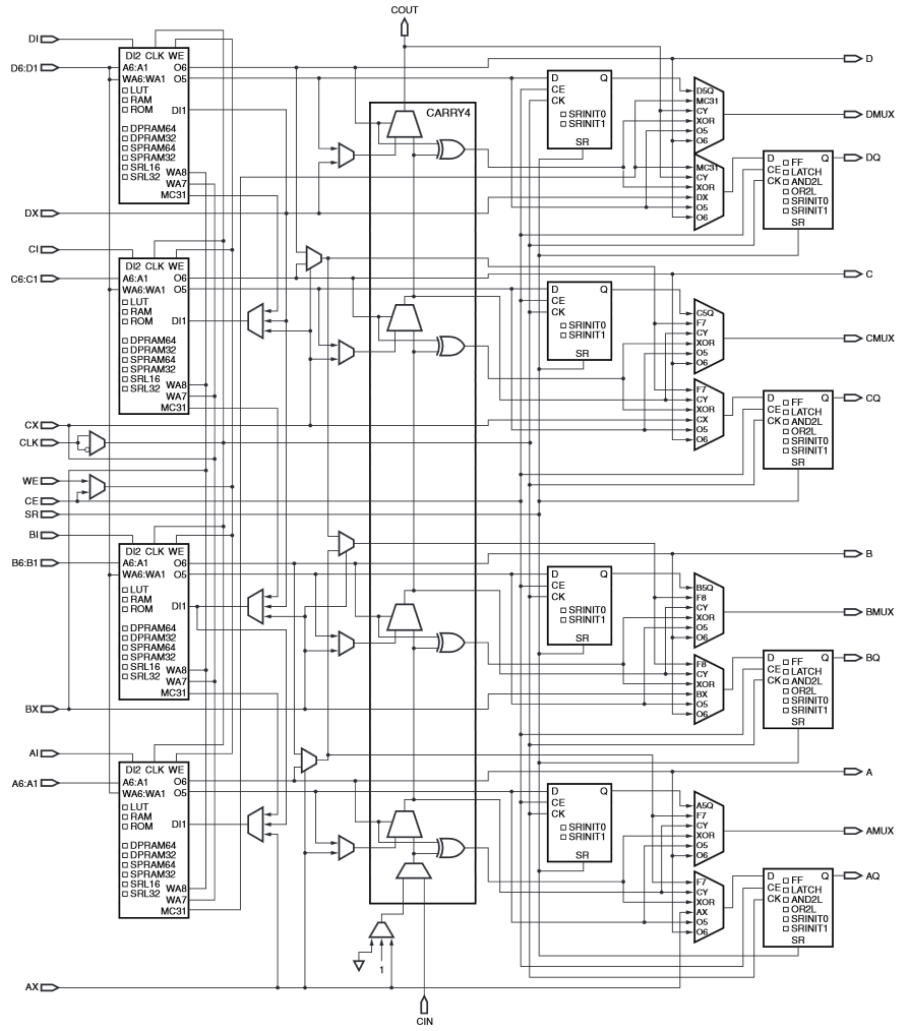
- generator funkcyjny (4 sztuki) — został zrealizowany jako LUT (ang. *Look-Up Table*) posiadający 6 wejść i dwa niezależne wyjścia. Zatem możliwe jest zaimplementowanie: 6-wejściowej funkcji logicznej, dwóch 5-wejściowych funkcji logicznych (ze wspólnym wejściem), dwóch funkcji logicznych z 3 i 2 wejściami. 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 $\times$ 1 jednoportowa, 128 $\times$ 1 dwuportowa i 64 $\times$ 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 zwarte są w dokumencie [3] dostępnym na stronie `www.xilinx.com`.

1.3.2 Pozostałe zasoby

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

- CMT (ang. *Clock Managment Tiles*) — bloki zarządzania sygnałem zegarowym, które zapewniają generację różnych częstotliwości zegara, równomierną propagację sygnału zegarowego oraz tłumienie zjawiska *jitter* (zakłócenia fazy zegara). W układzie znajduje się od 2 do 6 tego typu modułów. Szczegółowe informacje dostępne w dokumencie [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 dokumencie [10],
- DSP48A1 — moduły z mnożarką 18 $\times$ 18 bitów oraz 48-bitowym akumulatorem. Ich liczba waha się od 9 do 180 w zależności od rozmiaru układu. Szczegółowe informacje



Rysunek 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]

dostępne w dokumencie [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 dokumencie [11],
- GTP Transceivers — moduły nadawczo-odbiorcze umożliwiające szybką transmisją 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 rodziny 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 rodziny 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 rodziny 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 linii globalnych i lokalnych. 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 wyobrażaliśmy).

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 na przełomie roku 1983/1984 przez firmę Gateway Design Automation Inc. Od tego czasu przeszedł wiele modyfikacji, został upubliczniony na zasadzie otwartego standardu i dokonano jego oficjalnej standaryzacji jako norma IEEE 1364 (po raz pierwszy w 1995, a później w 2001 roku). 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, weryfikacji i symulowania coraz bardziej złożonych systemów elektronicznych (połowa lat 80 XX wieku). Pomysł, 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, a są niezbędne przy symulacji 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. [1] W pracach używać będziemy wersji 14.7. Środowisko to nie różni się znacząco od typowych IDE np. Visual Studio lub Eclipse (choć jest może nieco "toporne").

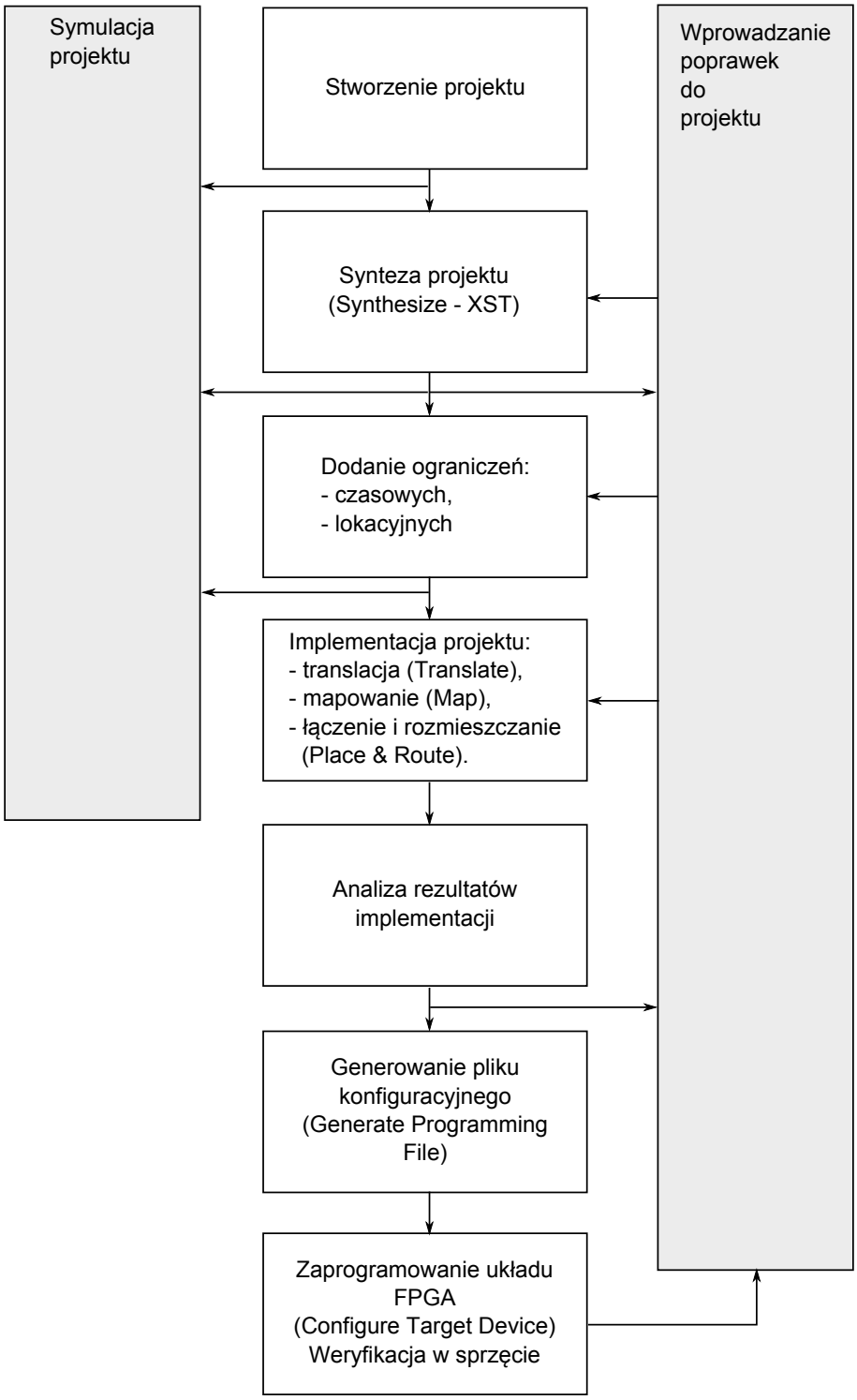
Jednakże kluczowe dla zrozumienia istoty projektowania logiki jest przeanalizowanie, co dzieje się z napisanym kodem zanim można go wgrać na kartę z układem FPGA. Etapy tworzenia logiki w układzie FPGA przedstawiono na rysunku 2.3:

- 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 kompilowane 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 FPGA,
- 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 lub też trzeba tworzyć dodatkową logikę, która wyświetli pewne istotne informacje kontrolne. O ewentualnych przyczynach nie mamy informacji (wyjątek stanowi narzędzie ChipScope omówione poniżej). Dużo lepszym i szybszym rozwiązaniem jest

[1] Dla nowszych układów firmy Xilinx tj. serii 7 oraz Ultra Scale dedykowane jest środowisko Vivado. Niestety układy serii 6 nie są w nim wspierane.



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

symulacja, gdzie praktycznie możemy uzyskać kompletną informację o zachowaniu się modułu. Możemy ją wykonać na różnym etapie projektu (behawioralnym, *post-translate, post-map, post place &route*). Zagadnienie to zostanie szczegółowo omówione w rozdziale 4.

- 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 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 pełnej symulacji rozwiązania (choć oczywiście istnieją modele symulacyjne np. zewnętrznych pamięci RAM). Warto również zaznaczyć, że ChipScope ma ograniczone możliwości analizy dużej liczby danych, przykładowo strumienia wideo.

Z powyższego opisu wyraźnie wynikają różnice, w tworzeniu projektu na CPU i FPGA. Na CPU mamy do dyspozycji "sztywną" architekturę (którą znamy dokładnie lub nie) i piszemy na nią kod. Na FPGA musimy sobie stworzyć architekturę obliczeniową, tj. moduły realizujące poszczególne operacje. Raczej nie mówi się w tym przypadku o wykonywaniu jakiegoś ciągu instrukcji. Jak zobaczmy w trakcie kursu istota projektowania logiki polega na wykonywaniu elementów obliczeniowych oraz ustalaniu przepływu danych pomiędzy nimi.

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 (wszystkie elementy).

Na laboratorium będziemy używać wersji System Edition. Ponieważ w ramach kursu proponowane będą różne zadania domowe oraz dodatkowe, zatem przydatne 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 dostępny na płycie używanej na laboratorium tj. Atlys jest wspierany przez wersję WebPACK.

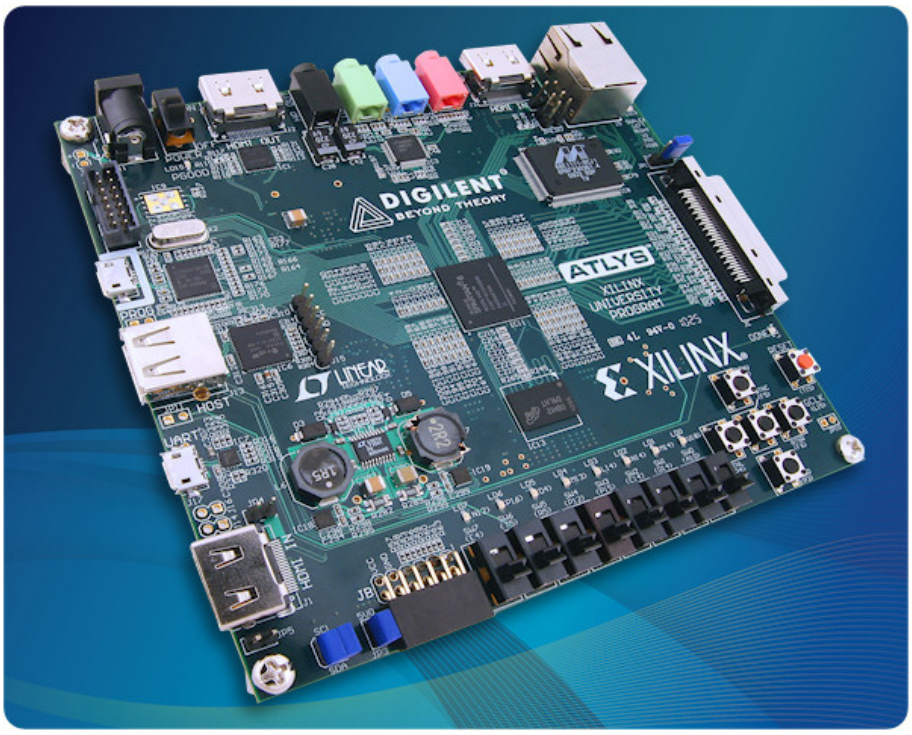
Instalacja jest dość prosta. Na stronie: `http://www.xilinx.com/products/design-tools/ise-de` należy wybrać Download ISE WebPACK software for Windows and Linux. Zostaniemy przeniesieni na stronę **Downloads**. Tam wybieramy **ISE Design Tools** i **ISE Design Suite - 14.7 Full Product Installation**. System operacyjny – *de gustibus* (choć z uwagi na kompatybilność z projektami w pracowni zalecany jest Linux (Debian/Ubuntu)). Niestety należy się zarejestrować na stronie Xilinx'a. Po rejestracji i logowaniu uzyskujemy dostęp do ściągania. Można ściągną 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.

Jej podstawowym elementem jest układ FPGA Spartan 6 LX45 firmy Xilinx. Plasuje 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,



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

- 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 (**uwaga** – proszę to robić bardzo ostrożnie, bo można zniszczyć port),
- chowamy do pudełka kartę oraz kable USB.

2.5 Zadania do wykonania na laboratorium

Zadanie 2.1 Stworzyć logikę, która umożliwi sterowanie diodami za pomocą przełączników. ■

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

Wykonanie:

1. otwórz program **ISE Design Suite 14.7** (instrukcja na stronie kursu),
2. utwórz nowy projekt — **File->New Project**,
3. w oknie dialogowym ustal nazwę **Name** (np. intro) oraz folder **Location** ("swój" folder),
4. ustal typ nadrzędnego pliku na HDL, pozostałe formaty to (schemat – niewygodny przy dużych projektach, EDIF (forma netlisty) oraz plik NGC/NGD omówiony wcześniej),
5. w następnym 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* – przypisanie asynchroniczne, które można utożsamiać z fizycznym połączeniem dwóch "kabli").
11. mając utworzony i skończony moduł omówimy środowisko ISE.
    Po lewej stronie ekranu (rozmieszczenie domyślne – można je przywrócić opcją *Layout->Load Default Layout*) 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 nadrzędnym (*Top Module*) dla projektu. Uwaga. Proces syntezy i implementacji oraz symulacji (oprócz behawioralnej) można wykonać tylko dla pliku oznaczonego jako nadrzędny. 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 wykonać

dla danego pliku. W rozważanym przypadku nasz plik jest nadrzędny 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 graficznymi,
- 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 syntezy 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 (liczniki, dekodery itp.), **niezależny** od docelowego układu FPGA.
- Synthesize - XST -> View Technology Schematic — przeglądarka schematu technologicznego, 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 syntezy,
- Implement Design -> Translate — uruchomienie procesu translacji,
- Implement Design -> Translate -> Generate Post-Translate Simultion 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/korekty procesu rozmieszczania i łączenia,
- Implement Design-> Map -> Generate Post-Map Simultion Model — stworzenie modelu symulacyjnego po etapie mapowania,
- Implement Design -> Place&Route — uruchomienie procesu rozmieszczania i łączenia,
- Implement Design -> Place&Route -> Generate Post-Place & Route Static Timing — generacja statycznej analizy czasowej po etapie rozmieszczania i łączenia,
- Implement Design -> Place&Route -> Generate Post-Place & Route Static Timing -> Analyze Post–Place&Route Static Timing — analiza statycznych zależności czasowych po etapie rozmieszczania 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 Simultion Model — stworzenie modelu symulacyjnego po etapie rozmieszczania 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 — wsteczna analiza lokalizacji pinów tj. uaktualnianie pliku UCF na podstawie rezultatów fazy rozmieszczania i łączenia,
- Implement Design -> Place&Route -> Back-annotate Pin Locations -> View Locked Pin Constraints — wsteczna 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 nadrzędny 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** – szczególnie przy uruchamianiu symulacji.
- 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 4).

Uwaga. 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 (na stronie kursu).

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. **uwaga** – w trakcie pracy nad projektem nie jest konieczne każdorazowe powtarzanie powyższego procesu. Program iMPACT wystarczy otworzyć raz i potem tylko reprogramować układ (po zakończeniu generacji pliku konfiguracyjnego).
22. przetestuj działanie układu tj. czy zmiana stanu przełącznika skutkuje zaświeceniem się odpowiedniej diody.

2.6 Zadania do wykonania w domu

Zadanie 2.2 Proszę pobrać i zainstalować program ISE Design Suite w wersji WebPACK ze strony *www.xilinx.com* – opis w rozdziale 2.3.1. ■

Zadanie 2.3 Proszę zapoznać się z podstawowymi informacjami o budowie układów FPGA – rozdział 1 niniejszego skryptu. ■

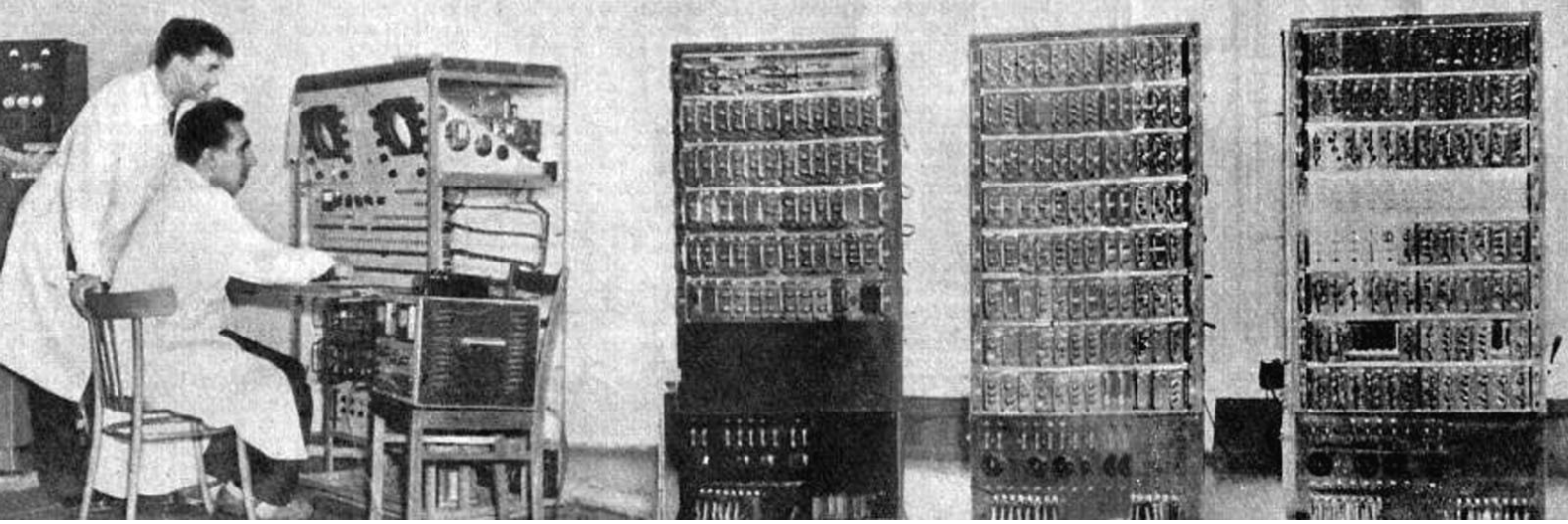
2.7 Podsumowanie

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.



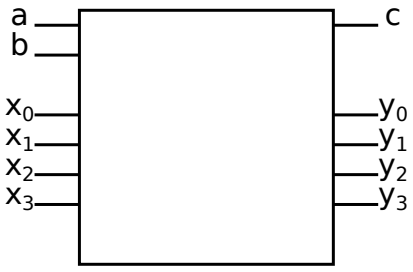
3 — Wstęp do projektowania struktury FPGA

3.1 Język Verilog – wprowadzenie

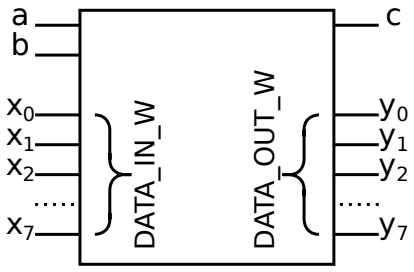
W niniejszym rozdziale zostaną przedstawione podstawowe elementy strukturalne występujące w języku Verilog, które umożliwiają projektowanie logiki w układach rekonfigurowalnych. Warto podkreślić, że jest to wariant “minumum”. Dalszych informacji należy szukać w szeroko rozumianym Internecie oraz w wielu dostępnych książkach (w języku polskim “Wprowadzenie do języka Verilog", Zbigniew Hajduk, BTC).

3.1.1 Moduł

Moduł jest podstawowym elementem, który jest wykorzystywany do opisywania struktury układów scalonych w języku Verilog. W zależności od potrzeb projektanta, 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). Z zewnątrz można również dostarczyć zestaw parametrów, które mogą zmieniać zachowanie elementów wewnątrz modułu – przykładowo szerokość danych wejściowych lub wyjściowych (por. rysunek 3.2). W języku Verilog, moduł odpowiada najczęściej jednemu plikowi o rozszerzeniu .v i jest definiowany następującym kodem:



Rysunek 3.1: Przykładowy moduł



Rysunek 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**. Uwaga. Ustalanie początkowych wartości ścieżek stosuje się tylko **w wybranych** sytuacjach. Typowa ścieżka łącząca dwa moduły **nie powinna być inicjowana**. Ścieżkę należy utożsamiać z fizycznym "kablem". Ma to swoje konsekwencje, które zostaną szerzej omówione w dalszej części kursu. W tym miejscu warto wspomnieć, że:

- do ścieżki nie można przypisać wyjść z dwóch różnych modułów. Tak jak nie można połączyć wyjść np. dwóch bramek AND i liczyć, że na wyjściu uzyskamy poprawną wartość.
- jeśli ścieżkę zainicjujemy wartością 0 (tj. podłączymy ją "na stałe" do masy), to próba przypisania do niej wartości skończy się błędem.

Ścieżka może składać się z pojedynczej linii lub być wielobitową szyną danych. W jezyku 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:

$$X'Yv \tag{3.1}$$

gdzie: X – to wartość określająca liczbę bitów zapisywanej liczby,
Y – określa sposób zapisu v (b – binarny, h – heksadecymalny, d – dziesiętny),
v – określa wartość wyrażenia w odpowiednim zapisie.

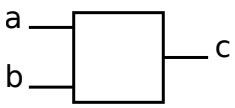
Np. jeśli portowi ma zostać przypisana liczba 123 zapisana na 8 bitach można tego dokonać na kilka sposobów:

Kod 3.1.5 — Zapisanie wartości w różnych formatach:

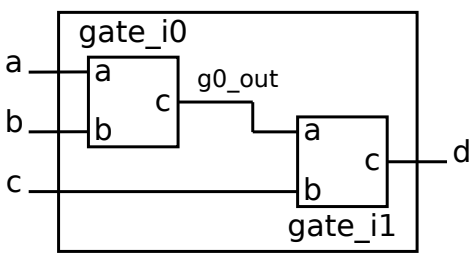
```
wire [7:0]value;
assign value=8'd123;      //decimal
assign value=8'h7b;       //hexadecimal
assign value=8'b01111011;//binary
```

3.1.4 Łączenie modułów

Raz zdefiniowany moduł może zostać wielokrotnie wykorzystany w innym module. Tworzenie instancji 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 3.3), zbudowano moduł dany kodem 3.1.7 (por. rysunek 3.4), który wykorzystuje dwie instancje pierwszego z modułów.



Rysunek 3.3: Moduł podstawowy



Rysunek 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 (jak dla modułu `gate_i1`). Proszę zwrócić uwagę na specyficzną **składnię modułu parametryzowalnego** 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). Oczywiście wraz ze wzrostem komplikacji modułów sprzętowych odchodzi się od projektowania strukturalnego na rzecz behawioralnego, czy wręcz generowania logiki na podstawie języków wysokiego poziomu tzw. HLS (ang. *High Level Synthesis*). Można to porównać do przejścia pomiędzy assemblerem, a językami typu C/C++ in nowszymi. Naukę 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:

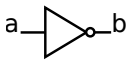
a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

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:



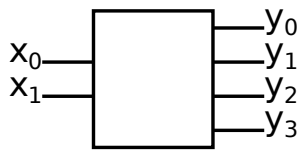
a	b
0	1
1	0

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

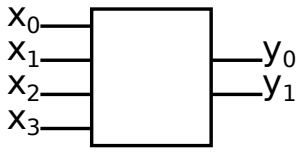
Kod 3.1.11 — Dekoder:

```
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: `assign y = warunek logiczny ? opcja 1 : opcja 2`. Będzie ono często wykorzystywane w ramach niniejszego kursu.

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:



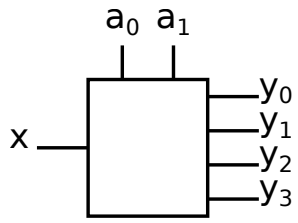
x_3	x_2	x_1	x_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

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 Demultiplekser

Demultiplekser 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:



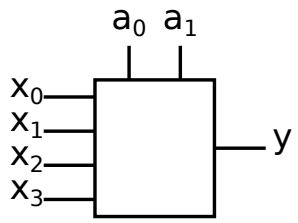
x	a_1	a_0	y_3	y_2	y_1	y_0
v	0	0	0	0	0	v
v	0	1	0	0	v	0
v	1	0	0	v	0	0
v	1	1	v	0	0	0

Kod 3.1.13 — Demultiplekser:

```
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 Multiplekser

Multiplekser 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:



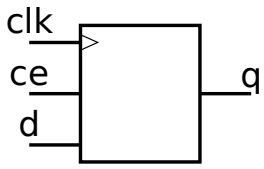
x_3	x_2	x_1	x_0	a_1	a_0	y
v_3	v_2	v_1	v_0	0	0	v_0
v_3	v_2	v_1	v_0	0	1	v_1
v_3	v_2	v_1	v_0	1	0	v_2
v_3	v_2	v_1	v_0	1	1	v_3

Kod 3.1.14 — Multiplekser:

```
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". Jest też elementem synchronicznym, tj. sposób jego pracy jest ściśle związany z sygnałem zegarowym. Wartość wyjścia nie zmienia się odpowiednio do każdej zmiany wejścia, ale zmiany są zsynchronizowane z narastającym (lub opadającym) zboczem zegara. Pomiędzy zboczami wartość wyjścia jest ustalona (zarejestrowana). Wartość wyjściowa jest opóźniona o jeden takt zegara w stosunku do wartości wejściowej. Dodatkowo możliwe jest włączanie/wyłączania 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	↑	q
v	1	↑	v

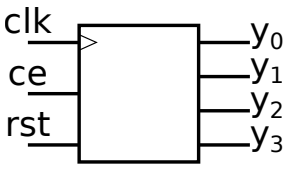
Kod 3.1.15 — Rejestr:

```
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, w odróżnieniu od ścieżek, należy **zawsze inicjalizować** wartością domyślną (zwykle 0). Znacząco ułatwia to późniejszą symulację modułu. Składania `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 (jak w typowym języku programowania). Dlatego można użyć polecenia `if else`. Poza blokami `always` wszystko wykonuje się równolegle (kolejność położenia modułów w kodzie nie ma znaczenia). Również poszczególne bloki `always` (procesy) wykonują się względem siebie równolegle. Zarejestrowaną wartość należy wyprowadzić na port wyjściowy (instrukcja `assign`).

3.1.14 Licznik

Licznik 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 takcie 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:

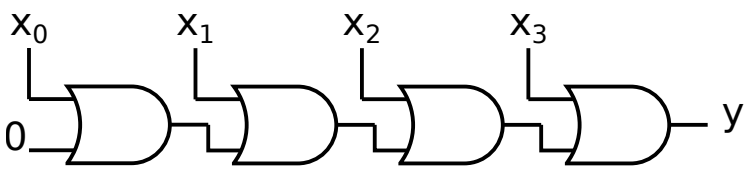
```
module cnt
(
  input clk,
  input ce,
  input rst,
  output [3:0]y
);
reg [3:0]val=4'b0; // init

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 przydatnych instrukcji jest **generate**. Jej działanie jest zbliżone do makra preprocesora 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 automatyczną generację fragmentów logiki, 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 parametryzowalnej liczbie wejść. Na rysunku 3.5 przedstawiono przykładowy moduł składający się z czterech bramek OR. Przedstawiony kod (kod 3.1.18) umożliwia generację odpowiedniej liczby dwuwejściowych bramek OR i połączenie ich w zadaną strukturę:



Rysunek 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
endgenerate
endmodule
```

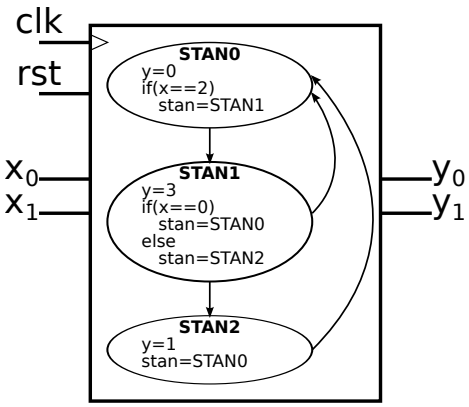
Kod 3.1.18 — Łańcuch bramek OR:

```
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
```

Podczas analizy modułu proszę zwrócić uwagę na rolę ścieżki `chain`.

3.1.16 Maszyna stanów

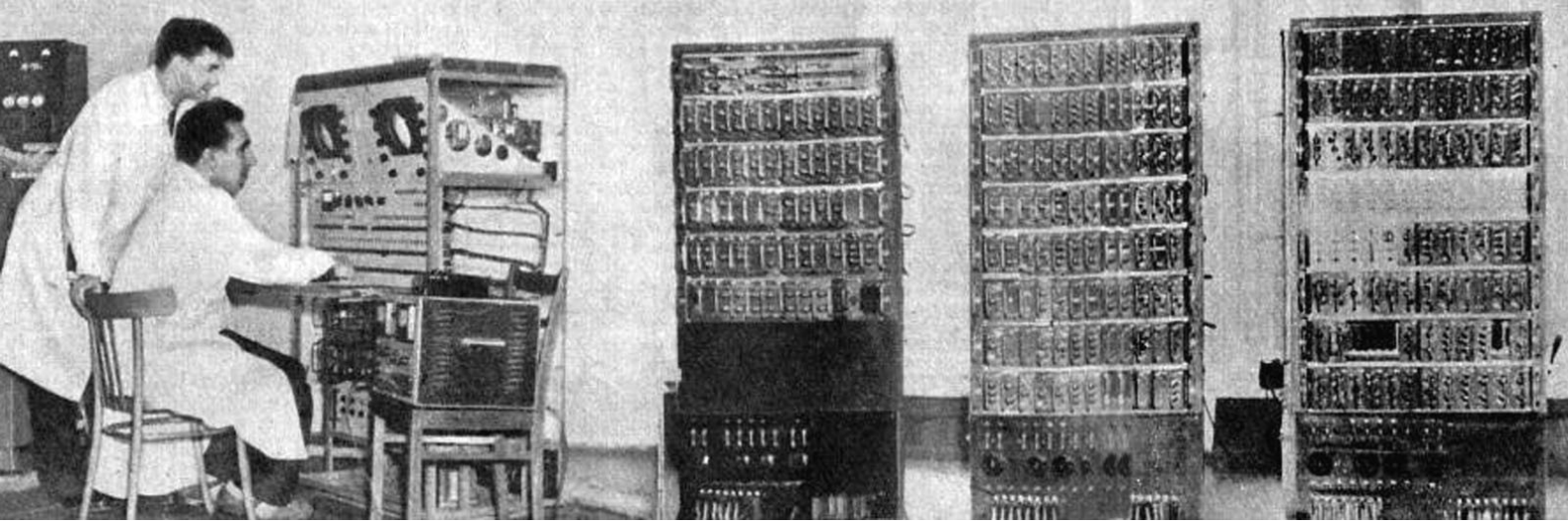
Maszyny stanów (ang. *Finite State Machines*) to bardziej złożone moduły, które poprzez sekwencję stanów mogą realizować praktycznie dowolną funkcjonalność. Są one wykorzystywane do realizacji protokołów komunikacyjnych, obsługi pamięci RAM, buforów FIFO i wielu innych celów. Wartość wyjścia jest zależna 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 (zakładamy, że mamy 3 stany). Przy pomocy polecenia **localparam** zdefiniowano trzy parametry (STATE0 – STATE2), 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.



Rysunek 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
```



4 — Weryfikacja i testowanie projektu

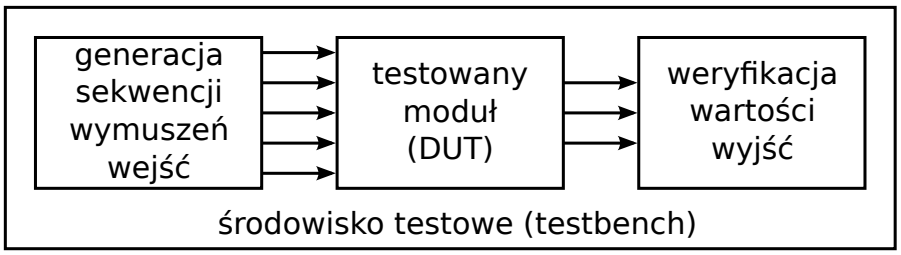
Testowanie i weryfikacja zaprojektowanych modułów sprzętowych jest bardzo **ważnym** i **złożonym** zagadnieniem. Wynika to bezpośrednio z zasygnalizowanego wcześniej problemu z oceną poprawności zaimplementowanego modułu sprzętowego. Po pierwsze, po uruchomieniu naszej logiki w układzie FPGA, zwykle dostajemy odpowiedź binarną, tj. moduł działa, albo nie działa. Czasami można się "domyślać" dlaczego coś nie działa, ale postępowanie typu "napiszmy kod i spróbujmy" jest sprzeczne z dobrą praktyką inżynierską i zwykle znacznie wydłuża, a nie skraca czas pracy nad projektem.

Po drugie, aby stwierdzić, że moduł został poprawnie zrealizowany należy wygenerować szereg sekwencji testowych (w idealnym przypadku trzeba sprawdzić wszystkie możliwe kombinacje sygnałów wejściowych). Podczas pracy układu FPGA trudno dostarczyć odpowiednie sekwencje testowe z wysoką częstotliwością, bez konieczności wykorzystania specjalizowanych urządzeń takich jak generatory sygnałów. Trudno jest również "podejrzeć" stan modułu wewnątrz układu FPGA, co ogranicza możliwości lokalizacji i usuwania błędów. Pewien wyjątek stanowi narzędzie ChipScope Pro. Jednak posiada ono pewne ograniczenia (np. maksymalny rozmiar bufora, konieczność transmisji danych do komputera), które uniemożliwiają jego wykorzystanie w przypadku systemów przetwarzania strumienia wizyjnego.

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 (analizę praktycznie każdego sygnału i rejestru), które mogłyby być trudne do weryfikacji w układzie pracującym z wysoką częstotliwością. Po drugie pozwala na przetestowanie wielu sytuacji, których wygenerowanie w działającym systemie mogłoby być kłopotliwe. Po trzecie, symulacja pozwala na zaoszczędzenie czasu potrzebnego na syntezowanie oraz implementację logiki do pliku bit, który umożliwia zaprogramowanie układu FPGA.

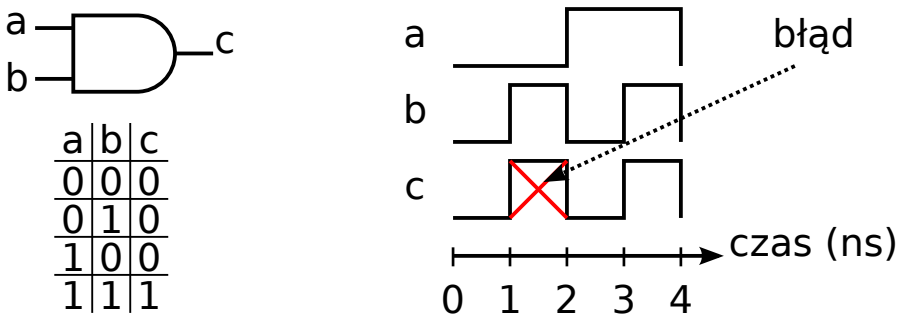
Ponieważ podstawowym elementem wykorzystywanym do opisu struktury układów scalonych w języku Verilog jest moduł, weryfikacja zaprojektowanego rozwiązania również opiera się o weryfikację działania poszczególnych modułów. W tym celu wykorzystywana jest metodologia przedstawiona na rysunku 4.1. Tworzone jest środowisko testowe (ang. *testbench*), które będzie podlegało symulacji. Jest ono zbudowane z trzech elementów. Testowany moduł, który oznacza się jako DUT (ang. *design under test*) lub UUT (ang. *unit 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).



Rysunek 4.1: Schemat środowiska testowego

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.



Rysunek 4.2: Testowanie bramki AND

Oczywiście w przedstawionym przypadku sprawdzenie poprawności jest bardzo proste. Łatwo jest określić wszystkie możliwe stany wejść i zauważyć błąd. Dla bardziej skomplikowanych modułów, które posiadają dziesiątki portów, rejestry opóźniające i maszyny stanów, określenie prawidłowych sygnałów stymulacyjnych oraz stwierdzenie czy otrzymane wyniki są poprawne, może stanowić nie lada wyzwanie (i zwykle wymaga dwóch warunków: cierpliwości i metodyczności).

4.1 Język Verilog – konstrukcje symulacyjne

Do tej pory poznaliśmy 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 (tzw. proces),
- **`assign`** – dla logiki asynchronicznej

Dla potrzeb 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 (tj. nie są "syntezowalne").

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 do 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), // out
  .b(b)  // out
);

and_gate dut
(
  .a(a), // in
  .b(b), // in
  .c(c)  // out
);

verify v_i
(
  .c(c) // in
);

endmodule
```

Moduł *stimulate* generuje sygnały, moduł *and_gate* jest testowaną bramką AND, a moduł *verify* jest odpowiedzialny za sprawdzanie poprawności stanu wyjścia. Symulacja w narzędziu ISE DS jest uruchamiana poprzez wybranie *View->Simulation*, zaznaczeniu pliku, który jest środowiskiem testowym w oknie hierarchii projektu oraz kliknięciu na *Simulate Behavioral Model* w oknie *Processes*. **Uwaga.** Proszę zawsze zwracać uwagę, który plik Państwo zaznaczacie przy uruchamianiu symulacji. Zwykle chcemy aby był to stworzony przez nas *testbench*, a nie któryś z modułów.

4.1.2 Generacja sekwencji testowych

Do opisu sekwencji wejściowej najlepiej użyć wyrażenia `initial begin end;`. Definiuje ono obszar, w którym kolejne instrukcje są wykonywane bezpośrednio w tym samym czasie, a do

przejścia do innego momentu w czasie wykorzystuje się instrukcje 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 (symulacja się "zawiesi"). Przykład użycia pętli `while` zaprezentowano w kodzie4.1.3. Wykorzystano ją do generacji sygnału zegarowego. Warto zwrócić uwagę, że w większości przypadków będziemy mieli do czynienia z tzw. logiką synchroniczną, której działanie uzależnione jest od sygnału zegarowego. Zatem moduł generacji zegara będzie występował w prawie wszystkich testbench'ach.

Uwaga. Blok `initial` można również wykorzystać w "zwykłym" module. W takim przypadku umożliwia on inicjalizację wybranych sygnałów lub rejestrów. Jest on wykonywanym tylko raz, w momencie uruchomienia logiki.

Do zapisu wartości wykorzystuje się rejestry (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 rejestry *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 czasowych, 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:

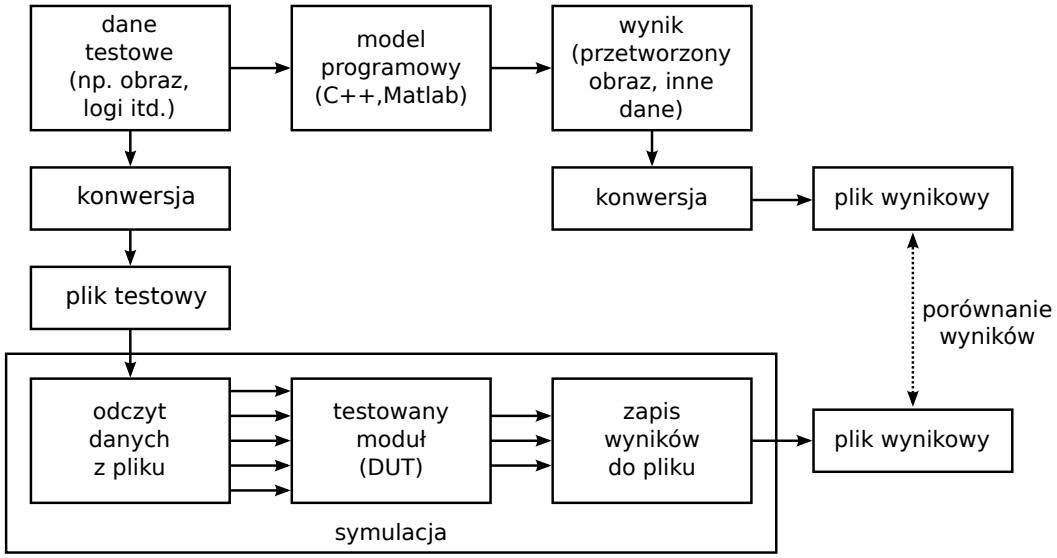
```
module verify
(
  input c
);

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 dość 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 wideo z kamery, segmentację obiektów pierwszoplanowych, czy choćby konwersję z przestrzeni barw RGB na YCbCr itp.), konieczne jest stworzenie tak zwanego modelu programowego zaprojektowanej 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 zwykle 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.



Rysunek 4.3: Model programowy

Wczytane wartości są następnie przetwarzane przez testowany moduł, a wyniki są zapisywane do pliku wynikowego. Porównanie wartości plików wyjściowych z modelu programowego 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. Należy zaznaczyć, że druga opcja, która wydaje się dość nieprawdopodobna, w praktyce zdarza się dość często. Podobnie jak 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, a w konsekwencji frustracji projektanta.

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.

Warto także podkreślić, że przy weryfikacji modułów trzeba być **metodycznym i cierpliwym.** Jeśli realizowany algorytm składa się z kilku etapów (tj. kilku modułów), testujemy rozpoczynając od pierwszego i dołączając kolejne moduły. Zapisujemy także "działające kopie" (ang. *working copy*). Staramy się także używać "reprezentatywnych" wektorów testowych (jeśli nie używamy wszystkich możliwych kombinacji). Z tego, że nasz moduł generuje prawidłowe wyniki dla jednego zestawu danych (np. podajemy na wejście cały czas ten sam obraz) nie można wnioskować o jego poprawności. "Droga na skróty", czyli napisanie całej logiki i symulacja, czasem się sprawdza, ale częściej uzyskujemy błędne wyniki i ostatecznie i tak musimy analizować poprawność każdego fragmentu osobno (co powoduje niepotrzebną frustrację). Identyczna uwaga odnosi się również do uruchamiania logiki w sprzęcie. Przy czym, jeśli wyniki symulacji wskazują na poprawność implementacji to możemy się pokusić o **jedną** próbę uruchomienia całości. Jeśli się ona nie powiedzie, to stosujemy podejście z dołączeniem kolejnych modułów. Temat ten zostanie jeszcze poruszony i rozwinięty w dalszej części skryptu.

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 wbudowanych (ang. `embedded`). W związku z tym, model programowy w podstawowej wersji, istnieje już przed podjęciem prac

nad implementacją sprzętową (najlepiej gdy autorami tego algorytmu jesteśmy my sami, gdyż jedynie samodzielna implementacja pozwala w pełni zrozumieć niuansy niektórych algorytmów). 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 **??**).

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`.

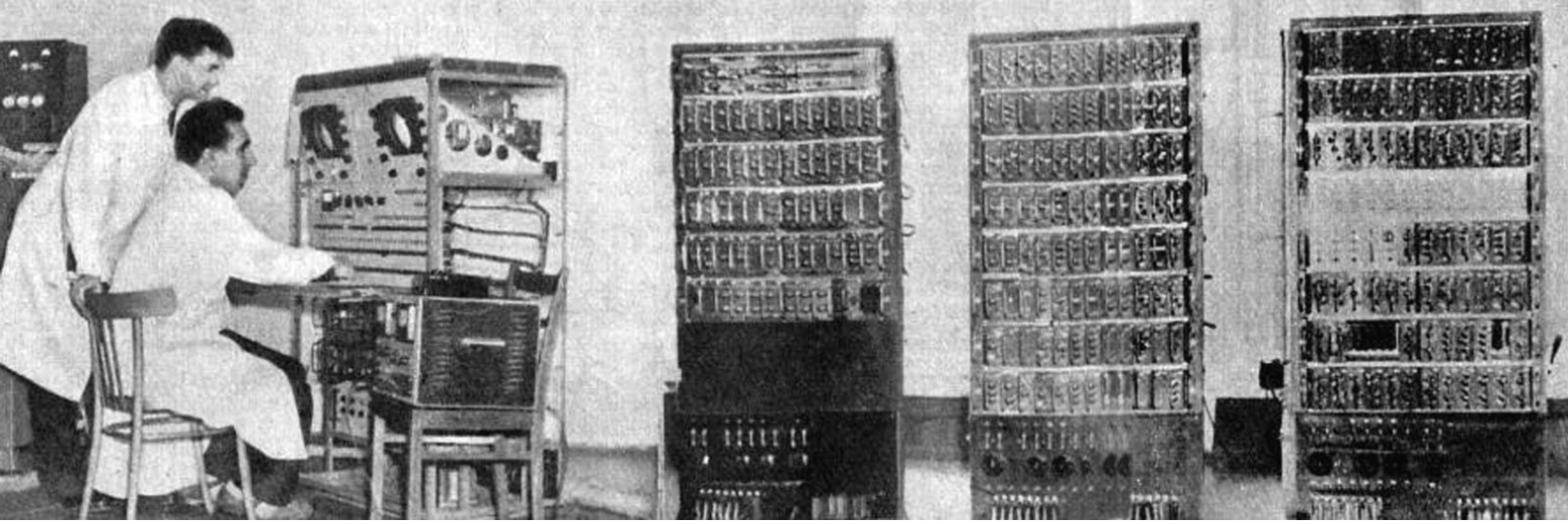
Moduł, który umożliwia odczytanie czterech binarnych wartości z pliku oraz ich przypisanie do wyjść *a* i *b*, został przedstawiony w kodzie 4.2.1. Natomiast moduł, który umożliwia zapisanie wartości portu *c* do pliku wynikowego zaprezentowano w kodzie 4.2.2.

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 zadnia, 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" system z poprawnych komponentów. Warto zatem poświęcić więcej czasu na testy cząstkowe, niż po złożeniu całego systemu głowić się "dlaczego to nie działa" (co i tak sprowadza się do testów cząstkowych). Proszę zwrócić uwagę, że takie podejście dobrze stosować przy każdym projekcie informatycznym, nie tylko tworzeniu logiki FPGA.

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. Uwaga. Na potrzeby tak prostego testu **nie warto** implementować osobnych modułów generujących dane i sprawdzających wyniki. ■

Podpowiedzi:

- utwórz nowy projekt w ISE oraz dodaj do niego nowy plik Verilog, w którym znajdzie się opis parametryzowalnej 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 (dużo pisania kodu). 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 syntezy modułu oraz sprawdź użycie zasobów (*Slice LUT's* i *LUT-FF pairs*). Czy mamy do czynienia z modułem synchronicznym, czy asynchronicznym ?
- w celu lepszego zrozumienia powiązania pomiędzy opisanym kodem, a faktycznymi zasobami układu FPGA oglądniemy 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 (trudniej się pomylić przy uruchamianiu symulacji). 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 nadrzędnego, w którym wygenerujemy sygnał testowy (wszystkie możliwe kombinacje sygnałów wejściowych) i instancji modułu testowanego (bramki AND). Poprawność sprawdzimy "ręcznie" – 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*),
- Przebieg sygnałów, ew. kod.

W pierwszej tj. *Instance and Process Name* uzyskujemy dostęp do hierarchii symulowanego modułu. W naszym przypadku mamy nadrzędny 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 i je analizować. Warto zauważyć, że domyślnie umieszczone są tam sygnały zdefiniowane w testbench'u (zegar oraz wejście i wyjście z modułu AND).

Dodamy teraz pomocniczy sygnał *chain*. Warto zauważyć, 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. Jest on istotny przy modułach IPCore (będą omówione później), gdyż dopiero po tym czasie rozpoczynają "normalne" działanie. 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. O tym kiedy dokładnie potrzebny jest reset symulacji można przekonać się "empirycznie", po prostu aplikacji ISim wygeneruje błąd, jeśli nasza ingerencja w kod była "zbyt poważna".

- 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.
- warto czytać komunikaty w konsoli – zarówno błędy jak i ostrzeżenia. Pozwala to "wychwycić" np. niepoprawne szerokości użytych sygnałów.
- 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 przykładach w rozdziale 4). Warto także zmodyfikować instancję *uut*, tak aby można było podawać parametr N modułu licznika. Przykład jak to zrobić przedstawiono poniżej:

Kod 5.1.1 — Przykład instancji parametryzowalnego 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 z wyjściem z modułu. Jego długość należy określić ręcznie, albo ew. za pomocą deklarowania parametrów w sposób zbliżony do zastosowanego w module licznika. Proszę przetestować licznik dla co najmniej dwóch wartości parametru N.

Symulator ISim umożliwia także pracę z kodem. Przejdź 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 prawie identyczne. Dobrze jednak edytować kod tylko w ISE, gdyż inaczej dość łatwo sobie "namieszać" (albo coś nadpisać, albo czegoś nie zapisać). 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.

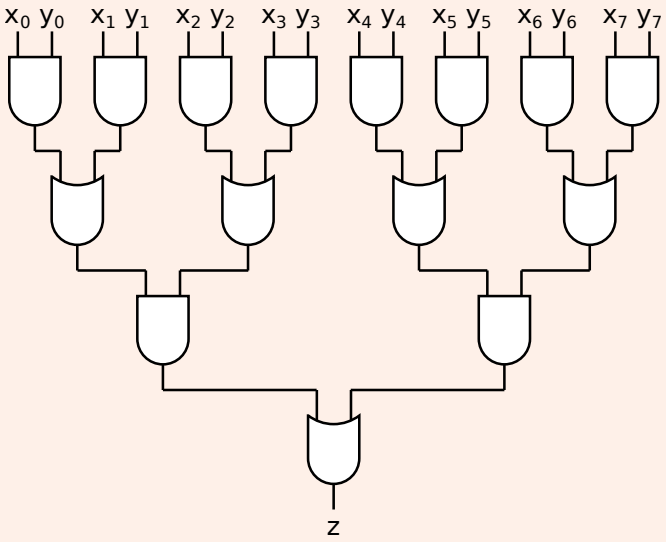
- możliwość "podglądania" wartości zmiennych – po ustawieniu na niej kursora,
- pracę krokową – F11.

Warto pamiętać o tej funkcjonalności, gdyż czasami bywa przydatna (np. analiza działania 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:



Proszę zwrócić uwagę, że na schemacie występują dwa typu bramek. 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. Rozwiązanie za pomocą jednej instrukcji `generate` wymaga trochę "gimnastyki" indeksami. Jednakże implementacja bez `generate` nie spełnia warunków zadania. Operator modulo w Verilog jest taki sam jak w C/C++.

Uwaga. Warto podglądnąć 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 rejestry mogą zostać wykorzystane do realizacji linii opóźniającej. Rozwiązanie takie zaprezentowano na rysunku 5.1.

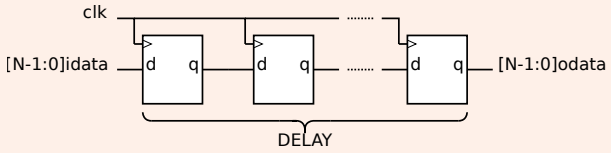
Proszę zaprojektować moduł, który posiada dwa parametry:

- 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 – generował DELAY bloków rejestrów o szerokości N, połączonych jak na rysunku 5.1

Pozostały interfejs proszę wykonać analogicznie jak dla modułu opóźniającego z rozdziału 3.1.13.



Rysunek 5.1: Linia opóźniająca

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

Uwaga. Moduł będzie wykorzystywany w ramach dalszej części kursu.

■

Zadanie należy zacząć od stworzenia modułu pojedynczego opóźnienia – analogicznie jak w rozdziale 3.1.13, przy czym trzeba "uzmiennić" szerokość szyny 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. Przykładowa składnia:

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

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

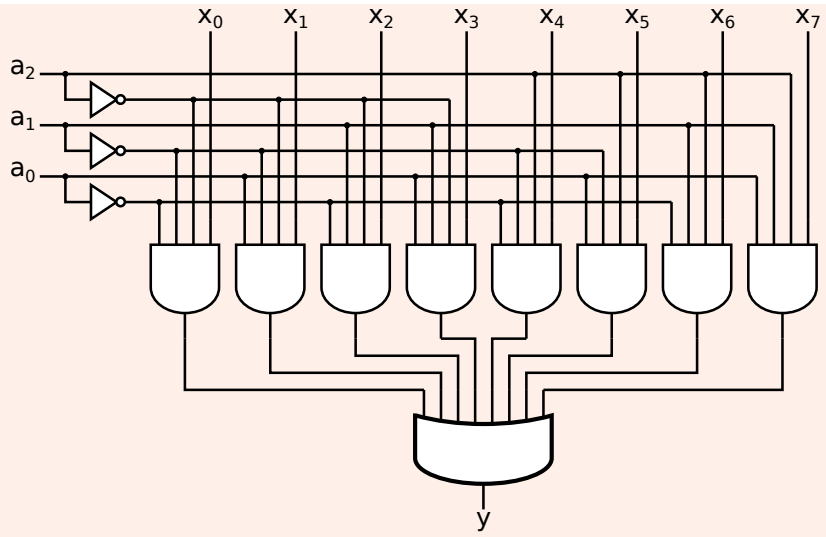
```
.idata(tdata[i]),
.odata(tdata[i+1])
```

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

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

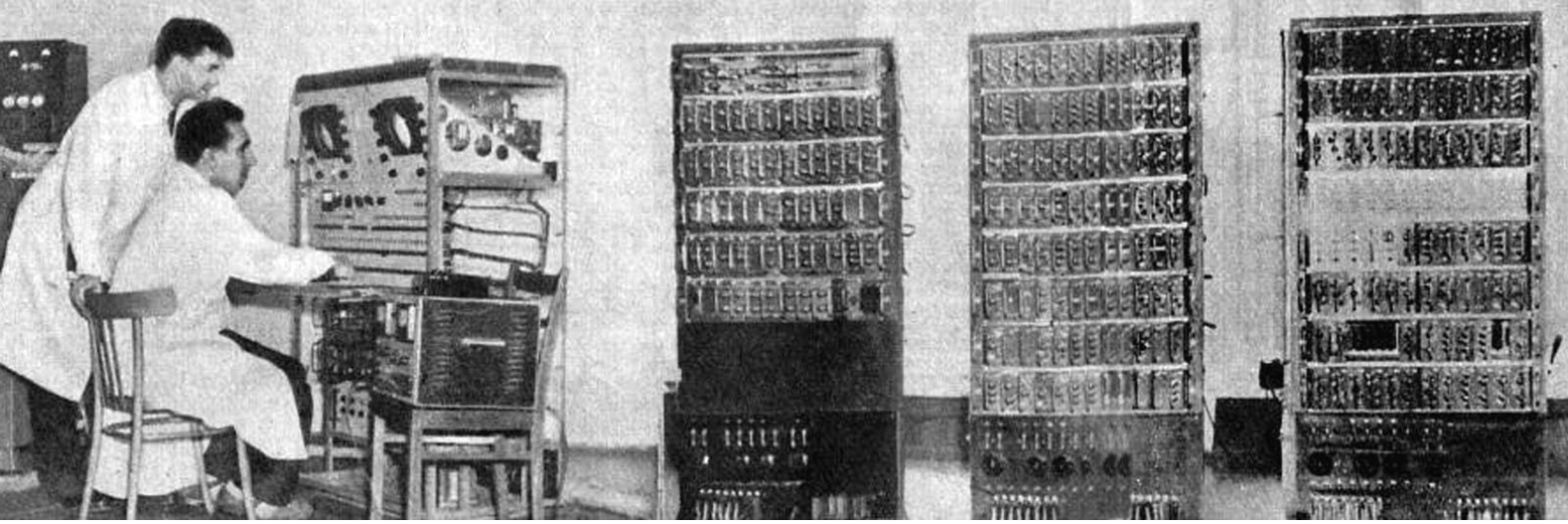
5.2.2 Tajemniczy moduł

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 postawione w treści pytanie. Analiza funkcjonalności modułu może znacznie ułatwić jego implementację.



6 — Maszyny stanowe i zaawansowane testowanie

W tym rozdziale zamieszczono zadania dotyczące realizacji maszyn stanowych w języku Verilog oraz zaawansowanego testowania m.in. odczytu i zapisu danych z/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 (1 bit).

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 oraz sygnału *send*.
2. W stanie drugim, na wyjście *txd* powinna zostać podana wartość '1' (bit startu) oraz powinno nastąpić przejście do stanu trzeciego.
3. W trzecim stanie, na wyjście *txd* powinny być przesyłane kolejne bity portu data z utworzonego rejestru, 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' (bit stopu) oraz maszyna powinna wrócić do stanu pierwszego.

Opisana maszyna realizuje bardzo uproszczoną wersję UART (ang. *Universal Asynchronous Receiver and Transmitter*). Wejściowe dane (8 bitów) są serializowane (zamieniane z postaci równoległej na szeregową) i wysyłane. Dodatkowo dodawane są bity startu i stopu. W ten sposób wykonywana jest transmisja z wykorzystaniem standardu RS-232.

■

Wskazówki:

- należy zacząć od nowego projektu w ISE.
- do realizacji detekcji narastającego zbocza sygnału *send* warto zapamiętać (w rejestrze) jego poprzednią wartość i sprawdzać z "aktualną".
- 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`. Uwaga. Stan 3 (wysyłanie danych) ma się "wykonać" 8 razy, a nie zawierać pętlę `for`.

Zadanie 6.2 W zadaniu 6.1 zaprojektowano maszynę stanów umożliwiającą serializację danych (tj. zamianę z 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 innego 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++ (lub dowolnym innym języku), który dokona serializacji danych z pliku wejściowego i zapisze dane do pliku wyjściowego. Będzie to nasz programowy model referencyjny.

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

Wskazówki:

- zaczynamy od wygenerowania nadrzędnego 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 (#) (1 ns stan wysoki i 1 ns stan niski). 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 takt ustalamy dla bezpieczeństwa. Podsumowując - przy odczycie 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. znaki '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 dodatkowe 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 *rxd* (dane) oraz dwa wyjścia *data* i *received* (flaga ustawiana po otrzymaniu paczki danych). Proszę również stworzyć moduł, który będzie zapisywał otrzymane 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 (całkowitoliczbowe, stałoprzecinkowe), float, double (zmiennoprzecinkowe)), 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 (wynika to wprost z dostępnych zasobów obliczeniowych). Natomiast w układzie reprogramowalnym może zostać wykorzystany sumator 17-bitowy. Zaoszczędzone w ten sposób zasoby są wystarczające do realizacji np. 8-bitowego sumatora, które może być użyty w innym miejscu tworzonego systemu.

Zaczniemy zatem od przypomnienia bitowego formatu zapisu liczb całkowitych i stałoprzecinkowych 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:

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

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.

$$\mathrm{A} = \underbrace{\left\| c_6 \mid c_5 \mid c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0 \right\|}_{N_c=7} \qquad \mathrm{B} = \underbrace{\left\| c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0 \right\|}_{N_c=5}$$

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 wejściowych 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 wyniku bez straty informacji, np. dla ułamka $1/3$. Temat zapisu liczb ułamkowych zostanie przedstawiony w rozdziałach 7.1.3 i 7.1.4.

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

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 pewną 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:

$$w = -z2^{N_c} + \sum_{i=0}^{N_c-1} c_i 2^i \tag{7.2}$$

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 $\mid c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0$
 N_z=1 N_c=5

B = z $\mid c_3 \mid c_2 \mid c_1 \mid c_0$
 N_z=1 N_c=4

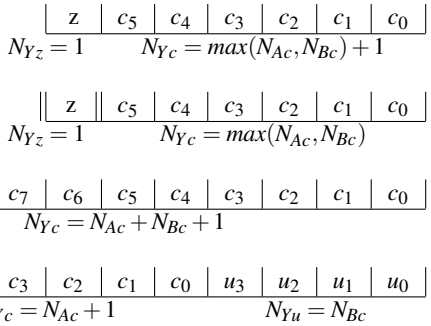
wartość minimalna: -32
wartość maksymalna: 31

wartość minimalna:- 16
wartość maksymalna :15

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

operacja	wartość	
	min.	maks.
Y=A+B	-48	46
Y=A-B	-47	47
Y=A*B	-496	512
Y=A/B	-32	32

format



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). Słowo 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:

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

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

wartość minimalna: 0
wartość maksymalna: 31,75
rozdzielczość części ułamkowej: 0,25

wartość minimalna: 0
wartość maksymalna: 15,875
rozdzielczość części ułamkowej: 0,125

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

operacja	wartość	
	min.	maks.
Y=A+B	0	19.625
Y=A-B	-3,875	15,75
Y=A*B	0	61,03125
Y=A/B	0	126

format

c_4	c_3	c_2	c_1	c_0	u_2	u_1	u_0

$N_{Yc} = max(N_{Ac}, N_{Bc}) + 1$ $N_{Yu} = max(N_{Au}, N_{Bu})$

z	c_3	c_2	c_1	c_0	u_2	u_1	u_0

$N_{Yz} = 1$ $N_{Yc} = max(N_{Ac}, N_{Bc}) + 1$ $N_{Yu} = max(N_{Au}, N_{Bu})$

c_5	c_4	c_3	c_2	c_1	c_0	u_4	u_3	u_2	u_1	u_0

$N_{Yc} = N_{Ac} + N_{Bc}$ $N_{Yu} = N_{Au} + N_{Bu}$

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 dokładności.

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:

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

Poniżej przedstawiono sposób kodowania dla dwóch wektorów:

A = | z | c_3 | c_2 | c_1 | c_0 | u_1 | u_0 |
N_z=1 N_c=4 N_u=2

wartość minimalna: -16,0
wartość maksymalna: 15,75
rozdzielczość części ułamkowej: 0,25

B = | z | c_2 | c_1 | c_0 | u_2 | u_1 | u_0 |
N_z=1 N_c=3 N_u=3

wartość minimalna: -8,0
wartość maksymalna: 7,875
rozdzielczość 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	
	min.	maks.
Y=A+B	-10,0	9.625
Y=A-B	-9,875	9,75
Y=A*B	-15,5	16,0
Y=A/B	-64	64

format

z	c_3	c_2	c_1	c_0	u_2	u_1	u_0	
N_z=1	$N_{Yc} = max(N_{Ac}, N_{Bc}) + 1$				$N_{Yu} = max(N_{Au}, N_{Bu})$			

z	c_3	c_2	c_1	c_0	u_2	u_1	u_0	
$N_{Yz} = 1$	$N_{Yc} = max(N_{Ac}, N_{Bc}) + 1$				$N_{Yu} = max(N_{Au}, N_{Bu})$			

z	c_4	c_3	c_2	c_1	c_0	u_4	u_3	u_2	u_1	u_0
$N_{Yz} = 1$	$N_{Yc} = N_{Ac} + N_{Bc} + 1$						$N_{Yu} = N_{Au} + N_{Bu}$			

z	c_6	c_5	c_4	c_3	c_2	c_1	c_0	u_3	u_2	u_1	u_0
$N_{Yz} = 1$	$N_{Yc} = N_{Ac} + N_{Bu} + 1$							$N_{Yu} = N_{Au} + N_{Bc} + 1$			

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 ze zmienną długością słowa? 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 zmienno-przecinkowego na stałoprzecinkowy. Załóżmy, że mamy dany prosty algorytm operujący na liczbach typu *double* lub *float*. Oprócz bardzo specyficznych obliczeń, taka reprezentacja jest zwykle zupełnie satysfakcjonująca i uznawana za dokładną. Chcemy teraz przejść na format stałoprzecinkowy. Musimy podjąć dwie decyzje. Po pierwsze trzeba ustalić ile bitów przeznaczymy na część całkowitą. Decyzja ta zwykle jest dość prosta, musimy tylko oszacować maksymalne i minimalne wartości jakie mogą wystąpić na każdym etapie obliczeń.

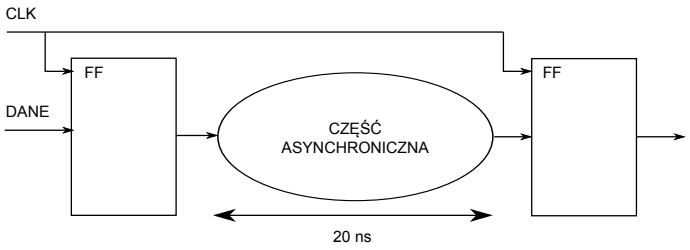
Po drugie, trzeba określić liczbę bitów, które przeznaczymy na część ułamkową. To już jest trudniejsze, gdyż precyzji *double* w ten sposób nigdy nie osiągniemy. Musimy się pogodzić z pewną utratą dokładności. Zatem zwykle tworzy się model stałoprzecinkowy algorytmu i empirycznie sprawdza, jak precyzja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

Zasygnalizowana metodologia zostanie zademonstrowana praktycznie w ramach niniejszego kursu.

7.3 Latencja

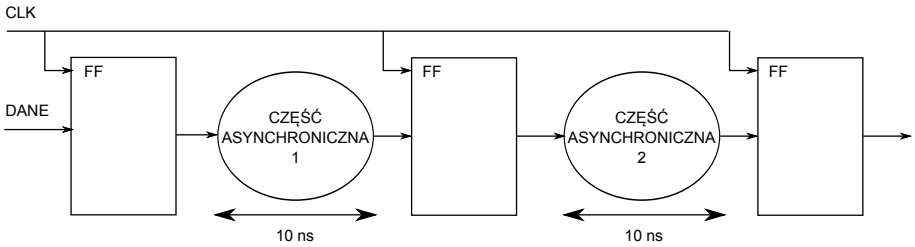
Oprócz formatu słowa, w którym są wykonywane obliczenia, drugim najważniejszym parametrem opisującym elementy wykonujące operacje arytmetyczne jest latencja. **Latencja** to liczba taktów zegara, która upływa od pojawienia się wartości na wejściu do ustalenia się prawidłowego rezultatu na wyjściu.

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.



Rysunek 7.1: Logika 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 wprowadzenie dodatkowego przerzutnika. Przykład zaprezentowano na rysunku 7.2.

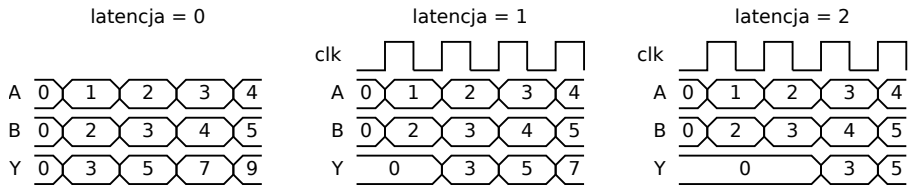


Rysunek 7.2: Logika synchroniczna i asynchroniczna – podział

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 wyjścia. Oczywiście nie oznacza to, że dzieje się to "w nieskończenie 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 zauważyć, że w przypadku latencji, opóźnione są wartości wejścia od wyjścia, natomiast poszczególne wyniki nie są od siebie opóźnione.



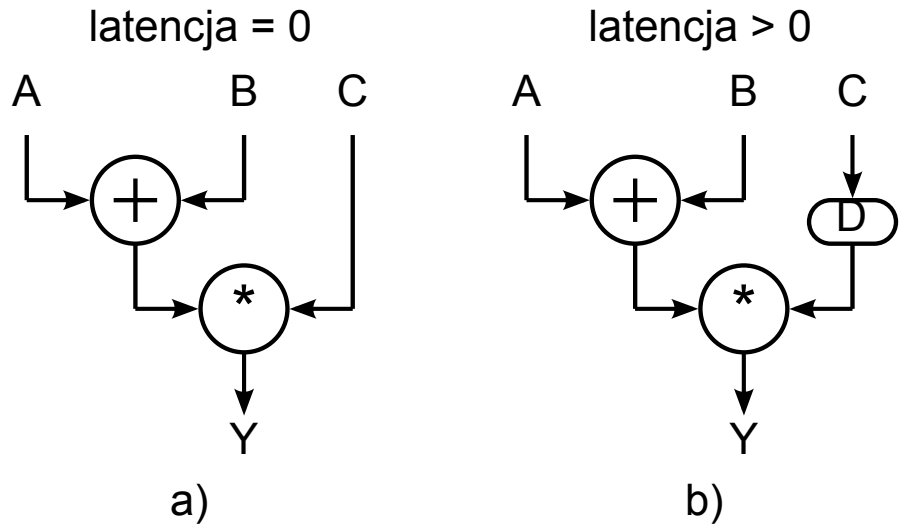
Rysunek 7.3: Wykonywanie obliczeń przez sumator z latencją równą 0, 1 i 2 taktom zegara

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

$$Y = (A + B) * C \tag{7.5}$$

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 takcie 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.



Rysunek 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, co jest niekorzystne. Ponadto, gdy połączy się dużo elementów o zerowej latencji, czas tych opóźnień sumuje się. 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. Rozbicie 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 wprost

w logice (LUT) lub za pomocą dedykowanych zasobów – mnożarek (DSP). Zostanie to pokazane w dalszej części kursu.

	oparte o LUT			oparte o DSP		
Latencja	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

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

W praktyce projektowania systemów wykonujących operacje arytmetyczne w układach FPGA, przyjmuje się, że jeśli tylko jest to możliwe (nie ma bardzo dużych ograniczeń na zasoby) to należy wykorzystywać elementy obliczeniowe 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 syntezowalne [1], 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 predefinowanych bloków logicznych (aplikacja *CORE generator*) i wygenerowania ich jako bloków *IP*

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

Core (czarna skrzynka, w której nie da się nic zmienić). Uwaga. Narzędzie *CORE generator* zostanie omówione w ramach dalszej części kursu. *CORE generator* pozwala również na wygenerowanie sprzętowych dzielarek, które mogą dokonywać dzielenia dowolnych dwóch liczb (stałoprzecinkowych).

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 posiada np. sprzętowej instrukcji obliczania logarytmu. Funkcja ta jest obliczana przez bibliotekę programową, która rozwija w szereg odpowiednie przybliżenie numeryczne. W układach FPGA również nie występują wyspecjalizowane moduły sprzętowe, do obliczania tego typu funkcji. Operacje takie realizuje się poprzez wygenerowanie elementu typu IP Core w narzędziu CORE generator lub samodzielne zaprojektowanie odpowiedniego modułu.

7.5.1 Tablicowanie wartości funkcji

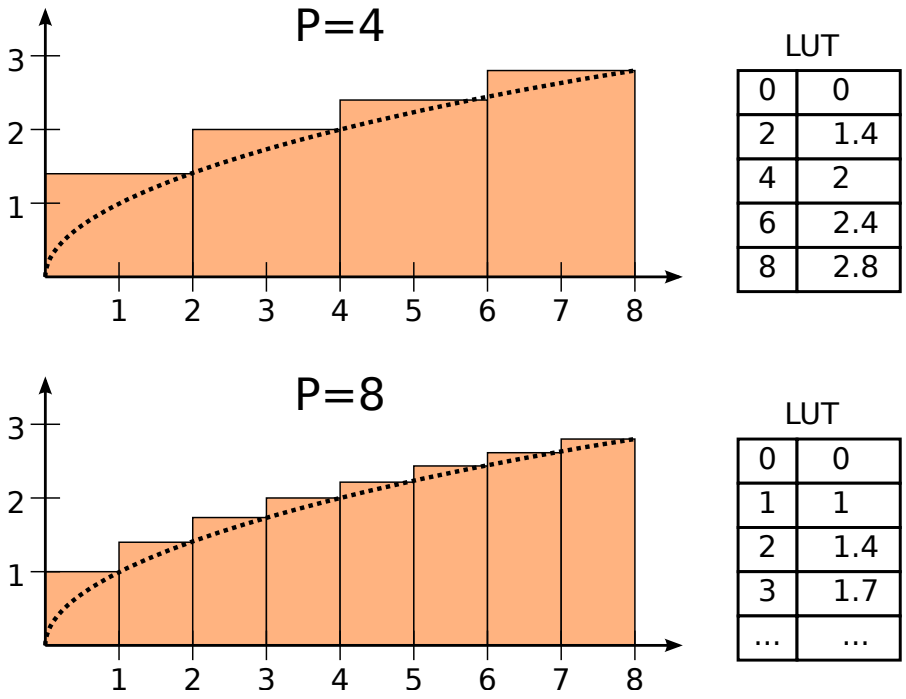
Bezpośrednie obliczanie wartości skomplikowanych funkcji na platformie rekonfigurowalnej jest utrudnione, wymaga bowiem zastosowania dużej liczby elementów logicznych i wielu iteracji metody przybliżonej. W związku z tym, często stosowanym podejściem, umożliwiającym realizację tych obliczeń w prosty 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. Jednakże 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 (ROM – ang. *read-only memory*). 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 dla wartości bliskich 0. W drugim przypadku funkcja jest tablicowana dla 8 przedziałów, rozmiar pamięci, która jest konieczna do ich przechowania jest dwukrotnie większy. Zmniejszył się jednak błąd powstający dla wartości bliskich 0.

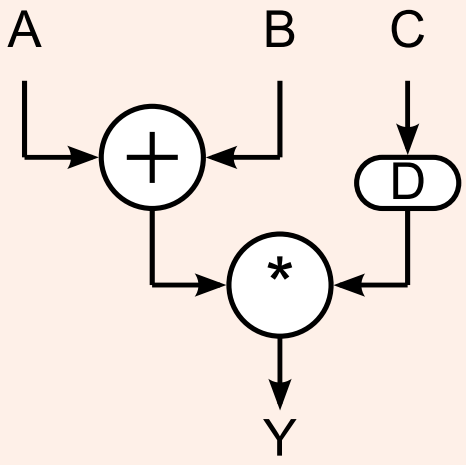


Rysunek 7.5: Tablicowanie wartości funkcji $y = \sqrt{x}$

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 (np. liniową) pomiędzy przedziałami histogramu, co pozwala poprawić dokładność.

7.6 Zadania do wykonania na laboratorium

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



Rysunek 7.6: Prosta operacja arytmetyczna

Załóż, 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 "wybieralną" 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*, symulacja rozwiązania i implementacja obliczeń w sprzęcie.
- W pierwszym kroku należy uruchomić pakiet Matlab, stworzyć nowy 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 (przedział jest zamknięty), 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 z 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       % sign
prec_i=1; %number of integer part bits (Nc)     % one bit
prec_f=8; %number of fractional part bits (Nu)  % eight bits
word = 1 + prec_i + prec_f;                     % whole word

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*). Ponadto wygodnie jest wprowadzić pomocniczą wartość – liczbę bitów przeznaczoną na część całkowitą (*prec_i*). Funkcja 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)` lub `o_fix.bin`.

- Potrzebujemy również 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);
```

- Dokonaj konwersji trzech uzyskanych poprzednio liczb stałoprzecinkowych na format *double*. Porównaj tak uzyskane wartości z wejściowymi – tj. czy nastąpiła jakaś utrata dokładności.
- 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,sign,word,prec_f);
      y = fi(1.45,sign,word,prec_f);
      z=x+y

  z =
    1.7695
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 11
        FractionLength: 8
```

Uwaga. W przykładzie użyto takich samych parametrów jak podane w 7.6.1.
Zrealizuj dodawanie liczb A i B w dwóch wariantach: zmienno i stałoprzecinkowym. Porównaj uzyskane wyniki – np. wypisz obok siebie na konsolę 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 różnicy 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. Uwaga 1. Otrzymany wykres pokazuje, że nie z każdym zwiększeniem precyzji wiąże się poprawa dokładności. Osoby dociekliwe mogę przeanalizować przyczynę takiego stanu poprzez analizę wartości dla wartości `prec_f` w zakresie 0 do 5. Uwaga 2. Wybór precyzji jest zawsze kompromisem pomiędzy dokładnością obliczeń, a użytymi zasobami logicznymi. Pierwszym kryterium jest zawsze stabilność numeryczna algorytmu. W drugiej kolejności rozpatruje się wpływ precyzji na "wyniki". Przy czym "wyniki" są różnie definiowane, w zależności

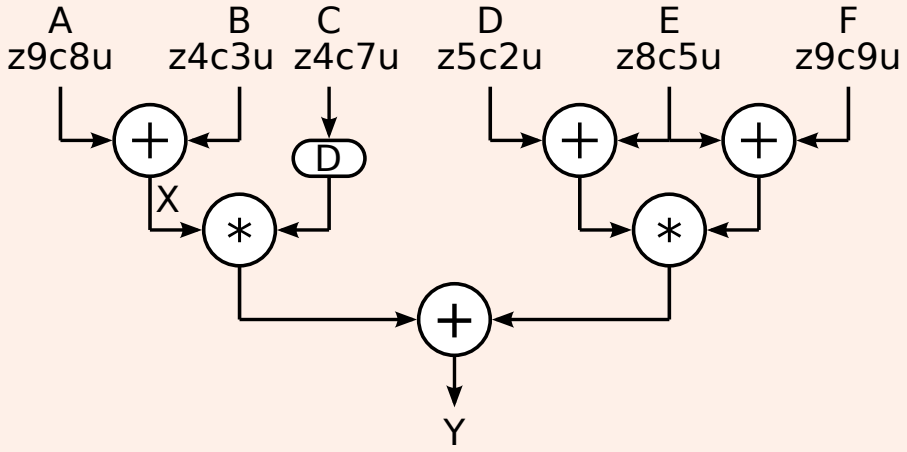
od aplikacji. Przykładowo mając do czynienia z detekcją twarzy, będziemy analizować jak użyty format stałoprzecinkowy wpływa na skuteczność detekcji. W naszym, dość prostym przypadku, praktycznie nie da się wybrać "źle", ale załóżmy, że błąd ma być raczej niewielki.

- 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 argumentów i wyniku ustawiamy zgodnie z ustaloną uprzednio precyzją. Uwaga. Należy się zastanowić nad tzw. najgorszym przypadkiem tj. np. wartość A i B = 1.
- W języku Veilog występuje typ `signed` tj. można np. zadeklarować połączenie ze znakiem – `wire signed [2:0] x;`. W przypadku operacji na liczbach ze znakiem należy go stosować, 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!** . Dodatkowo, są one dość trudne do wykrycia (choć uważana analiza raportu syntezy pozwala to wychwycić). Połączeń **nie należy** inicjalizować (w odróżnieniu od rejestrów). Przypisanie do *wire'a* wartości 0 powoduje stałe podłączenie tego sygnału do masy. Jeśli później będziemy chcieli coś do tego *wire'a* przypisać to w symulacji otrzymamy stan nieustalony 'X'.
- 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 Subtracter* (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 (w formie komentarza).
- 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. 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 (logiki ogólnego przeznaczania), czy DSP (Mult) – dedykowanych zasobów arytmetycznych. 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 (najprościej w postaci binarnej). Sprawdzamy czy uzyskujemy poprawny wynik. Dodatkowo sprawdzamy, czy wyniki pojawia się po tylu taktach zegara, po ilu się go spodziewamy. Uwaga. Moduły IP Core "stratują" dopiero po początkowym czasie inicjalizacji 100 ns. Uwaga. Do eliminacji błędów bardzo przydaje się model programowy. Dzięki niemu mamy wszystkie wyniki po-

średnie i możemy precyzyjnie zlokalizować na którym etapie występuje nieprawidłowość. Warto pamiętać o możliwościach jakie daje narzędzie ISim: podgląd wartości wszystkich sygnałów i rejestrów użytych w projekcie (por. podpowiedzi do zadania 5.1).

7.7 Zadania do wykonania w domu

Zadanie 7.2 Proszę zaprojektować moduł pokazany na rysunku 7.7.



Rysunek 7.7: Złożony moduł arytmetyczny

Realizacja ćwiczenia – podpowiedzi i uwagi:

- Proszę przerysować 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};
  ```

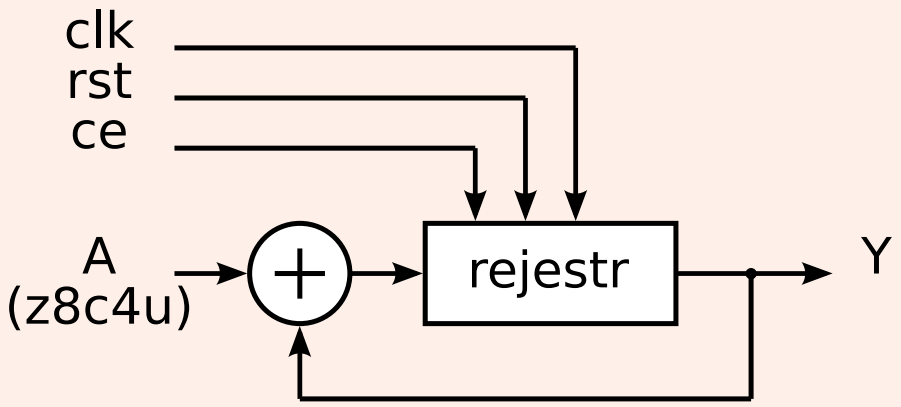
  Jak w związku z tym należy zmodyfikować formaty na wejściach i wyjściach do poszczególnych sumatorów (na kartce)? Zmodyfikuj sumator (IP CORE). Sprawdź czy teraz wynik jest poprawny.
- Proszę sprawdzić (symulacyjnie) czy konieczne jest takie rozszerzanie wektorów w przypadku mnożarek? Należy postąpić 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 (polecenie *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



Rysunek 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 rejestru. Wystąpienie sygnału *rst* powinno umożliwić wyzerowanie wartości w rejestrze. Przyjmijmy założenie, że rozmiar rejestru akumulacyjnego 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.

■

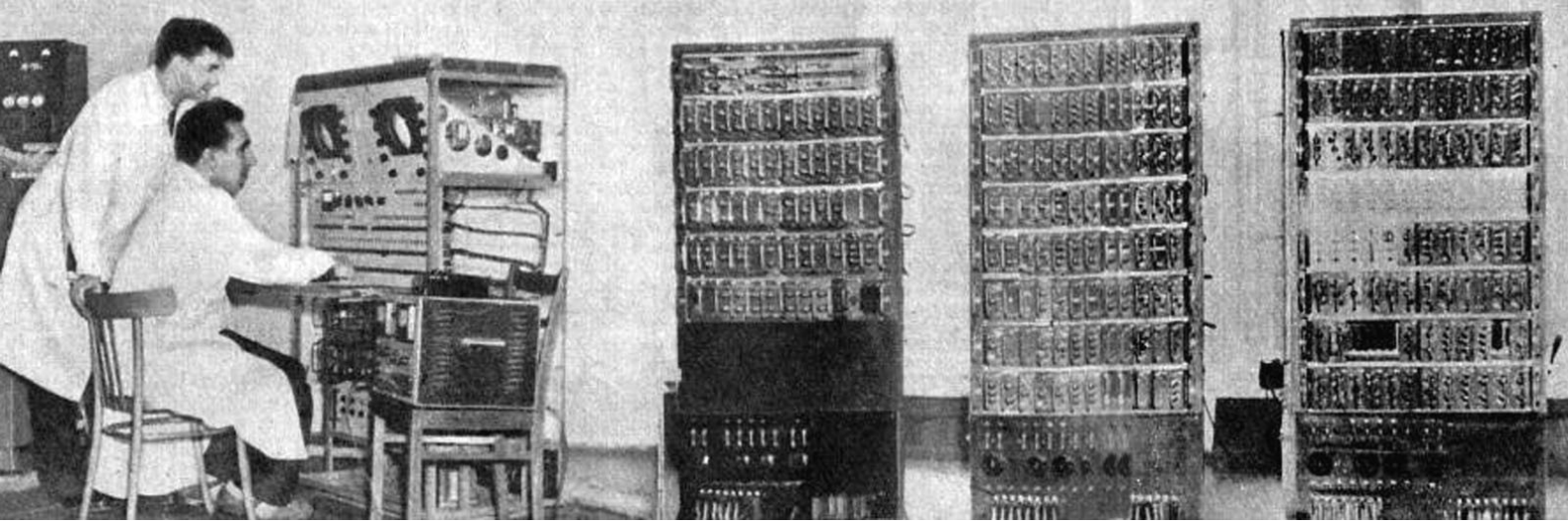
7.8 Zadania dodatkowe

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

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

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 do urządzenia wykonawczego (np. sterownika robota mobilnego, systemu sterującego sygnalizacją świetlną lub procesem produkcji). 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

Rysunek 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, zakłada się, że informacja nie jest tracona. Cały tor wizyjny wprowadza jedynie pewną, zwykle niewielką, latencję (dla przypomnienia – opóźnienie). Warto podkreślić, że w przetwarzaniu zrealizowanym na procesorze ogólnego przeznaczenia (np. w pakiecie Matlab lub bibliotece OpenCV) zwykle podstawową jednostką na której się operuje jest ramka obrazu. Ma to związek z urządzeniami do akwizycji, względnie programami do dekompresji obrazu, które dostarczają właśnie strumień ramek, a nie pojedynczych pikseli.

Szczegółowe omówienie poszczególnych etapów systemu wizyjnego 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, RGB ->CIE Lab), filtracja Gaussa (uśredniająca, dolnoprzepustowa), detekcja krawędzi (Sobel, Canny), filtracja medianowa oraz różne operacje morfologiczne (erozja, dylatacja). Posiadają one jedną wspólną cechę – informacja na wyjściu i wejściu jest zasadniczo taka sama (pewne wyjątki stanowią konwersja obrazu kolorowego do odcieni szarości oraz detekcja krawędzi).

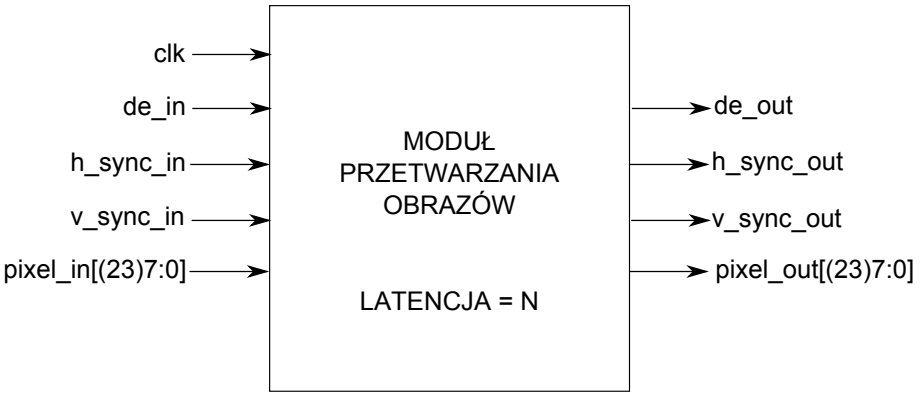
Natomiast analiza obrazu to proces wydobywania z niego istotnych informacji. Przy czym "istotność" ta jest ściśle powiązana z docelową aplikacją. Przykładami są: segmentacja obiektów (podział sceny na poszczególne obiekty: ludzi, samochody itp.), indeksacja (przypisanie do poszczególnych pikseli etykiet) 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 pola obiektów). Następuje tutaj zwykle znaczna redukcja informacji.

Ostatni etap to rozpoznawanie, w którym na podstawie cech obiektów (zebranych zazwyczaj w tzw. wektor cech) dokonuje się klasyfikacji obiektów, przykładowo 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 (ang. *Support Vector Machines*), 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, bardzo przydatne w wizyjnych systemach wspomagania kierowcy (rozwiązanie takie dostępne jest już w niektórych samochodach).

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. Warto jednak zaznaczyć, że konieczność użycia zewnętrznej pamięci RAM np. przy odejmowaniu dwóch kolejnych ramek (detekcja ruchu) lub indeksacji dwuprzebiegowej nie jest czynnikiem uniemożliwiającym potokową realizację tych operacji. Dodatkowo, zawsze warto pamiętać o możliwości realizacji "niewygodnych" etapów w ramach architektury opartej o soft-procesor Microblaze lub procesor ARM (dla układów typu SoC ang. *System-on-a-chip* – Zynq).

8.2 Typowy cyfrowy interfejs wizyjny

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



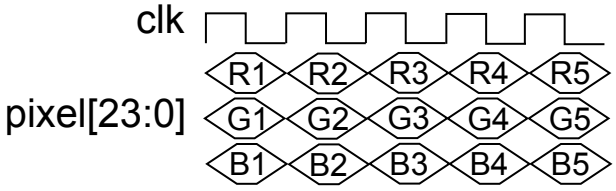
Rysunek 8.2: Schemat typowego modułu do przetwarzania obrazów

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 wektora cech.

Opis sygnałów:

- *clk* – zegar. W tym przypadku jest to tzw. zegar piksela, czyli zegar, który taktuje pojawianie się kolejnych pikseli. Przebieg zegara, dla strumienia RGB pokazano na rysunku 8.3. Warto zaznaczyć, że transmisja cyfrowego sygnału wideo odbywa się właśnie w taki szeregowy sposób (tj. piksel po pikselu).
- *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).



Rysunek 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 od lewej do prawej (skanowanie poziome) oraz od góry do dołu (skanowanie pionowe). Sygnały synchronizacji sterowały tym procesem. Przerwa w wyświetlaniu 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.

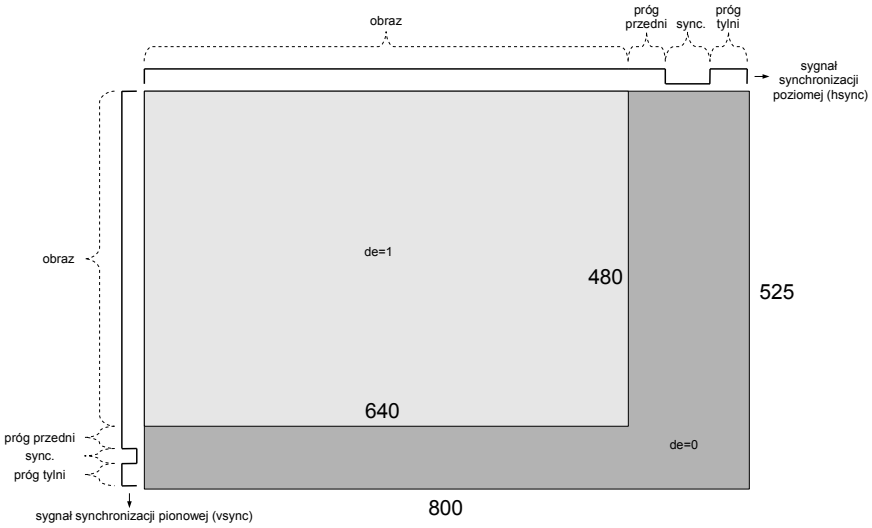
Na rysunku 8.4 pokazano sygnały synchronizacji dla obrazu o rozdzielczości 640×480.

Można zauważyć, że rzeczywista rozdzielczość obrazu jest większa od wyświetlanej (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



Rysunek 8.4: Synchronizacja dla obrazu o rozdzielczości 640×480

(ang. *Look-Up Table*) 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 rozważań związanych z operacjami kontekstowymi **??**.

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 (synchronizację monitora). 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 (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*.

Warto w tym miejscu jeszcze raz podkreślić, że sprawdzenie stworzonych modułów w symulacji jest **sprawę kluczową**. Szanse, że stworzymy poprawny moduł do przetwarzania obrazów

w pierwszej iteracji, jak pokazuje doświadczenie, są raczej niewielkie. Wykorzystanie symulacji pozawala nam zaoszczędzić **dużo czasu** i **łatwiej** wyeliminować wszystkie błędy.

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 (behawioralną) 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 rozdzielczości 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 związany z modułem źródłowym np. źx_"dla źródła sygnału. Praktyka ta pozwala uniknąć błędów w przypadku konieczności połączenia większej liczby modułów.

8.4 Uruchomienie toru wizyjnego na karcie Altys

Zadanie 8.2 Uruchom tor wizyjny na karcie Atlys. Potrzebne pliki zawarte są w archiwum *hdmi.zip*. Uwaga. W zależności od konkretnego stanowiska, konieczne jest zdefiniowanie lub zakomentowanie linii `` `define SPLITTER `` w pliku *hdmi_main.v*. W razie wątpliwości proszę zapytać Prowadzącego lub przeprowadzić dwa eksperymenty. W efekcie obraz z kamery wyświetlany będzie bezpośrednio na monitorze LCD.

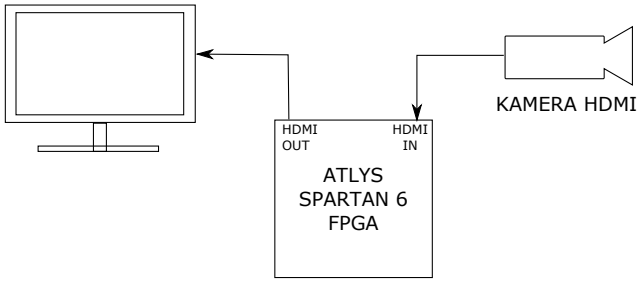
■

Sposób podłączenia karty Atlys przedstawiono na rysunku 8.4. Uwagi:

- na karcie Atlys używamy portów HDMI IN i HDMI OUT umieszczonych na **górnej krawędzi**,
- do HDMI IN podpinamy sygnał wizyjny z kamery,
- do HDMI OUT kabel łączący z monitorem LCD (proszę nie zamieniać kabli, bo nie będzie działać),
- 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.

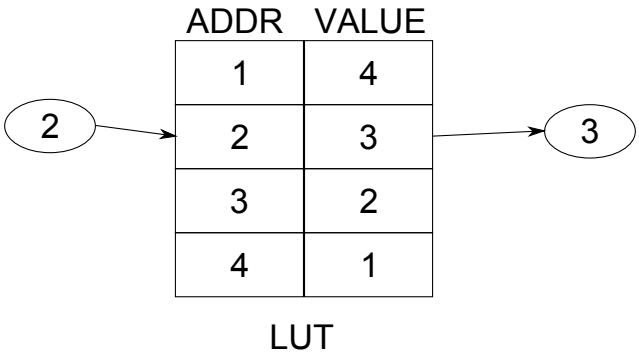
8.5 Realizacja operacji LUT

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.



Rysunek 8.5: Schemat podłączenia kamery i monitora do karty Atlys

Schematycznie zostało to przedstawione na rysunku 8.5. W tym przypadku piksel o wartości 2 zostaje zastąpiony pikselem o wartości 3.



Rysunek 8.6: Operacja LUT

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

Uwagi do realizacji:

- 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 konfiguratora 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 zakładce ustawiamy rejestrację wyników (*Output Options -> Registered*) oraz latencję (Pieline Stages) na 0 (taka się wyświetli).
- Na trzeciej zakładce należy załadować zawartość pamięci – podać odpowiedni plik **.coe*. W pliku tym zapisane są wartości, które zostaną załadowane do modułu ROM. Przykładowy plik zamieszczony jest w głównym folderze projektu. Podglądnij wczytaną zawartość. Wygeneruj moduł.
- Instancję LUT dodaj do modelu symulacyjnego toru wizyjnego i odpowiednio podłącz. De facto potrzebujemy trzech instancji – przetwarzamy każdą składową barwną niezależnie. Pamiętaj. Szablon instancji – Zakładka *Processes->CORE Generator->View HDL Instan-*

tiation Template. Nie zapomnij również utworzyć odpowiednich sygnałów wyjściowych. **Uwaga**. Przy przypisywaniu wartości wyjściowych (sekcja "*Output assigment*") warto raczej komentować poprzednie przypisania, niż je nadpisywać. W ten sposób stworzymy sobie "multiplekser" i łatwiej będzie nam wracać do poprzednich wersji.

- Poprawność metodologiczna wymaga aby opóźnić sygnały synchronizacji *de*, *hsync* i *vsync*. Wynika to z synchroniczności pamięci (opcja *Registered*). Najprościej do tego wykorzystać trzy rejestry pomocnicze i proces (instrukcja `always`).

Kod 8.5.1 — Przykład realizacji opóźnienia o jeden takt:

```
reg lut_de = 0;
reg lut_hsync = 0;
reg lut_vsync = 0;

always @(posedge rx_pclk)
begin
  lut_de <= rx_de;
  lut_hsync <= rx_hsync;
  lut_vsync <= rx_vsync;
end
```

- Sprawdź symulacyjnie poprawność rozwiązania. Uwaga. Wynikowy rysunek jest istotnie inny niż wejściowy. Sprawdź jaką funkcję LUT realizuje stworzony moduł.
- Dodaj (przekopiuj) moduły do toru wizyjnego przeznaczonego dla karty Altys i uruchom go. Nazewnictwo sygnałów jest takie samo w modelu symulacyjnym i pliku *hdmi_main*. Nie zapomnij wyjść modułów LUT połączyć z wyjściem w sekcji "*HDMI output port signal assigments*".

8.6 Zadania do wykonania w domu

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

Podpowiedzi:

- Klucz to postać pliku **.coe*. Należy wykonać analizę istniejącego lub przeglądnąć 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 Atyls.

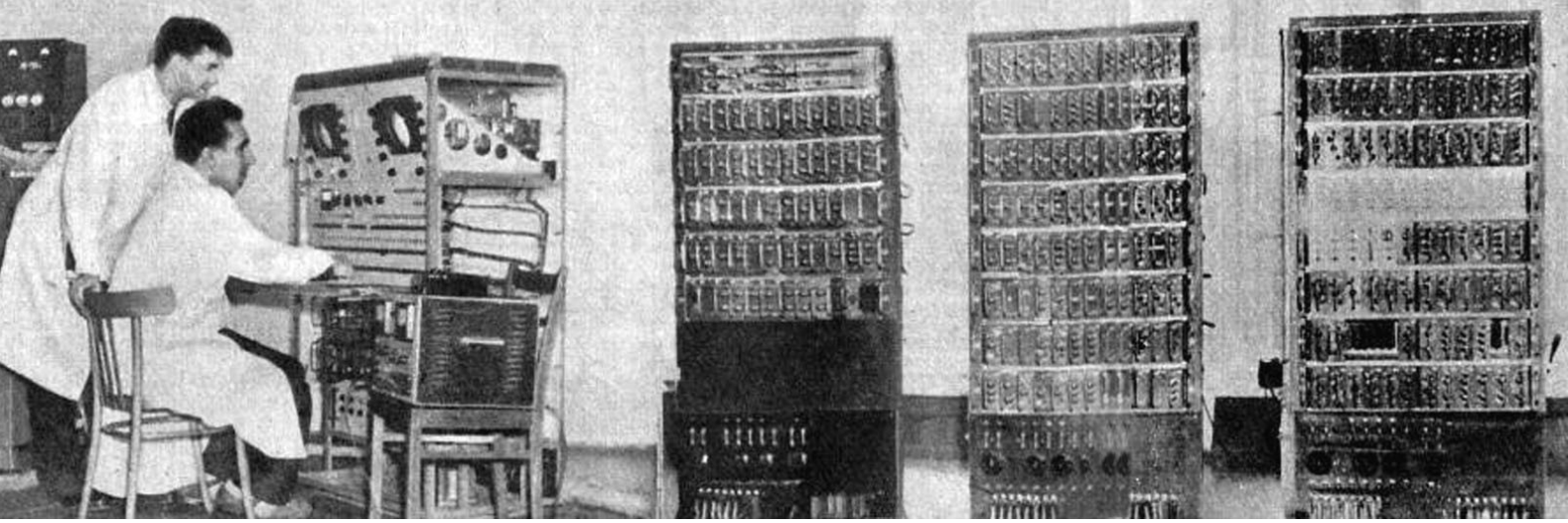
Zadanie 8.5 Wykonaj testy symulacyjne modułu LUT dla różnych modeli tj. behawioralnego, po fazach translacji, mapowania oraz łączenia i rozmieszczania. Zaobserwuj różnice. ■

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 *tb_mainLut*, 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 po czasie inicjalizacji 100 ns.
- uruchom symulację. Zwróć uwagę, aby na liście wyboru zaznaczona była pozycja *Behavioral*.
- 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 pojawieniem 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.
- **Uwaga**. Niekiedy (niestety nie udało się dokładnie zdiagnozować w jakich warunkach), symulacje inne niż behawioralna nie dają się uruchomić. Jeśli typowe podejścia takie jak: wyczyszczenie plików tymczasowych projektu i ponowne uruchomienie aplikacji ISE nie pomagają, to najprościej jest stworzyć projekt od nowa (5 min pracy).



Bibliografia

[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.