

Systemy Rekonfigurowalne

skrypt do ćwiczeń laboratoryjnych

ZYBO, Vivado 2017.4

Mateusz Komorkiewicz,
Tomasz Kryjak

Copyright © 2014-2018
Mateusz Komorkiewicz,
Tomasz Kryjak

PUBLISHED BY AGH

First printing, March 2014

Spis treści

1	Wprowadzenie	7
1.1	Słowo wstępne	7
1.2	Od lampy elektronowej do układu FPGA – rys historyczny	9
1.3	Budowa układów FPGA serii Artix 7 firmy Xilinx	9
1.3.1	Blok CLB	9
1.3.2	Pozostałe zasoby	10
2	Układy FPGA – pierwsze kroki	13
2.1	Wstęp	13
2.2	Język opisu sprzętu Verilog	13
2.3	Vivado – środowisko programistyczne	14
2.3.1	Vivado WebPACK	16
2.4	Zybo – platforma sprzętowa	16
2.4.1	Podłączanie i odłączanie kart Zybo	17
2.5	Zadania do wykonania na laboratorium	17
2.6	Zadania do wykonania w domu	20
2.7	Podsumowanie	20
3	Wstęp do projektowania struktury FPGA	21
3.1	Język Verilog – wprowadzenie	21
3.1.1	Moduł	21
3.1.2	Opis połączeń	22
3.1.3	Zapis liczby	23
3.1.4	Łączenie modułów	24
3.1.5	Opis struktury a opis zachowania	25

3.1.6 Bramka AND	26
3.1.7 Bramka OR	26
3.1.8 Bramka NOT	26
3.1.9 Dekoder	26
3.1.10 Koder	27
3.1.11 Demultiplexer	27
3.1.12 Mux	28
3.1.13 Rejestr	28
3.1.14 Licznik	29
3.1.15 Instrukcja generate	30
3.1.16 Maszyna stanów	31
4 Weryfikacja i testowanie projektu	33
4.1 Język Verilog – konstrukcje symulacyjne	34
4.1.1 Środowisko testowe	35
4.1.2 Generacja sekwencji testowych	35
4.1.3 Weryfikacja uzyskanych wyników	37
4.2 Model programowy	38
4.2.1 Dostęp do plików na dysku komputera	40
5 Verilog i weryfikacja – praktyka	41
5.1 Zadania do realizacji na zajęciach	41
5.1.1 Kaskada bramek AND	41
5.1.2 Licznik dzielący modulo N	43
5.1.3 Złożony moduł logiczny	44
5.2 Zadania do wykonania w domu	44
5.2.1 Linia opóźniająca	44
5.2.2 Tajemniczy moduł	45
6 Maszyny stanowe i zaawansowane testowanie	47
6.1 Zadania do realizacji na laboratorium	47
6.2 Zadania do realizacji w domu	49
6.3 Zadania dodatkowe	49
7 Operacje arytmetyczne	51
7.1 Format zapisu liczb	51
7.1.1 Całkowitoliczbowy bez znaku	51
7.1.2 Całkowitoliczbowy ze znakiem	52
7.1.3 Stałoprzecinkowy bez znaku	53
7.1.4 Stałoprzecinkowy ze znakiem	54
7.2 Zmienna długość słowa	55
7.3 Latencja	55
7.4 Pisanie a generowanie	58

7.5	Pierwiastkowanie, funkcje trygonometryczne, logarytmy	59
7.5.1	Tablicowanie wartości funkcji	59
7.6	Zadania do wykonania na laboratorium	60
7.7	Zadania do wykonania w domu	64
7.8	Zadania dodatkowe	66
8	Potokowe przetwarzanie i analiza obrazów	67
8.1	Wstęp teoretyczny	67
8.2	Typowy cyfrowy interfejs wizyjny	68
8.3	Model programowy przetwarzania obrazów	70
8.4	Uruchomienie toru wizyjnego na karcie Zybo	71
8.5	Realizacja operacji LUT	74
8.6	Zadania do wykonania w domu	75
9	Segmentacja obszarów o kolorze skóry	77
9.1	Wprowadzenie	77
9.2	Konwersja RGB do YCbCr – podstawy	78
9.3	Binaryzacja	79
9.4	Filtracja	80
9.5	Wyznaczanie środka ciężkości	80
9.6	Przykład działania	81
9.7	Zadanie do wykonania w domu	81
9.7.1	Model programowy	81
10	Konwersja RGB do YCbCr	83
10.1	Model programowy	83
10.2	Implementacja sprzętowa	84
10.3	Uruchomienie na karcie Zybo	85
10.4	Implementacja progowania	87
10.5	Zadania dodatkowe	87
11	Środek ciężkości	89
11.1	Wyznaczanie środka ciężkości	89
11.2	Zadania do wykonania na laboratorium	91
11.3	Zadania do wykonania w domu	93
11.4	Zadania dodatkowe	93

12	Potokowa realizacja operacji kontekstowych	95
12.1	Koncepcja realizacji operacji kontekstowych w potokowym systemie wizjnym	95
12.2	Zadania do wykonania na laboratorium	96
12.3	Zadania do wykonania w domu	99
12.4	Zadania dodatkowe	99
13	Procesor w układzie FPGA	101
13.1	Czym jest procesor	101
13.2	Instrukcje, asembler, kod maszynowy	102
13.3	Opis proponowanego procesora	102
13.4	Przykład realizacji instrukcji <code>mov</code> i <code>movi</code>	103
13.5	Zadania do wykonania na laboratorium	106
13.6	Zadania do wykonania w domu	107
14	Procesor - testowanie i weryfikacja	109
14.1	Realizacja portów I/O	109
14.2	Weryfikacja sprzętowa	110
14.3	Zadania do wykonania na laboratorium	111
14.4	Zadania dodatkowe	112

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. Głównie z uwagi na efektywność energetyczną. Zasadniczo obserwuje się dwie drogi akceleracji. Po pierwsze, zrównoleglenie w architekturach homogenicznych (tj. jednorodnych). 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*). Druga opcja to również zrównoleglenie, ale w architekturach heterogenicznych (tj. niejednorodnych). W tym rozwiązaniu obok „tradycyjnych” rdzeni procesora występują bardziej wyspecjalizowane zasoby obliczeniowe (programowalne lub nie). Przykładem są procesory serii A firmy AMD (połączenie GPP i GPU) i różnego rodzaju układy SoC (ang. *System on Chip*) np. Zynq firmy Xilinx (ARM + FPGA) oraz seria TDAX firmy Texas Instruments dedykowana dla systemów ADAS (ang. *Advanced Driver Assistance Systems* – zaawansowane systemy wspomagania kierowcy), gdzie w ramach jednego układu umieszczony bloki DSP, procesor ogólnego przeznaczenia ARM M4, akcelerator do operacji przetwarzania obrazów oraz interfejsy wideo.

W dzisiejszych czasach programowanie w klasycznych językach C/C++/C#/Java/Python stało się wiedzą powszechną (w wielu przypadkach niezbędną) i jest nauczane na wielu różnych szczeblach i kierunkach edukacji. Podejście sekwencyjne 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. W tym miejscu warto zauważyć paradoks – nasz mózg jest strukturą super-równoległą, ale pracować woli sekwencyjne (zwykle „jednowątkowo”).

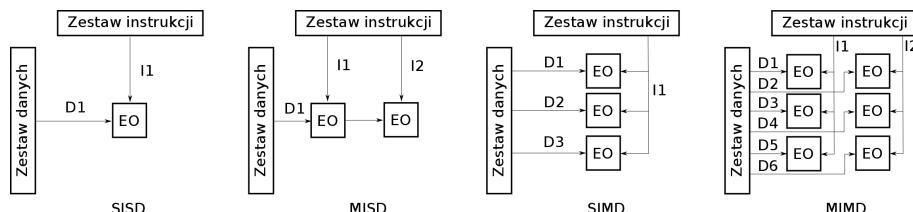
W programowaniu równoległym, na algorytm nie patrzy się jak na „sekwencję instrukcji”, ale na „zbiór elementów obliczeniowych” przez które „przepływa” strumień danych. Ponadto, aby wykorzystać możliwości jakie oferuje równoległość należy dość dobrze znać wykorzystywanyą platformę sprzętową (np. GPGPU, instrukcje wektorowe w CPU). Należy też rozwiązać szereg problemów związanych z synchronizacją różnych modułów obliczeniowych i sposobem dostępu do danych (ich współdzieleniem). W przypadku programowania na CPU, znajomość

sprzętu 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 Pythonie lub Javie zupełnie nie wiedząc jak działa system komputerowy, procesor, pamięć instrukcji i danych itp.

Programowanie równolegle 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 1966 r. (wciąż aktualną). Wyróżnia się w niej cztery klasy ze względu na liczbę równoległych strumieni danych i instrukcji (por. rysunek ??).

- 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).
- MISD (ang. *Multiple Instruction stream, Single Data stream*) – wiele strumieni instrukcji, pojedynczy strumień danych. W takim rozwiążaniu na tych samych danych wykonywany jest ciąg instrukcji. Ta nazwa można określić tzw. przetwarzanie potokowe, gdzie pojedynczy zestaw danych (np. obraz) poddawany jest różnym przekształceniom w sposób sekwencyjny (np. korekcji kolorów, binaryzacji, indeksacji).
- 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.
- MIMD (ang. *Multiple Instruction stream, Multiple Data stream*) – wiele strumieni instrukcji, wiele strumieni danych. W tym przypadku wiele niezależnych elementów obliczeniowych realizuje różne instrukcje na różnych danych. Jest to architektura typowa dla superkomputerów.



Rysunek 1.1: Taksonomia Flynna. EO – Element Obliczeniowy.

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

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

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 ścisłe określona. 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 *strictę* praktycznych umiejętności projektowania w języku opisu sprzętu Verilog (na zajęciach wykorzystany zostanie tylko pewien niewielki podzbior możliwości tego języka).

1.2 Od lampy elektronowej do układu FPGA – rys historyczny

Zasada działania większości maszyn cyfrowych jest podobna. Wykorzystują one odpowiednio połączone podstawowe elementy takie jak bramki logiczne (element realizujący obliczenia) i rejesty (element pamiętający) w celu realizacji bardziej złożonych funkcji. W pierwszych komputerach wykorzystywano przekaźniki i lampy elektronowe. Wraz z rozwojem elektroniki zastosowano tranzystory, a później układy scalone. Te ostatnie uległy miniaturyzacji – od dużych obudów, które zawierały kilka bramek logicznych, po nowoczesne procesory wykorzystywane w aplikacjach 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 sprzętowych. W latach 80 powstał problem opisu struktury i zachowania układów scalonych, w szczególności dokumentowania ich działania. Ciągła miniaturyzacja wymuszała tworzenie coraz bardziej skomplikowanych struktur, natomiast wykorzystywane metody projektowania układów przy pomocy schematu ideowego (graficznego) nie pozwalały na osiągnięcie tego celu w prosty sposób (inaczej mówiąc metody „zle się skalowały”). 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.

W dzisiejszych czasach również możemy zaobserwować pewne „przesilenie” związane ze sposobem projektowania logiki FPGA. Nie da się ukryć, że używanie klasycznych języków

opisu sprzętu wymaga dużych kompetencji z dziedziny elektroniki i jest zarezerwowane dla stosunkowo wąskiego grona specjalistów. Stanowi to niewątpliwą barierę przed upowszechnieniem technologii. Dlatego też od lat trwały pracy nad tzw. narzędziami syntezy wysokiego poziomu – HLS (ang. *High Level Synthesis*). Idea jest następująca. Mając na wejściu kod w C/C++ (lub dowolnym innym „popularnym” języku), przy minimalnej ingerencji (np. tylko poprzez dodanie kilku pragm), uzyskać w sposób automatyczny kod VHDL lub Verilog. Na przestrzeni ostatnich kilkunastu lat propozycji było dużo – zarówno komercyjnych, jak i akademickich, ale dopiero wsparcie „dużych graczy” na rynku FPGA takich jak Xilinx (Vivado HLS) oraz Intel (dawniej Altera, Intel HLS) spowodowało, że narzędzia te stały się realną alternatywą wobec projektowania niskopoziomowego. Istnieją również narzędzia „systemowe”. Umożliwiają one przygotowanie konfiguracji FPGA i programu procesora dla układu heterogenicznego za pomocą jednego języka i w obrębie jednego środowiska. Wspomagają one również wybór najbardziej złożonych obliczeniowo operacji, które powinny zostać przyspieszone w logice reprogramowalnej. Przykładem takiego narzędzia jest SDSoC firmy Xilinx. Zagadnienia te zostaną obszernie przedstawione w ramach innych kursów w dalszym toku studiów („Narzędzia HLS”).

1.3 Budowa układów FPGA serii Artix 7 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 Artix-7 firmy Xilinx. W tym miejscu należy wyjaśnić, że w trakcie kursu wykorzystywana będzie karta Zybo firmy Digilent wyposażona w układ Zynq SoC (ang. *System on Chip*). Układ Zynq określa się mianem „heterogeniczny”, gdyż w jednej obudowie zawiera dwa rodzaje zasobów sprzętowych: logikę reprogramowalną (układ FPGA) oraz system procesorowy (dwurdzeniowy procesor ARM Cortex-A9). W ramach niniejszych zajęć będzie wykorzystywana tylko część reprogramowalna, która dla układu na karcie Zybo oparta jest o logikę serii Artix-7 firmy Xilinx. Wykorzystanie możliwości tworzenia systemów sprzętowo-programowych z wykorzystaniem układów Zynq jest przedmiotem dalszego kursu pt. „Programowo-Sprzętowa Realizacja Algorytmów”. Ogólny opis rodziny Zynq można odnaleźć w dokumencie [3].

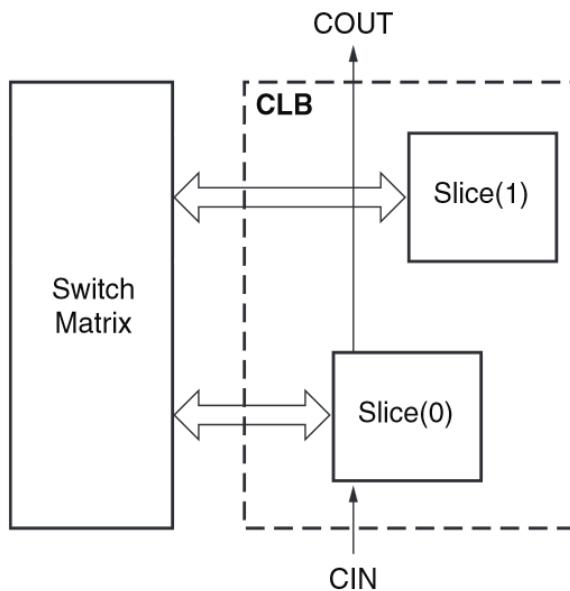
1.3.1 Blok CLB

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 przełączników (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 Zynq występują dwa rodzaje slice’ów: SLICEL, SLICEM. Schemat budowy pojedynczego *slice’u* typu M zamieszczono na rysunku 1.3.1.

Złożony on jest z następujących elementów:

- generator funkcyjny (4 sztuki) — został zrealizowany jako LUT (ang. *Look-Up Table*) posiadający 6 wejść i dwa niezależne wyjścia. Zatem możliwe jest zaimplementowanie: 6-wejściowej funkcji logicznej, dwóch 5-wejściowych funkcji logicznych (ze wspólnym wejściem), dwóch funkcji logicznych z 3 i 2 wejściami. Ponadto multipleksery umożliwiają tworzenie funkcji 7- i 8-wejściowych poprzez łączenie elementów LUT. Generator funkcyjny może zostać też skonfigurowany jako (tylko SLICEM):
 - synchroniczna pamięć RAM, zwana pamięcią rozproszoną (ang. *Distributed RAM*) — o różnym rozmiarze i liczbie portów (256×1 jednoportowa, 128×1 dwuportowa i 64×1 czteroportowa), przy czym jeden port umożliwia synchroniczny zapis



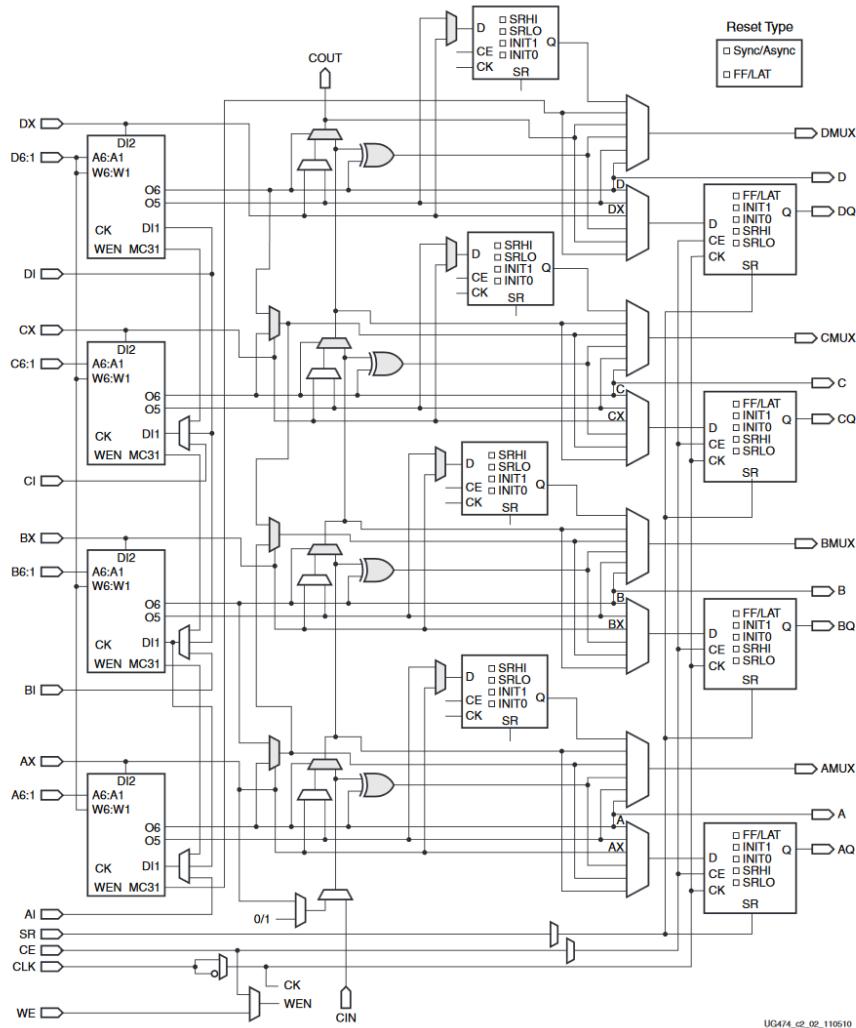
Rysunek 1.2: Schemat budowy bloku CLB. Źródło: [6]

- 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,
 - szybka logika przeniesienia — wykorzystywana przy realizacji operacji arytmetycznych.
- Bardziej szczegółowe informacje zwarte są w dokumencie [6] dostępnym na stronie www.xilinx.com.

1.3.2 Pozostałe zasoby

Wśród pozostałych zasobów dostępnych w układzie FPGA serii 7 warto wymienić:

- CMT (ang. *Clock Management 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 8 tego typu modułów. Szczegółowe informacje dostępne w dokumencie [5],
- Block RAM (BRAM) — bloki dedykowanej dwuportowej pamięci RAM o rozmiarze 36 Kb, które mogą zostać również skonfigurowane jako moduły FIFO. Dostępny rozmiar pamięci w zakresie od 1,8 Mb do 26,5 Mb. Szczegółowe informacje dostępne w dokumencie [10],
- DSP48A1 — moduły z mnożarką 25×18 bitów oraz 48-bitowym akumulatorem. Ich liczba waha się od 66 do 2020 w zależności od rozmiaru układu. Szczegółowe informacje dostępne w dokumencie [7],
- Select I/O — zasoby wejścia/wyjścia, podzielone na banki (liczba banków zależy od typu, rozmiaru i obudowy układu i waha się od 100 do 400 końcówek – podłączonych do części FPGA). Mogą zostać skonfigurowane do pracy z wieloma standardami (pojedynczymi i różnicowymi): LVCMOS, LVTTL, HSTL, PCI, SSTL, LVDS i innym. Szczegółowe informacje dostępne w dokumencie [4],
- GTP/GTX Transceivers — moduły nadawczo-odbiorcze umożliwiające szybką transmisję szeregową z prędkością do 12,5 Gb/s (GTX) i 6,25 (GTP). Wykorzystywane w interfejsach: Serial ATA, Aurora, 1G Ethernet, PCI Express i innych. Ich liczba waha się od 0 do 16.



Rysunek 1.3: Schemat budowy slice'u typu M. Z lewej cztery elementy LUT, pośrodku szybka logika przeniesienia, po prawej przerzutniki (4+4) oraz multipleksery. Źródło: [6]

Nie występują we wszystkich układach z rodziną Zynq. Szczegółowe informacje dostępne w dokumentach [9] oraz [8],

- Zintegrowany moduł PCI Express — wspiera transmisję z przepustowością 2,5 Gb/s w standardzie PCI Express Gen 1 oraz 5 Gb/s w standardzie Gen 2. Nie występuje we wszystkich układach z rodziną Zynq.
- XADC – konwerter analogowo cyfrowy. Moduł zawiera dwa 12-bitowe moduły o przepustowości 1 MSPS (ang. *Mega Sample Per Second*). Wspiera do 17 kanałów. Zawiera czujnik temperatury oraz zasilania. Szczegółowe informacje dostępne są w dokumencie [11]

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 opiszymy tworzoną architekturę,
- środowisko programistyczne i „kompilator”,
- platforma sprzętowa – karta z układem FPGA.

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

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

2.2 Język opisu sprzętu Verilog

Na laboratoriach będziemy wykorzystywać język opisu sprzętu Verilog, który został zaproponowany na przełomie roku 1983/1984 przez firmę Gateway Design Automation Inc. Od tego czasu przeszedł wiele modyfikacji, został upublicznyony na zasadzie otwartego standardu i dokonano jego oficjalnej standaryzacji jako norma IEEE 1364 (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). Pomyśl, aby „przetwarzać” kod na logikę (co określa się mianem syntezы) pojawił się dopiero później. Stąd w językach tych występuje szereg instrukcji, których nie da się zrealizować w FPGA, a są niezbędne przy symulacji np. obsługa 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 (nieco łatwiej o błąd). Od strony „możliwości” oba języki są jednak bardzo 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 Vivado – środowisko programistyczne

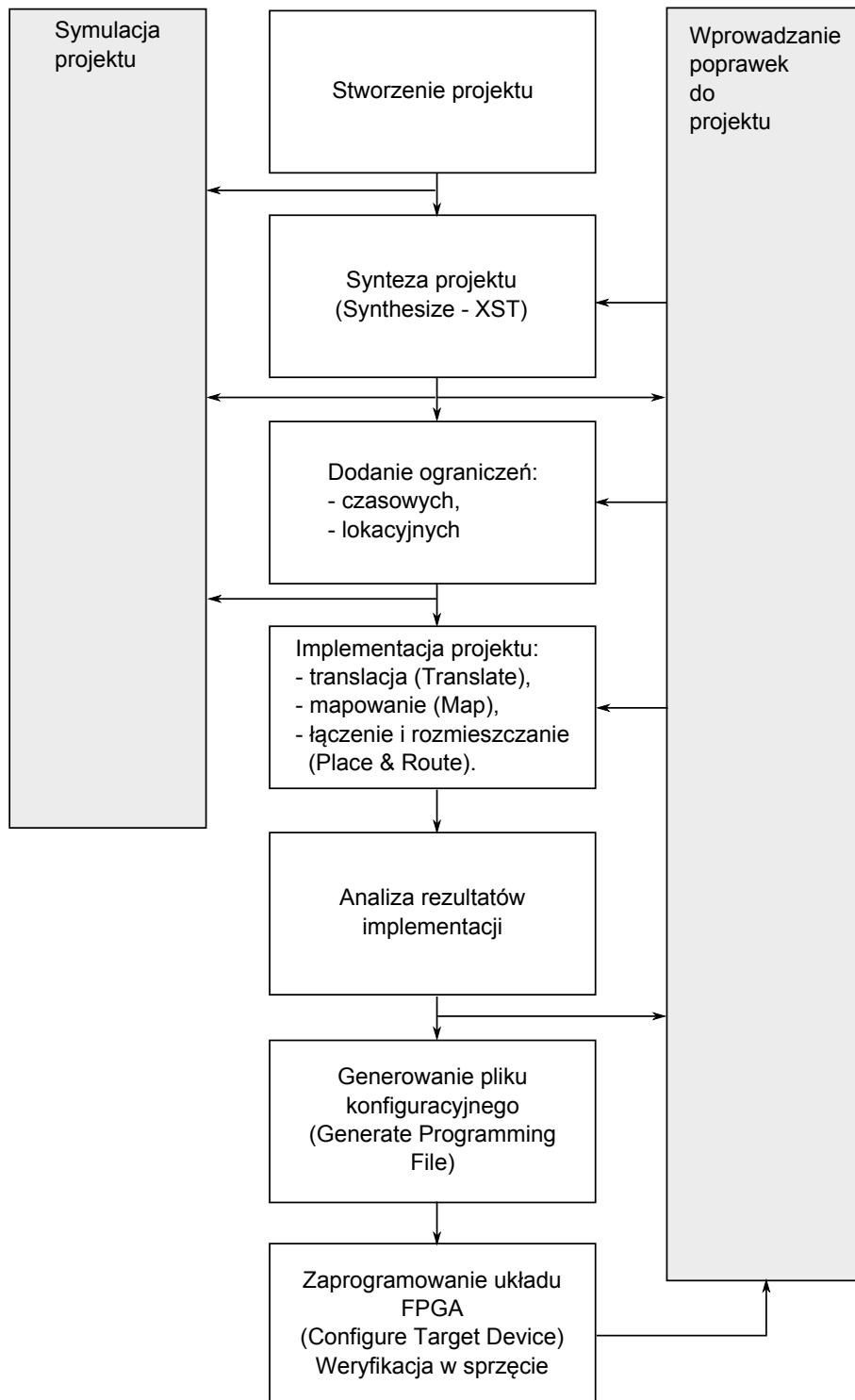
Na laboratoriach będziemy korzystać w układów FPGA serii 7 firmy Xilinx. Dedykowane dla nich środowisko programistyczne to Vivado. W pracach używać będziemy wersji 2017.4. Środowisko to nie różni się znacząco od typowych IDE np. Visual Studio lub Eclipse (choć w pewnych aspektach jest może nieco „toporne”).

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 Vivado (ustawienie parametrów), jak i opisanie wykorzystywanej logiki (VHDL, Verilog),
- synteza (*Synthesize-XST*) — na wejściu dostępne są pliki 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 i uruchomieniu jej na docelowej platformie sprzętowej i poddaniu procedurze ewaluacji.

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 Integrated Logic Analyzer (ILA) omówione poniżej). Dużo lepszym i szybszym rozwiązaniem jest symulacja, gdzie praktycznie uzyskujemy kompletną информацию o zachowaniu się modułu. Możemy ją wykonać na różnym etapie projektu



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

(behavioralnym, *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 narzędzia Inte-

grated Logic Analyzer. 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 ILA ma ograniczone możliwości analizy dużej liczby danych, przykładowo strumienia wideo (bufory, do których zapisywane są dane są stosunkowo niewielkie np. 2048 próbek).

Z powyższego opisu wyraźnie wynikają różnice, w tworzeniu projektu na CPU i FPGA. Na CPU mamy do dyspozycji „sztywną” architekturę (która 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 zobaczymy w trakcie kursu istota projektowania logiki polega na wykonywaniu elementów obliczeniowych oraz ustalaniu przepływu danych pomiędzy nimi.

2.3.1 Vivado WebPACK

Środowisko Vivado występuje w trzech wersjach.

- WebPACK – darmowej,
- Design Edition (bez narzędzia *System Generator for DSP*),
- System Edition (wszystkie elementy),
- Lab Edition – tylko programowania i analiza logiki,
- 30-dniowej ewaluacyjnej.

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. Vivado WebPACK. Z punktu widzenia funkcjonalności nie różni się ona od wersji System Edition. Ograniczono tylko możliwe do wyboru układy FPGA/Zynq (do tych mniejszych). Układ Zynq dostępny na płycie używanej na laboratorium tj. Zybo jest wspierany przez wersję WebPACK.

Instalacja jest dość prosta. Na stronie: <https://www.xilinx.com/support/download.html> należy wybrać odpowiednią wersję Vivado i plik instalacyjny. Wybór systemu operacyjnego – *de gustibus* (choć z uwagi na kompatybilność z projektami w pracowni zalecany jest Linux (Debian/Ubuntu))¹. Niestety należy się zarejestrować na stronie Xilinx'a. Po rejestracji i logowaniu uzyskujemy dostęp do ściągania. Można ściągnąć via downloader lub bezpośrednio (na stronie Downloads jest opis jak). Po instalacji należy wybrać licencję WebPACK i uzyskać ją na stronie www (wykorzystuje się to samo konto). Należy wziąć pod uwagę, że instalator to ok. 20 GB, a aplikacja po instalacji może zajmować ok. 40 GB. Druga sprawa do „apetyt” oprogramowania na moc obliczeniową i pamięć. Należy sobie otwarcie powiedzieć, że na typowym laptopie Vivado będzie działać wolno. W szczególności proszę nawet nie rozważać takich wariantów jak maszyna wirtualna, czy instalacja na napędzie zewnętrznym (nawet USB 3.0).

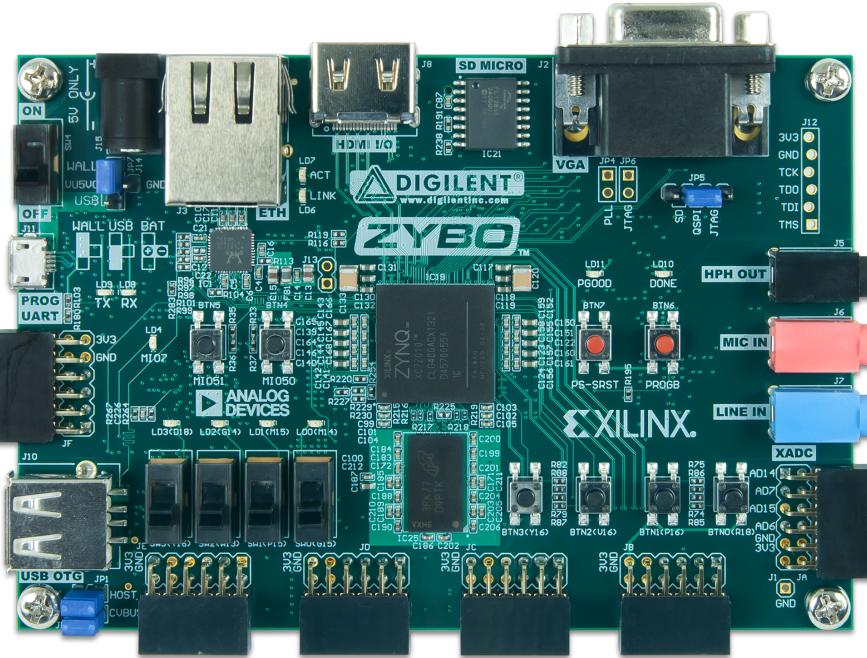
2.4 Zybo – platforma sprzętowa

Zdjęcie używanej na zajęciach platformy sprzętowej przedstawiono na rysunku 2.4.

Jej podstawowym elementem jest układ Zynq SoC w wersji XC7Z010 firmy Xilinx. Plasuje się on “w dole” rodziny Zynq. Ponadto na płycie umieszczono:

- 512 MB pamięci RAM DDR3,

¹W razie problemów/pytań co do instalacji na tych systemach, proszę skontaktować się z prowadzącym zajęcia. Użytkownicy systemu Windows, niestety, są pozostawieni „samii sobie” – ale zasadniczo oprogramowanie działa, może nawet bardziej stabilnie. Problemy są tylko z ew. przenoszeniem.



Rysunek 2.2: Karta uruchomieniowa Zybo firmy Digilent z układem Zybo SoC firmy Xilinx

- kontroler Ethernet (10/100/1000),
- port USB 2.0 (programowanie, transfer danych),
- port HDMI (wejście lub wyjście),
- wyjściowy port VGA,
- 128 Mb pamięci SPI Flash do przechowywania konfiguracji i danych,
- sześć portów Pmod do podłączania peryferiów,
- kodek audio z wyjściem słuchawkowym oraz wejściem z mikrofonu i liniowym,
- 5 diod LED, 6 przycisków oraz 4 przełączniki.

Uwaga. Od drugiej połowy dostępne w sprzedaży są nowe karty Zybo: Z7-10 i Z7-20. Pierwsza z nich zawiera ten sam układ, co „stare” Zybo, więcej RAM (1 GB), wejście i wyjście HDMI oraz złącze Pcam ze wsparciem MIPS CSI-2 (ang. *Camera Serial Interface*). Druga różni się od Z7-10 większym układem Zynq tj. XC7Z020 (ma on ponad dwukrotnie więcej zasobów logicznych).

2.4.1 Podłączanie i odłączanie kart Zybo

Podłączanie:

- wyciągamy kartę z pudełka,
- podpinamy kabel USB (służy zarówno do programowania jak i do komunikacji szeregowej(UART)). Ponadto stanowi również zasilanie (pozwala na to odpowiednie ustawienie zwojki obokłącznika),
- przełączamy wyłącznik na płytce.

Odłączanie:

- przełączamy wyłącznik na płytce,
- odpinamy kabel (**uwaga** – proszę to robić bardzo ostrożnie, bo można zniszczyć port),
- chowamy do pudełka kartę oraz kabel 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 Vivado, które będą niezbędne do wykonania pozostałych ćwiczeń.

Wykonanie:

1. otwórz program **Vivado 2014.4** (ikona na pulpicie),
2. utwórz nowy projekt — **File->New Project** oraz **Next**,
3. w oknie dialogowym ustal nazwę **Project name** (np. intro) oraz folder **Project location** („swój” folder),
4. wybierz typ projektu jako **RTL Project**,
5. pliki do projektu dodamy później (**Next**),
6. modułów IP nie będziemy dodawać (**Next**),
7. ograniczenia dodamy później (**Next**),
8. na zakładce **Default Part** wybierz **Boards** i z listy Zybo (uwaga nie Z7-10/Z7-20).
9. zakończ tworzenie projektu poprzez **Finish**,
10. utwórz nowy plik (moduł) **File->Add Sources** lub **Project Manager->Add Sources**, wybierz typ **Add or create design sources**, stwórz plik (**Create File**), nazwij plik *led_button*, naciśnij **Finish**,
11. ustal interfejs modułu (tj. jego sposób komunikacji ze światem zewnętrznym):
 - *sw* — input — sygnał z przełączników (4 bity). Na płycie są 4 przełączniki dwupołożeniowe (należy zaznaczyć opcję “Bus” i w MSB wpisać 3, a LSB 0).
 - *led* — output — sygnał do diod (4 bity). Na płycie są 4 diody, który wykorzystamy (należy zaznaczyć opcję “Bus” i w MSB wpisać 3, a LSB 0).
 - zakończ kreator **OK**. 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).
12. stworzony moduł pojawi się w hierarchii projektu (**Design Sources**).
13. poprzez dwukrotne kliknięcie należy go otworzyć. Proszę zwrócić uwagę na postać modułu (słowa kluczowe *module* i *endmodule* oraz wejście i wyjście).
14. 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”). Kod wpisujemy pomiędzy zakończenie deklaracji interfejsu, a *endmodule*.
15. mając utworzony i skończony moduł omówimy środowisko Vivado.

Po lewej stronie ekranu (rozmieszczenie domyślne – można je przywrócić opcją *Layout->Default Layout*) znajduje się okno **Flow Navigator**. Jest ono podzielone na kilka sekcji.

- Project Manager
 - Settings – otwiera okno z ustawieniami projektu oraz narzędzi (opcji jest bardzo dużo i raczej nie będziemy z nich korzystać),
 - Add Sources – kreator dodawania nowych źródeł do projektu,
 - Language Templates – pozwala otworzyć okno, w którym dostępnych jest szereg szablonów w języku Verilog i VHDL np. wzór instancji przerzutnika czy modułu pamięci BRAM.
 - IP Catalog – otwiera zakładkę biblioteką modułów IP (ang. *Intellectual Property*). W „branży” układów programowalnych tym mianem określa się kon-

figurowalne moduły sprzętowe np. sumator, mnożarkę, czy kontroler pamięci RAM.

- IP Integrator – obsługa schematów (tworzenie, otwieranie, generowanie).
- Simulation – symulacja
 - Run Simulation – pozwala uruchomić symulację.
- RTL Analysis – pierwszy (wstępny) etap analizy projektu
 - Open Elaborated Design
 - * Report Methodology –
 - * Report DRC – analiza zasad projektowania (DRC – ang. *Design Rule Check*),
 - * Schematic – otwiera zakładkę ze schematem systemu.
- Synthesis – synteza projektu
 - Run Synthesis – uruchomienie syntezy projektu.
 - Open Synthesized Design
 - * Constrain Wizzard
 - * Edit Timing Constraints
 - * Set Up Debug
 - * Report Timing Summary
 - * Report Clock Networks
 - * Report Clock Interaction
 - * Report Methodology
 - * Report DRC
 - * Report Noise
 - * Report Utilization
 - * Report Power
 - * Schematic
- Implementation – implementacja projektu
 - Implementation Settings,
 - Run Implementation,
 - Open Implemented Design
 - * Constrain Wizzard
 - * Edit Timing Constraints
 - * Set Up Debug
 - * Report Timing Summary
 - * Report Clock Networks
 - * Report Clock Interaction
 - * Report Methodology
 - * Report DRC
 - * Report Noise
 - * Report Utilization
 - * Report Power
 - * Schematic
- Program and Debug – generowanie pliku konfiguracyjnego oraz ew. analiza działania
 - Generate Bitstream – generowanie pliku bit.
 - Open Hardware Manager
 - * Open Target – nawiązanie połączenia z układem,
 - * Program Device – zaprogramowania układu,
 - * Add Configuration Memory Device

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 **Flow Navigator** co chcemy z nim zrobić. **Zawsze warto sprawdzić co zaznaczyliśmy** – szczególnie przy uruchamianiu symulacji.

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 dokumentacji środowiska Vivado: <https://www.xilinx.com/products/design-tools/vivado.html#documentation>

16. dodaj do projektu plik **ZYBO_Master.xdc** (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).
17. **Add Sources-> Add or Create Constraints->Add File.** Proszę sprawdzić czy zaznaczona jest opcja *Copy constraints files into project*
18. dodany plik **xdc** pojawi się w hierarchii (**Constraints**). Otwórz go.
19. odszukaj sekcje odpowiedzialne za diody (**led**) i przełączniki (**sw**). Odkomentuj stosowne linijki.
20. projekt jest gotowy do syntezy i implementacji. Uruchom kolejne etapy. Uwaga. Można też od razu uruchomić generowanie pliku bit (**Generate Bitstream**). Wtedy wszystkie wymagane etapy uruchomią się automatycznie.
21. przejrzyj raport — **Design Sumary**. Zwróć szczególną uwagę na zużycie zasobów logicznych, a raczej jego brak (*Slice Registers* i *LUTs*) oraz *IOBs*,
22. skonfiguruj układ FPGA karty Zybo. Wybierz polecenie **Open Hardware Manager->Open Target->Auto Connect**. Otworzy się **Hardware Manager**. Upewnij się, że karta jest podłączona do komputera PC kablem USB oraz, że zworka obok włącznika jest ustawiona na USB (zasilanie z USB).
23. zaprogramuj układ. Możliwości są co najmniej dwie. Można na zielonym pasku w górnej części ekranu wybrać **Program device**, bądź w hierarchii sprzętu (okno **Hardware**) wybrać **xc7z010_1**, kliknąć prawym przyciskiem myszy i też wybrać **Program device**. Wybór pliku bit należy zatwierdzić przyciskiem **Program**.
24. po zaprogramowaniu powinna zapalić się zielona dioda (podpisana **Done**).
25. 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 Vivado w wersji WebPACK ze strony www.xilinx.com – opis w rozdziale 2.3.1.

Uwaga. Jeśli dla kogośściagnięcie 16 GB jest niedogodne to prowadzący zajęcie dysponuje plikiem dla wszystkich systemów operacyjnych.

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 Vivado w zakresie tworzenia nowego projektu, dodawania do niego plików oraz ich syntezy i implementacji do postaci plików konfiguracyjnych (tzw. bitów),
 - odpowiednio podłączyć kartę Zybo 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 serii 7 firmy Xilinx oraz karty ewaluacyjnej Zybo firmy Digilent.

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. Zynq-7000 All Programmable SoC Overview. 2011.
- [4] Xilinx Inc. 7 Series FPGAs SelectIO Resources User Guide. 2016.
- [5] Xilinx Inc. 7 Series FPGAs Clocking Resources User Guide. 2016.
- [6] Xilinx Inc. 7 Series FPGAs Configurable Logic Block - User Guide. 2016.
- [7] Xilinx Inc. 7 Series FPGAs DSP48E1 Slice User Guidee. 2016.
- [8] Xilinx Inc. 7 Series FPGAs GTP Transceivers User Guid. 2016.
- [9] Xilinx Inc. 7 Series FPGAs GTX/GTH Transceivers User Guid. 2016.
- [10] Xilinx Inc. 7 Series FPGAs Memory Resources User Guide. 2016.
- [11] Xilinx Inc. 7 Series FPGAs XADC Dual 12-Bit 1MSPS Analog-to-Digital Converter User Guide . 2016.
- [12] J. C. Russ. *Image Processing Handbook, Fourth Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2002.
- [13] R. Tadeusiewicz. *Komputerowa analiza i przetwarzanie obrazów*. Wydawnictwo Fundacji Postępu Telekomunikacji, 1997.
- [14] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.