

# 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

<b>1</b>	<b>Wprowadzenie</b>	<b>5</b>
1.1	Słowo wstępne	5
1.2	Od lampy elektronowej do układu FPGA – rys historyczny	7
1.3	Budowa układów FPGA serii Artix 7 firmy Xilinx	8
1.3.1	Blok CLB	8
1.3.2	Pozostałe zasoby	9
<b>2</b>	<b>Układy FPGA – pierwsze kroki</b>	<b>11</b>
2.1	Wstęp	11
2.2	Język opisu sprzętu Verilog	11
2.3	Vivado – środowisko programistyczne	12
2.3.1	Vivado WebPACK	14
2.4	Zybo – platforma sprzętowa	14
2.4.1	Podłączanie i odłączanie kart Zybo	15
2.5	Zadania do wykonania na laboratorium	16
2.6	Zadania do wykonania w domu	18
2.7	Podsumowanie	19
<b>3</b>	<b>Wstęp do projektowania struktury FPGA</b>	<b>21</b>
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	Multiplexer .....	28
3.1.13	Rejestr .....	28
3.1.14	Liczniak .....	29
3.1.15	Instrukcja generate .....	30
3.1.16	Maszyna stanów .....	31
<b>4</b>	<b>Weryfikacja i testowanie projektu .....</b>	<b>33</b>
<b>4.1</b>	<b>Język Verilog – konstrukcje symulacyjne</b>	<b>34</b>
4.1.1	Środowisko testowe .....	35
4.1.2	Generacja sekwencji testowych .....	35
4.1.3	Weryfikacja uzyskanych wyników .....	37
<b>4.2</b>	<b>Model programowy</b>	<b>38</b>
4.2.1	Dostęp do plików na dysku komputera .....	40
<b>5</b>	<b>Verilog i weryfikacja – praktyka .....</b>	<b>41</b>
<b>5.1</b>	<b>Zadania do realizacji na zajęciach</b>	<b>41</b>
5.1.1	Kaskada bramek AND .....	41
5.1.2	Liczniak dzielący modulo N .....	43
5.1.3	Złożony moduł logiczny .....	44
<b>5.2</b>	<b>Zadania do wykonania w domu</b>	<b>44</b>
5.2.1	Linia opóźniająca .....	44
5.2.2	Tajemniczy moduł .....	45
<b>6</b>	<b>Maszyny stanowe i zaawansowane testowanie .....</b>	<b>47</b>
<b>6.1</b>	<b>Zadania do realizacji na laboratorium</b>	<b>47</b>
<b>6.2</b>	<b>Zadania do realizacji w domu</b>	<b>49</b>
<b>6.3</b>	<b>Zadania dodatkowe</b>	<b>49</b>
<b>7</b>	<b>Operacje arytmetyczne .....</b>	<b>51</b>
<b>7.1</b>	<b>Format zapisu liczb</b>	<b>51</b>
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
<b>7.2</b>	<b>Zmienna długość słowa</b>	<b>55</b>
<b>7.3</b>	<b>Latencja</b>	<b>55</b>
<b>7.4</b>	<b>Pisanie a generowanie</b>	<b>58</b>

<b>7.5</b>	<b>Pierwiastkowanie, funkcje trygonometryczne, logarytmy</b>	<b>59</b>
7.5.1	Tablicowanie wartości funkcji . . . . .	59
<b>7.6</b>	<b>Zadania do wykonania na laboratorium</b>	<b>60</b>
<b>7.7</b>	<b>Zadania do wykonania w domu</b>	<b>64</b>
<b>7.8</b>	<b>Zadania dodatkowe</b>	<b>66</b>
<b>8</b>	<b>Potokowe przetwarzanie i analiza obrazów</b>	<b>67</b>
<b>8.1</b>	<b>Wstęp teoretyczny</b>	<b>67</b>
<b>8.2</b>	<b>Typowy cyfrowy interfejs wizyjny</b>	<b>68</b>
<b>8.3</b>	<b>Model programowy przetwarzania obrazów</b>	<b>70</b>
<b>8.4</b>	<b>Uruchomienie toru wizyjnego na karcie Zybo</b>	<b>71</b>
<b>8.5</b>	<b>Realizacja operacji LUT</b>	<b>73</b>
<b>8.6</b>	<b>Zadania do wykonania w domu</b>	<b>75</b>
<b>9</b>	<b>Segmentacja obszarów o kolorze skóry</b>	<b>77</b>
<b>9.1</b>	<b>Wprowadzenie</b>	<b>77</b>
<b>9.2</b>	<b>Konwersja RGB do YCbCr – podstawy</b>	<b>78</b>
<b>9.3</b>	<b>Binaryzacja</b>	<b>79</b>
<b>9.4</b>	<b>Filtracja</b>	<b>80</b>
<b>9.5</b>	<b>Wyznaczanie środka ciężkości</b>	<b>80</b>
<b>9.6</b>	<b>Przykład działania</b>	<b>81</b>
<b>9.7</b>	<b>Zadanie do wykonania w domu</b>	<b>81</b>
9.7.1	Model programowy . . . . .	81
<b>10</b>	<b>Konwersja RGB do YCbCr</b>	<b>83</b>
<b>10.1</b>	<b>Model programowy</b>	<b>83</b>
<b>10.2</b>	<b>Implementacja sprzętowa</b>	<b>84</b>
<b>10.3</b>	<b>Uruchomienie na karcie Zybo</b>	<b>85</b>
<b>10.4</b>	<b>Implementacja progowania</b>	<b>87</b>
<b>10.5</b>	<b>Zadania dodatkowe</b>	<b>87</b>
<b>11</b>	<b>Środek ciężkości</b>	<b>89</b>
<b>11.1</b>	<b>Wyznaczanie środka ciężkości</b>	<b>89</b>
<b>11.2</b>	<b>Zadania do wykonania na laboratorium</b>	<b>91</b>
<b>11.3</b>	<b>Zadania do wykonania w domu</b>	<b>93</b>
<b>11.4</b>	<b>Zadania dodatkowe</b>	<b>93</b>

<b>12</b>	<b>Potokowa realizacja operacji kontekstowych .....</b>	<b>95</b>
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
<b>13</b>	<b>Procesor w układzie FPGA .....</b>	<b>101</b>
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
<b>14</b>	<b>Procesor - testowanie i weryfikacja .....</b>	<b>109</b>
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”).

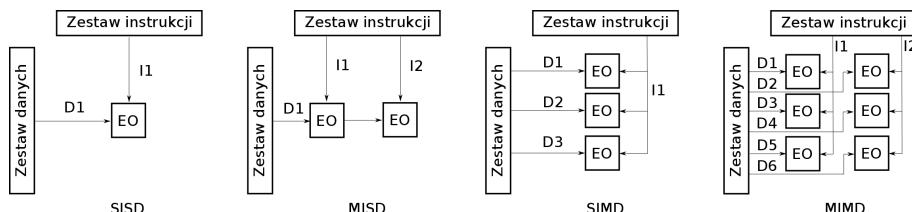
W programowaniu równoległym, na algorytm nie patrzy się jak na „sekwencję instrukcji”, ale na „zbiór elementów obliczeniowych” przez które „przepływa” strumień danych. Ponadto, aby wykorzystać możliwości jakie oferuje równoległość należy dość dobrze znać wykorzystywanyą platformę sprzętową (np. GPGPU, 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 1.1).

- 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 Flynn'a. 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”<sup>1</sup> układów FPGA różni się dość zasadniczo od

<sup>1</sup>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 „złe 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 [?].

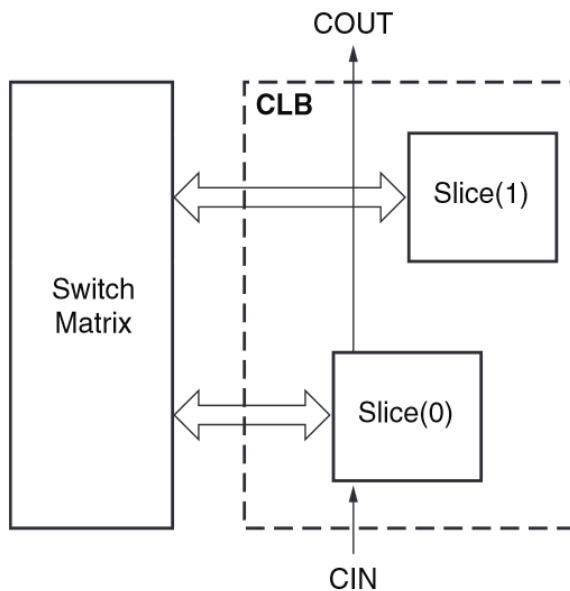
#### 1.3.1 Blok CLB

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 ( $256 \times 1$  jednoportowa,  $128 \times 1$  dwuportowa i  $64 \times 1$  czteroportowa), przy czym jeden port umożliwia synchroniczny zapis



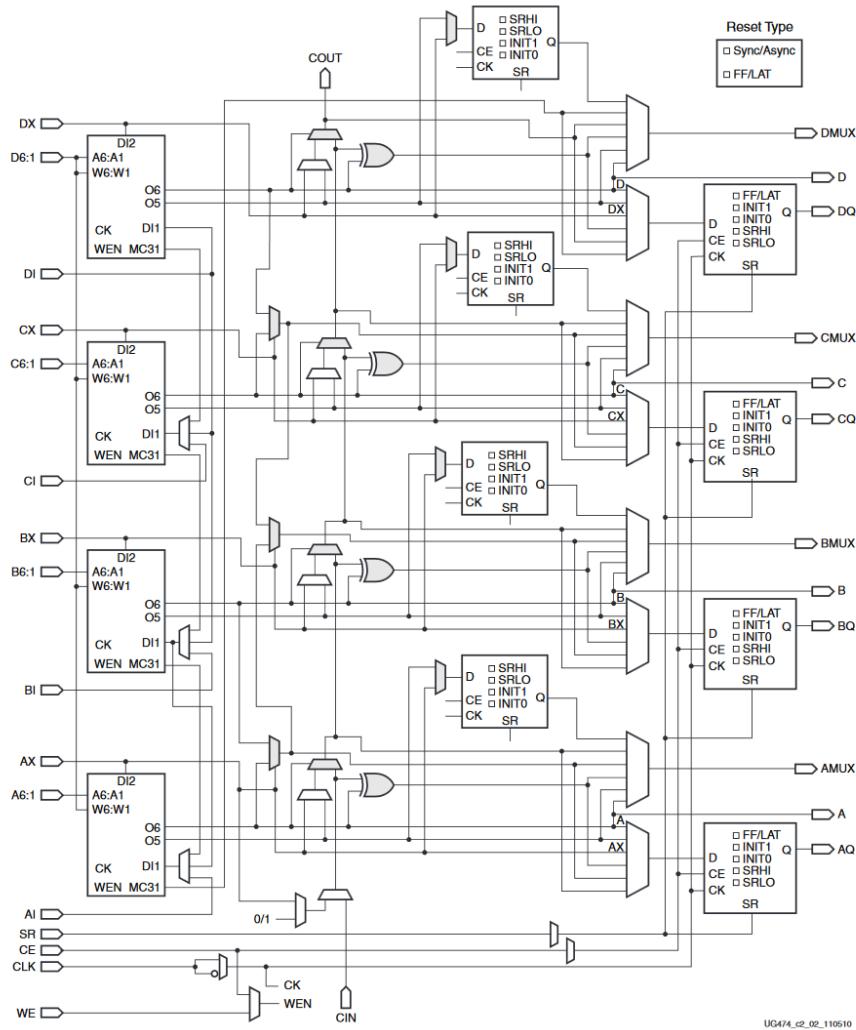
Rysunek 1.2: Schemat budowy bloku CLB. Źródło: [?]

- 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 [?] dostępnym na stronie [www.xilinx.com](http://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 [?],
- 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 [?],
- DSP48A1 — moduły z mnożarką  $25 \times 18$  bitów oraz 48-bitowym akumulatorem. Ich liczba waha się od 66 do 2020 w zależności od rozmiaru układu. Szczegółowe informacje dostępne w dokumencie [?],
- 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 [?],
- 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: [?]

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

- 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 [?].

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

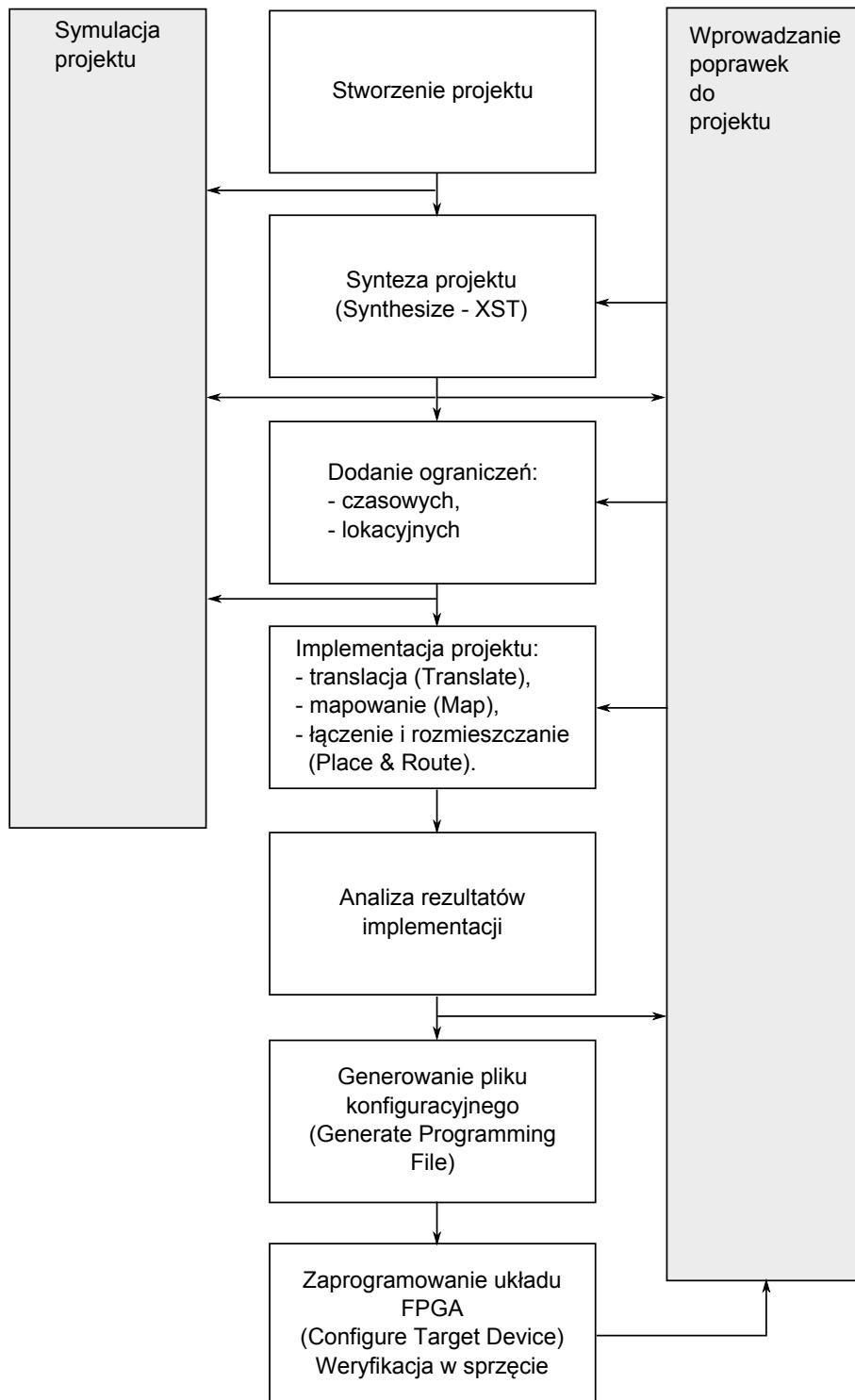
Na laboratoriach będziemy korzystać w układów FPGA serii 7 firmy Xilinx. Dedykowane dla nich środowisko programistyczne to Vivado. W pracach używać będziemy wersji 2017.4. Środowisko to nie różni się znacząco od typowych IDE np. Visual Studio lub Eclipse (choć w pewnych aspektach jest może nieco „toporne”).

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))<sup>1</sup>. 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).

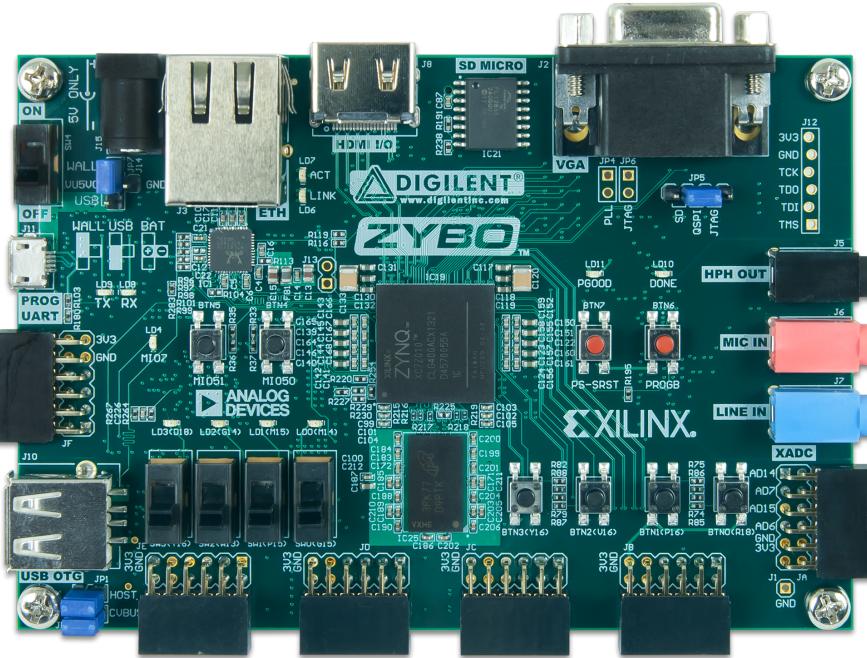
## 2.4 Zybo – platforma sprzętowa

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,

<sup>1</sup>W 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),
- przełączamywyłącznik na płytce.

Odłączanie:

- przełączamywyłącznik 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](http://www.xilinx.com) – opis w rozdziale 2.3.1. ■

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

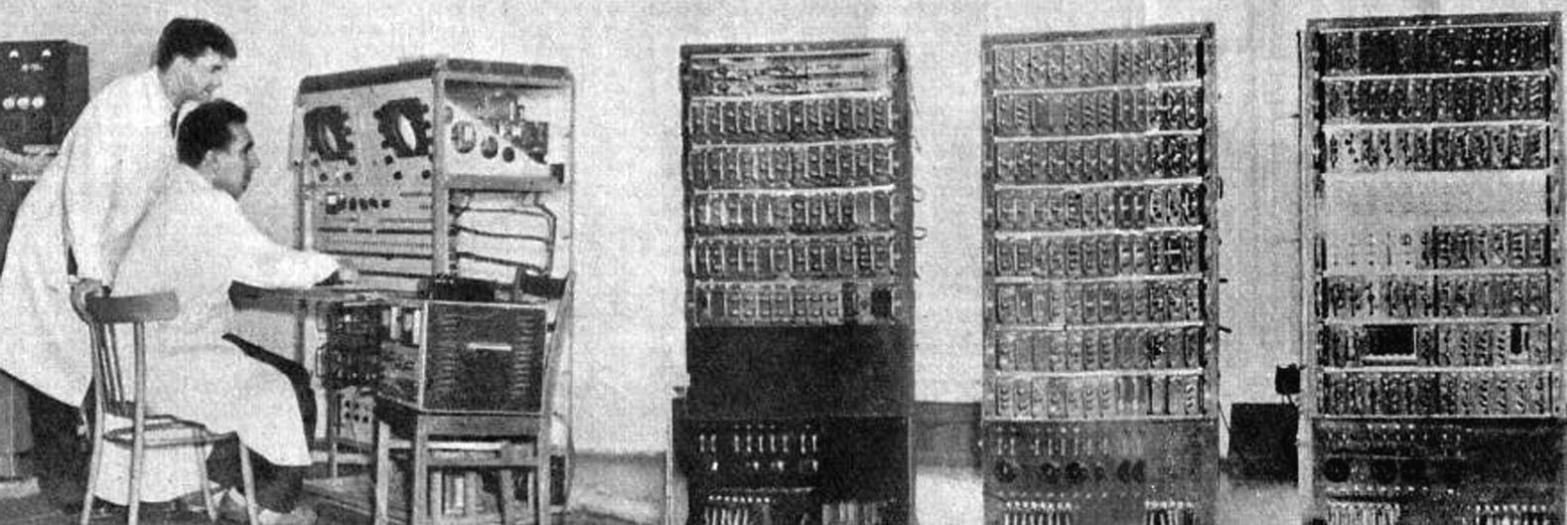
**Zadanie 2.3** Proszę zapoznać się z podstawowymi informacjami o budowie układów FPGA – rozdział 1 niniejszego skryptu. ■

## 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.





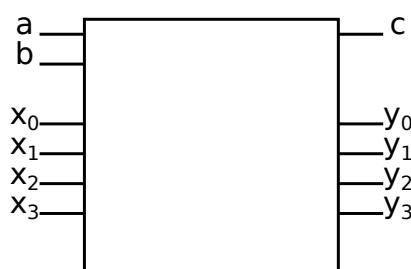
## 3 — Wstęp do projektowania struktury FPGA

### 3.1 Język Verilog – wprowadzenie

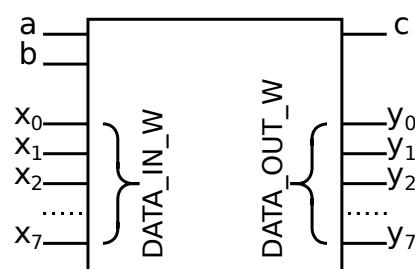
W niniejszym rozdziale zostaną przedstawione podstawowe elementy strukturalne występujące w języku Verilog, które umożliwiają projektowanie logiki w układach rekonfigurowalnych. Warto podkreślić, że jest to wariant „minumum”. Dalszych informacji należy szukać w szeroko rozumianym Internecie oraz w wielu 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 wewnętrznych 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 (sensowną) 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 języku Verilog jest definiowana przy pomocy wyrażenia **wire**.

**Kod 3.1.3 — Moduł z połączeniami:**

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

assign fixed1=8'hcc;

endmodule
```

**Kod 3.1.4 — Zmiana połączeń:**

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

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

endmodule
```

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

**3.1.3 Zapis liczby**

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

$$X'Yv$$

(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.

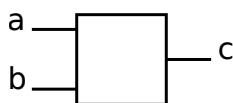
Np. jeśli portowi ma zostać przypisana liczba 123 zapisana na 8 bitach można tego dokonać na kilka sposobów:

**Kod 3.1.5 — Zapisanie wartości w różnych formatach:**

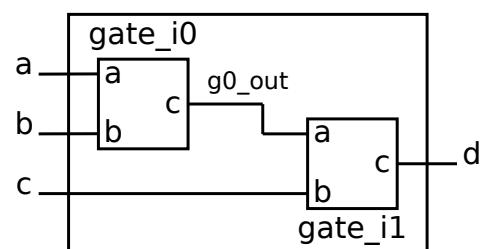
```
wire [7:0] value;
assign value=8'd123;           //decimal
assign value=8'h7b;            //hexadecimal
assign value=8'b01111011; //binary
```

### 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 zauważyc, 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 specyfczną **składnię modułu parametryzowanego** tj. użycie znaku `#` oraz fakt, że nazwa modułu jest oddzielona od nazwy instancji właśnie deklaracją parametrów (jest to nieco nietypowe i początkowo mało intuicyjne).

### 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++ i 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:



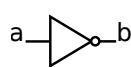
a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

#### 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:



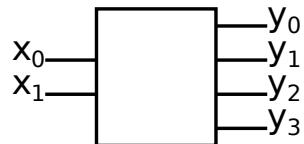
a	b
0	1
1	0

#### 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

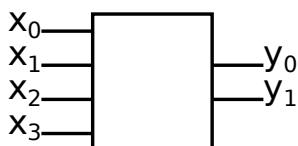
**Kod 3.1.11 — Dekoder:**

```
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:**

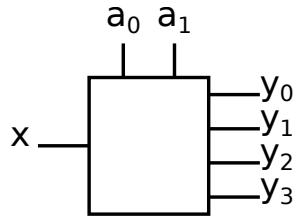
```
module encoder
(
    input [3:0]x,
    output [1:0]y
);

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

endmodule
```

**3.1.11 Demultiplexer**

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



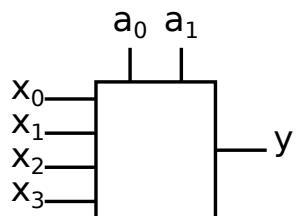
x	a <sub>1</sub>	a <sub>0</sub>	y <sub>3</sub>	y <sub>2</sub>	y <sub>1</sub>	y <sub>0</sub>
v	0	0	0	0	0	v
v	0	1	0	0	v	0
v	1	0	0	v	0	0
v	1	1	v	0	0	0

### Kod 3.1.13 — Demultiplexer:

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

### 3.1.12 Multiplexer

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



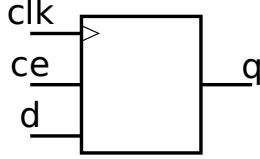
x <sub>3</sub>	x <sub>2</sub>	x <sub>1</sub>	x <sub>0</sub>	a <sub>1</sub>	a <sub>0</sub>	y
v <sub>3</sub>	v <sub>2</sub>	v <sub>1</sub>	v <sub>0</sub>	0	0	v <sub>0</sub>
v <sub>3</sub>	v <sub>2</sub>	v <sub>1</sub>	v <sub>0</sub>	0	1	v <sub>1</sub>
v <sub>3</sub>	v <sub>2</sub>	v <sub>1</sub>	v <sub>0</sub>	1	0	v <sub>2</sub>
v <sub>3</sub>	v <sub>2</sub>	v <sub>1</sub>	v <sub>0</sub>	1	1	v <sub>3</sub>

### Kod 3.1.14 — Multiplexer:

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

### 3.1.13 Rejestr

Rejestr jest podstawowym elementem „z pamięcią”. 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. Pomiedzy zboczami wartością wyjścia jest ustalona (zarejestrowana). Wartość wyjściowa jest opóźniona o jeden takt zegara w stosunku do wartości wejściowej. Dodatkowo możliwe jest włączanie/wyłączania rejestrów 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 rejestrów 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

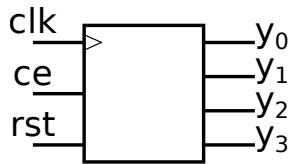
assign q=val;

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. Składnia `always @ (posedge clk)` oznacza, że kod zawarty wewnątrz wykona się tylko przy narastającym zboczu sygnału zegarowego (`clk`). W tak opisanym elemencie instrukcje wykonują się sekwencyjnie (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**

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



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

### Kod 3.1.16 — Licznik:

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

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

assign y=val;

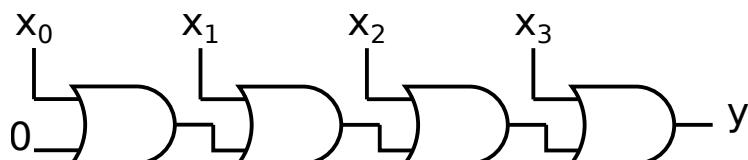
endmodule
```

### 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 projektanta, 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 parametryzowanej 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 zadawaną 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
    end
endgenerate
endmodule
```

**Kod 3.1.18 — Łańcuch bramek OR:**

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

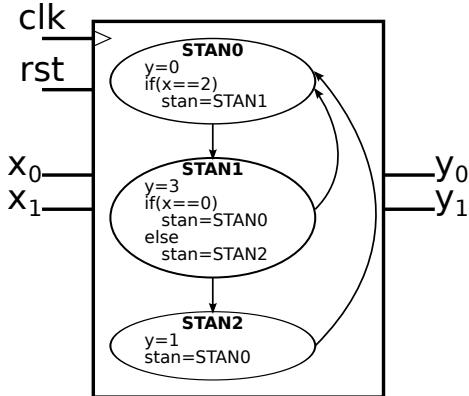
wire [LENGTH:0] chain;
assign chain[0]=1'b0;

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;

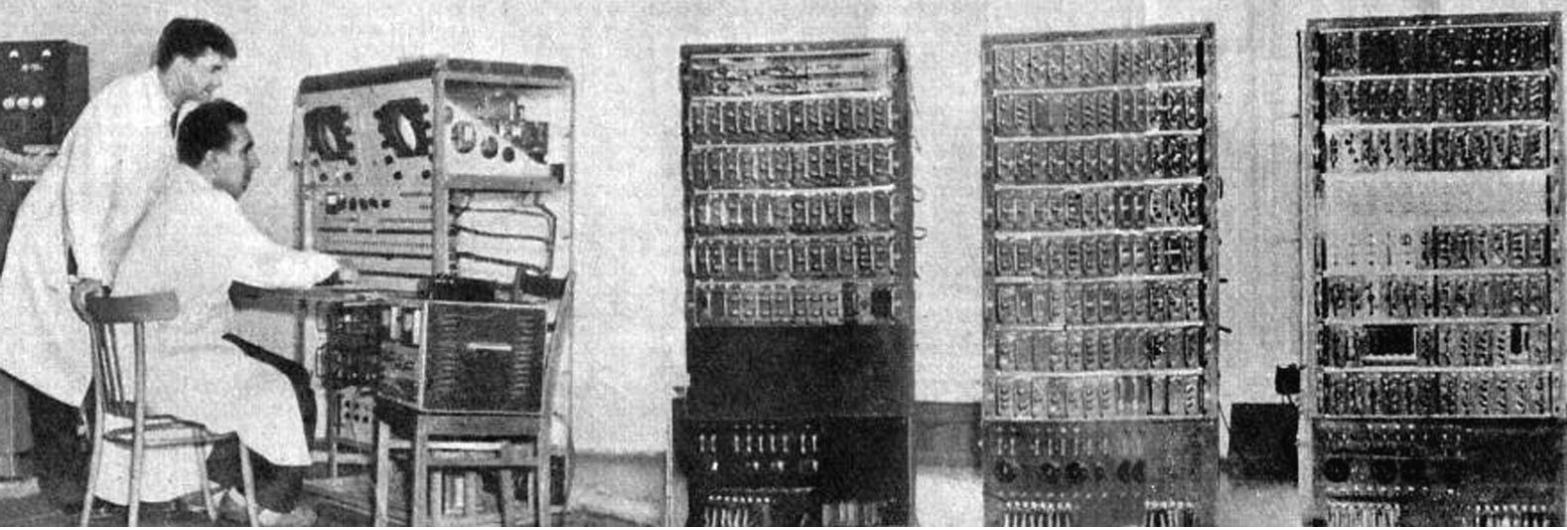
reg [1:0]state=STATE0;
reg [1:0]r_y;

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

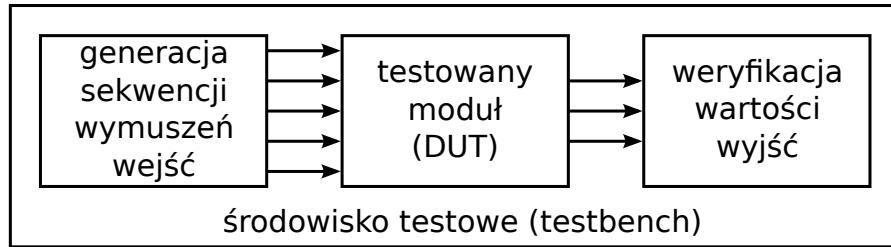
Testowanie i weryfikacja zaprojektowanych modułów sprzętowych jest bardzo **ważnym i złożonym** zagadnieniem. Wynika to bezpośrednio z zasygnalizowanego wcześniej problemu z oceną poprawności zaimplementowanego modułu sprzętowego. Po pierwsze, po uruchomieniu naszej logiki w układzie FPGA, zwykle dostajemy 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 (w odróżnieniu od pracy w językach programowania, gdzie wykrywanie błędów jest znaczco prostsze).

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 wewnętrz układowego, co ogranicza możliwości lokalizacji i usuwania błędów. Pewien wyjątek stanowi narzędzie Integrated Logic Analyzer (ILA). 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 mogłyby być kłopotliwe. Po trzecie, symulacja pozwala na zaoszczędzenie czasu potrzebnego na syntezowanie oraz implementację logiki do pliku bit, który umożliwia zaprogramowanie układu FPGA.

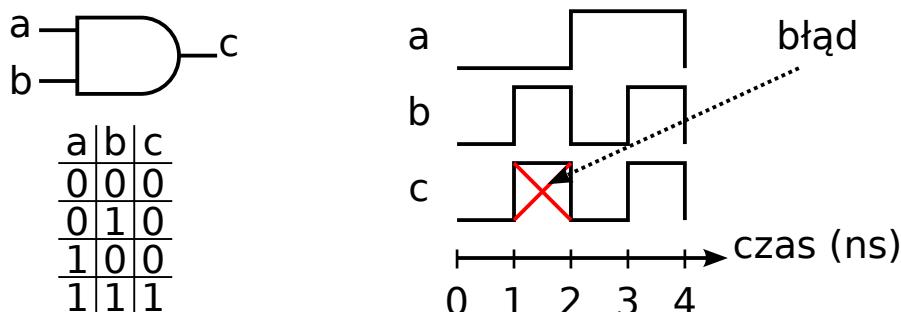
Ponieważ podstawowym elementem wykorzystywanym do opisu struktury układów scalonych w języku Verilog jest moduł, weryfikacja zaprojektowanego rozwiązania również opiera się o weryfikację działania poszczególnych modułów. W tym celu wykorzystywana jest metodologia przedstawiona na rysunku 4.1. Tworzone jest środowisko testowe (ang. *testbench*), które będzie podlegać 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 zauważać błędy. Dla bardziej skomplikowanych modułów, które posiadają dziesiątki portów, rejestrów opóźniających i maszyn stanów, określenie prawidłowych sygnałów stymulacyjnych oraz stwierdzenie czy otrzymane wyniki są poprawne, może stanowić nie lada wyzwanie (i zwykle wymaga 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 synchroicznej (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 zsyntezowalne”).

#### 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 Vivado jest uruchamiana poprzez wybór *Simulation->Run Simulation*. Uwaga. Symulacja uruchamiana jest dla pliku oznaczonego jako „top” w *Simulation Sources* – należy zawsze upewnić się, że wybrany jest poprawny plik.

#### 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 kodzie 4.1.3. Wykorzystano ją do generacji sygnału zegarowego. Warto zwrócić uwagę, że w większości przypadków będziemy mieli do czynienia z tzw. logiką synchroniczną, której 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 (jak nazwa wskazuje – odpowiada za inicjalizację).

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

#### Kod 4.1.2 — Generacja sekwencji wejściowej:

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

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

assign a=r_a;
assign b=r_b;

endmodule
```

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

**Kod 4.1.3 — Generacja sekwencji wejściowej:**

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

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

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

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

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

W powyższym przykładzie, wykorzystano dwa dodatkowe rejestrów *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

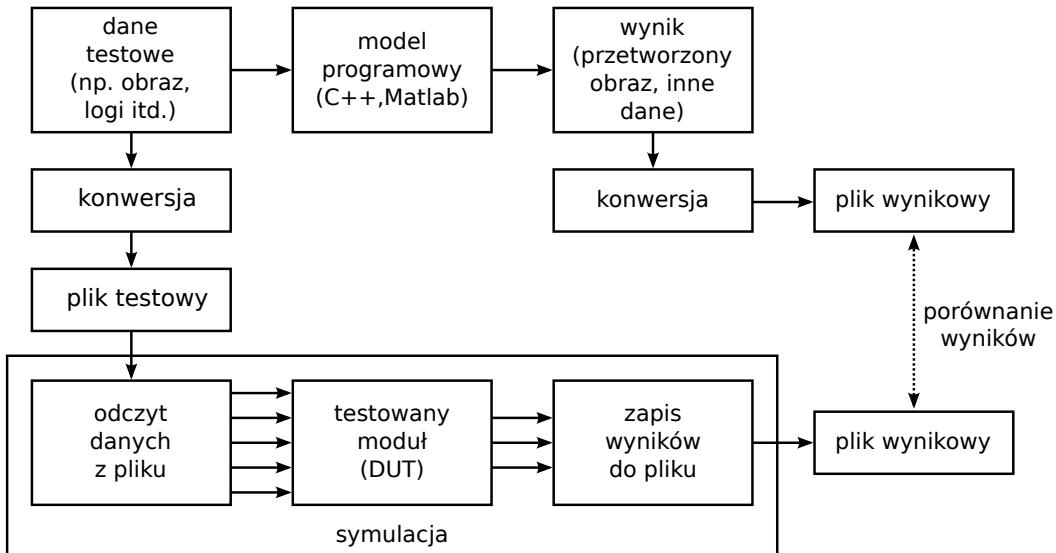
endmodule
```

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 video 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 modelem programowym i symulacji pozwala na sprawdzenie czy uzyskane wyniki są zgodne. Uzyskanie wyników niezgodnych świadczy o tym, że popełniono błąd albo podczas projektowania modułu sprzętowego albo podczas pisania modelu programowego. 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 rozpoczęjąc 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 symulacji, 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 sprzętce. 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 zauważać, 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 13.5.1. Natomiast moduł, który umożliwia zapisanie wartości portu *c* do pliku wynikowego zaprezentowano w kodzie 4.2.2.

##### Kod 4.2.1 — Odczyt:

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

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

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

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

endmodule
```

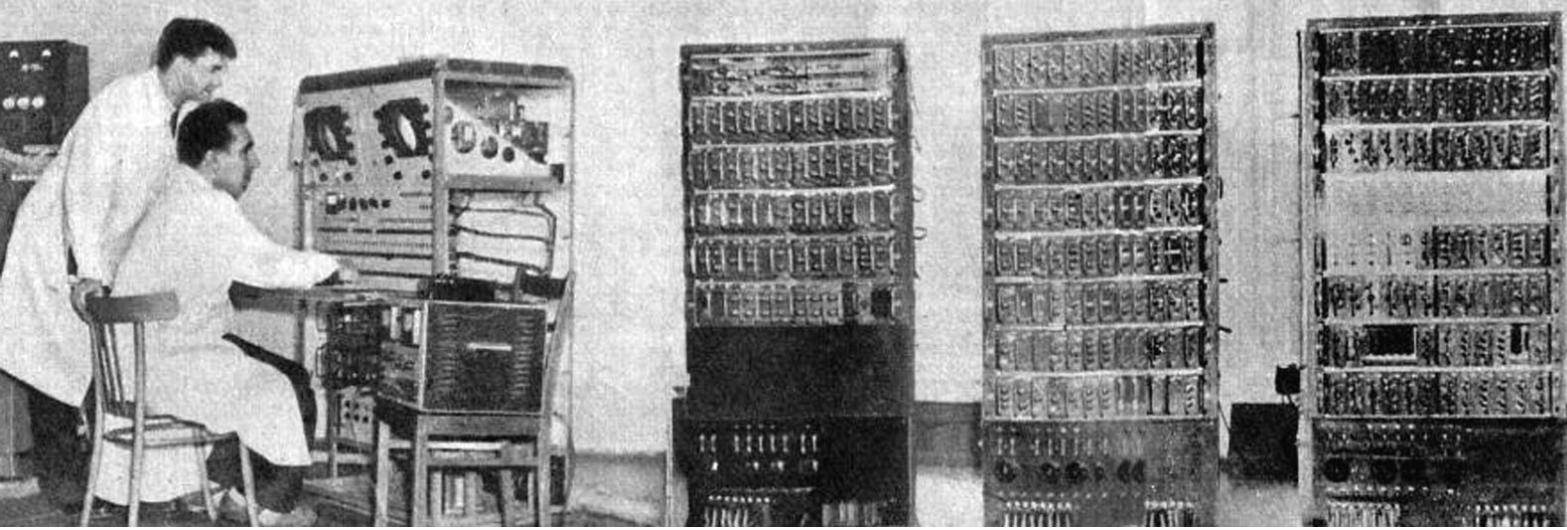
##### Kod 4.2.2 — Zapis:

```
module save_file
(
    input c
);

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

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

endmodule
```



## 5 — Verilog i weryfikacja – praktyka

W rozdziale zamieszczono zadania, które stanowią podsumowanie rozdziałów 3 oraz 4. Układ taki podyktowany jest przekonaniem, że zaimplementowany moduł (nawet najprostszy) powinien od razu zostać poddany weryfikacji. Jest to również dobra praktyka inżynierska – testujemy (dokładnie) kolejne fragmenty większej aplikacji. W ten sposób zyskujemy pewność, że „budujemy” system z poprawnych komponentów. Warto zatem poświęcić więcej czasu na testy częściowe (jednostkowe), niż po złożeniu całego systemu głowić się „dlaczego to nie działa” (co i tak sprowadza się do testów częściowych). 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 Vivado 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 syntezu modułu oraz sprawdź użycie zasobów (*Slice LUT's i LUT-FF pairs*). Np. w oknie, które pojawi się po zakończeniu syntezu należy wybrać *View Report* i wybrać *Utilization Report*. Czy mamy do czynienia z modułem synchronicznym, czy asynchronicznym ?
- w celu lepszego zrozumienia powiązania pomiędzy opisany kodem, a faktycznymi

zasobami układu FPGA oglądnijmy dwa schematy: RTL i po syntezie. Pierwszy jest dostępny w sekcji *RTL Analysis->Elaborated Design* drugi w *Synthesis->Synthesized Design*. Porównaj oba schematy. Zastanów się jakie zasoby FPGA wykorzystywane są do realizacji bramki AND.

- stwórz testbench. Dodaj nowy plik do projektu (*Add Source*) i wybierz *Add or create simulation sources*. Dobra praktyka to nazywanie testbench'a jako *tb\_nazwa\_modulu*. W ten sposób od razu można rozróżnić pliki do implementacji i testowania (trudniej się pomylić przy uruchamianiu symulacji).
- otwórz testbench i uzupełnij go analogicznie do kodu 4.1.3. Uwaga. Dla uproszczenia, w tym przypadku, **nie realizujemy** koncepcji trzech modułów tj. generatora sekwencji wejściowych, modułu testowanego oraz analizatora poprawności sygnałów wyjściowych. Ograniczmy się tylko do modułu nadziednego, w którym wygenerujemy sygnał testowy (wszystkie możliwe kombinacje sygnałów wejściowych) i instancji modułu testowanego (bramki AND). Poprawność sprawdzimy „ręcznie” – analizując wygenerowany przebieg. Parametr `LENGTH` można zdefiniować jako `parameter` w nagłówku modułu lub `localparam` wewnątrz modułu. Należy też zdefiniować `wire` na wyjście z modułu. Uwaga. Moduł nie powinien mieć wejść i wyjść.
- wykonaj symulację modułu. W tym celu w sekcji *Simulation* wybierz *Run Simulation->Run Behavioral Simulation*. Uwaga. Zawsze należy zwracać uwagę, jaki moduł wybrany jest w oknie *Sources*.

Ponieważ jest to pierwsza styczność z symulatorem Vivado warto go omówić. Po pierwsze, w odróżnieniu od wcześniejszego narzędzia dostępnego w ISE, jest to okno (widok), a nie osobna aplikacja.

Widok podzielony jest na trzy części:

- *Scope* – przełączalne z *Sources*, widoczna hierarchia modułów,
- *Objects* – lista wejść, wyjść, sygnałów, rejestrów itp,
- Przebieg sygnałów.

W oknie *Scope* wybieramy moduł, którego sygnały chcemy analizować. Stają się one widoczne w oknie *Objects*. Elementy te **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 *Run->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.

Przeanalizujmy teraz uzyskane wyniki. Widoczne są sygnały 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 (*Relaunch Simulation*). Wymagane jest ono w przypadku dokonania zmian w plikach źródłowych. Uwaga. Zmiana interfejsu modułu i inne poważne ingerencje wymagają ponownego uruchomienia symulacji. O tym kiedy dokładnie potrzebny jest reset symulacji można przekonać się „empirycznie”, po prostu narzędzie 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 symulatora (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 parametryzowanego modułu:

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

Opisany testbench należy przesymulować. Pewien problem stanowi *wire* związany 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 Vivado umożliwia także pracę z kodem. Przejdź do zakładki *Sources*. Wybierz plik z opisem modułu licznika. Otworzy się okno z kodem. Wybierz linijkę kodu i wstaw *breakpoint* – dla linii, w których jest to możliwe, wyświetlany jest czerwony okrąg. 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,

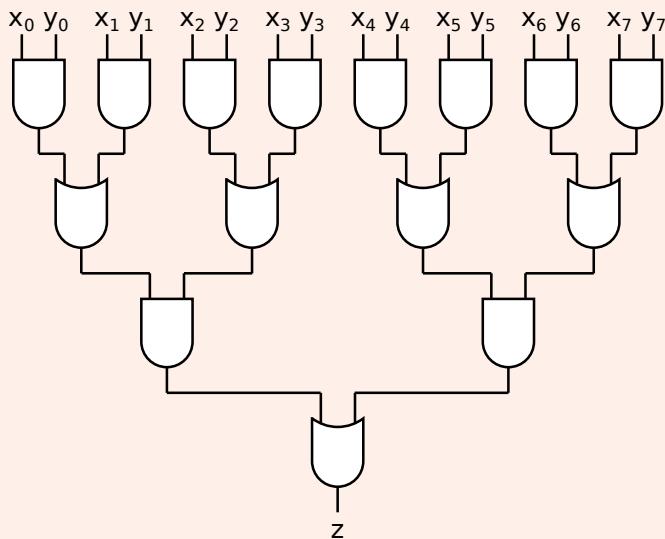
- pracę krokową – F8.

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 typy 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. Rozwiążanie 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 podglądać schemat RTL modułu – dobry sposób na sprawdzenie poprawności implementacji.

## **5.2 Zadania do wykonania w domu**

### 5.2.1 Linia opóźniająca

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

Proszę zaprojektować moduł, który posiada dwa parametry:

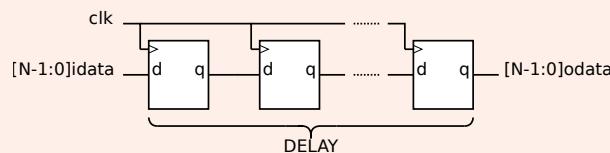
- N – szerokość portów wejściowego i wyjściowego w bitach,
  - DELAY – długość opóźnienia, które moduł powinien wprowadzać.

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  $\text{DELAY} > 0$  – generował  $\text{DELAY}$  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:  $\text{DELAY} = 0$  i  $\text{DELAY} > 0$ .

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  $\text{DELAY}$ . Wewnątrz modułu konieczne jest zrealizowanie instrukcji wyboru z wykorzystaniem *generate* – por. kod 3.1.17. Dla przypadku  $\text{DELAY} > 0$  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  $\text{DELAY}+1$ . Wewnątrz *generate* możemy to wykorzystać w sposób następujący:

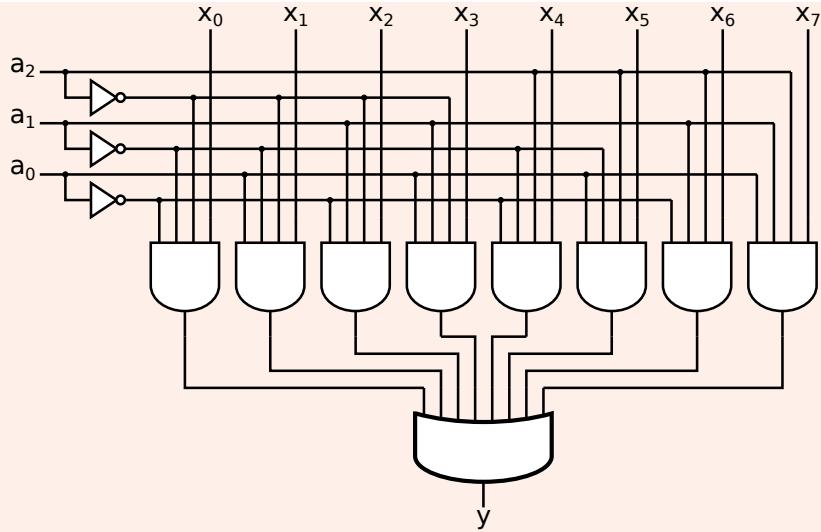
```
.idata(tdata[i]),  
.odata(tdata[i+1])
```

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

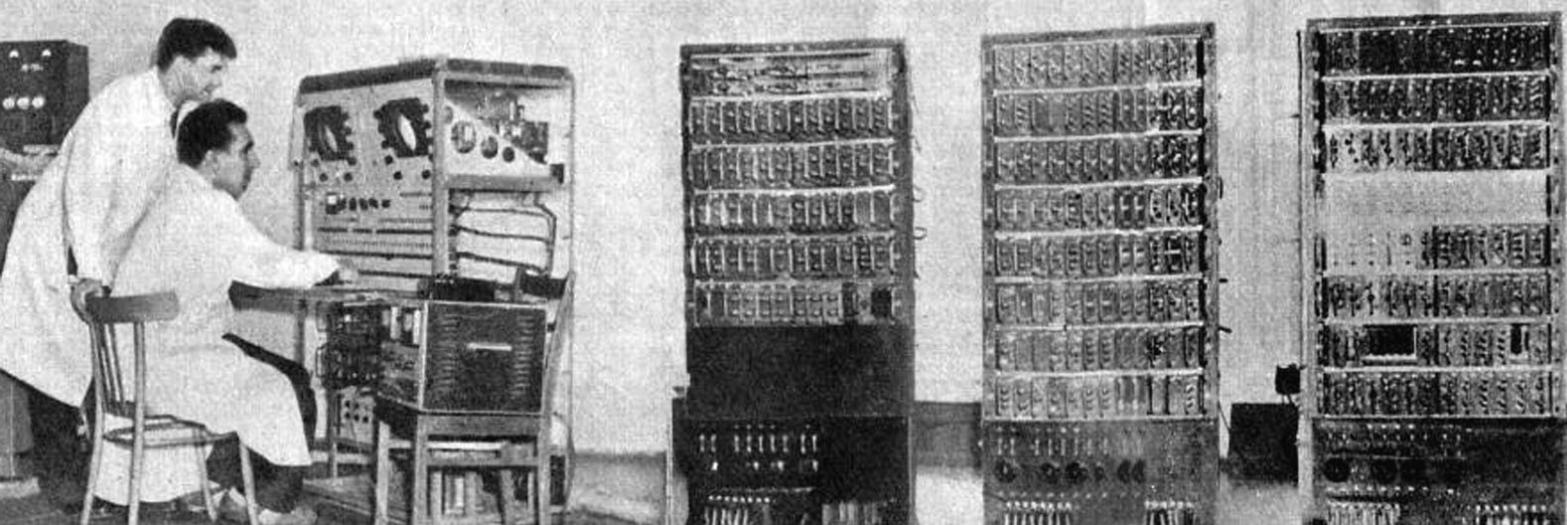
**Zadanie 5.5** Proszę opisać w języku Verilog następujący moduł:



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 wyjściu *txd* powinny być przesyłane kolejne bity portu *data* z utworzonego rejestrzu, od najmłodszego do najstarszego. Po przesłaniu wszystkich bitów należy przejść do stanu czwartego.
4. W ostatnim stanie, na wyjściu *txd* powinna być podana wartość '0' (bit stopu) oraz maszyna powinna wrócić do stanu pierwszego.

Opisana maszyna realizuje bardzo uproszczoną wersję układu 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 Vivado.
- 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` (to by oznaczało, że 8 bitów wysyłamy w jednym taktie zegara).

**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 usunąćauważone błędy.

**Wskazówki:**

- zaczynamy od wygenerowania nadrzędnego testbench'a, 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 13.5.1. 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 były 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 odczytce 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 behawioralną. 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” („Kate”). 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 modelem 10 wejściowej bramki OR. Proszę stworzyć nowy projekt w Vivado, dodać do niego pobrany plik (umieścić w folderze) i utworzyć środowisko testowe, które w automatyczny sposób umożliwia 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 otrzymywane dane do pliku (w postaci ciągu znaków ASCII) i dodać go do testbench'a. Działanie całego modułu należy sprawdzić symulacyjnie.

Uwagi:

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





## 7 — Operacje arytmetyczne

Główna 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, stało-przecinkowe), 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, który 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:

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

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

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

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

wartość minimalna: 0  
wartość maksymalna: 127

wartość minimalna: 0  
wartość maksymalna: 31

Wykonanie operacji arytmetycznych na wektorach A i B prowadzi do uzyskania wyników, których format jest inny niż 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	$c_7 \   \ c_6 \   \ c_5 \   \ c_4 \   \ c_3 \   \ c_2 \   \ c_1 \   \ c_0$ $N_{Y_c} = \max(N_{Ac}, N_{Bc}) + 1$
Y=A-B	-31	127	$\  z \  \ c_6 \   \ c_5 \   \ c_4 \   \ c_3 \   \ c_2 \   \ c_1 \   \ c_0$ $N_{Y_z} = 1 \quad N_{Y_c} = \max(N_{Ac}, N_{Bc})$
Y=A*B	0	3937	$c_{11} \   \ c_{10} \   \ c_9 \   \ c_8 \   \ c_7 \   \ c_6 \   \ c_5 \   \ c_4 \   \ c_3 \   \ c_2 \   \ c_1 \   \ c_0$ $N_{Y_c} = N_{Ac} + N_{Bc}$
Y=A/B	0	127	$c_6 \   \ c_5 \   \ c_4 \   \ c_3 \   \ c_2 \   \ c_1 \   \ c_0 \   \ u_4 \   \ u_3 \   \ u_2 \   \ u_1 \   \ u_0$ $N_{Y_c} = N_{Ac} \quad N_{Y_u} = N_{Bc}$

Należy zauważyć, że wynik dodawania i mnożenia dwóch nieujemnych 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:

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

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

$$A = \| z \| \ c_4 \ | \ c_3 \ | \ c_2 \ | \ c_1 \ | \ c_0 \| \quad N_c=5$$

wartość minimalna: -32  
wartość maksymalna: 31

$$B = \| z \| \ c_3 \ | \ c_2 \ | \ c_1 \ | \ c_0 \| \quad N_c=4$$

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

format

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline z & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline \end{array} \\ N_{Yz} = 1 \quad N_{Yc} = \max(N_{Ac}, N_{Bc}) + 1$$

Y=A-B	-47	47
-------	-----	----

$$\begin{array}{||c||c|c|c|c|c|c|c|} \hline z & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline \end{array} \\ N_{Yz} = 1 \quad N_{Yc} = \max(N_{Ac}, N_{Bc})$$

Y=A*B	-496	512
-------	------	-----

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline z & c_{10} & c_9 & c_8 & c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ \hline \end{array} \\ N_{Yz} = 1 \quad N_{Yc} = N_{Ac} + N_{Bc} + 1$$

Y=A/B	-32	32
-------	-----	----

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|} \hline z & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & u_3 & u_2 & u_1 & u_0 \\ \hline \end{array} \\ N_{Yz} = 1 \quad N_{Yc} = N_{Ac} + 1 \quad N_{Yu} = N_{Bc}$$

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 rówaniem:

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

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

$$A = \begin{array}{||c|c|c|c|c|c||c|c|} \hline c_4 & c_3 & c_2 & c_1 & c_0 & \parallel u_1 & u_0 \\ \hline \end{array} \\ N_c=5 \quad N_u=2$$

$$B = \begin{array}{||c|c|c|c|c||c|c|c|} \hline c_3 & c_2 & c_1 & c_0 & \parallel u_2 & u_1 & u_0 \\ \hline \end{array} \\ N_c=4 \quad N_u=3$$

wartość minimalna: 0

wartość minimalna: 0

wartość maksymalna: 31,75

wartość maksymalna: 15,875

rozdzielczość części ułamkowej: 0,25

rozdzielczość części ułamkowej: 0,125

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

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

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:

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

Poniżej przedstawiono sposób kodowania dla dwóch wektorów:

$$A = \left\| \begin{array}{c|c|c|c|c|c|c|c} z & c_3 & c_2 & c_1 & c_0 & u_1 & u_0 \\ \hline N_z=1 & N_c=4 & & & N_u=2 & & \end{array} \right\| \quad B = \left\| \begin{array}{c|c|c|c|c|c|c|c} z & c_2 & c_1 & c_0 & u_2 & u_1 & u_0 \\ \hline N_z=1 & N_c=3 & & & N_u=3 & & \end{array} \right\|$$

wartość minimalna: -16,0

wartość maksymalna: 15,75

rozdzierlczość części ułamkowej: 0,25

wartość minimalna: -8,0

wartość maksymalna: 7,875

rozdzierlczość 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

format

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

Y=A-B	-9,875	9,75
-------	--------	------

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

Y=A*B	-15,5	16,0
-------	-------	------

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

Y=A/B	-64	64
-------	-----	----

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

## 7.2 Zmienna długość słowa

Można zadać sobie pytanie: 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 zmiennoprzecinkowego na stałoprzecinkowy. Założmy, że mamy dany prosty algorytm operujący na liczbach typu *double* lub *float*. Oprócz specyficznych obliczeń, taka reprezentacja jest zwykle zupełnie satysfakcyjną i uznawana za dokładną. Chcemy teraz przejść na format stałoprzecinkowy. Musimy podjąć dwie decyzje. Po pierwsze trzeba ustalić ile bitów przeznaczmy na część całkowitą. Decyzja ta zwykle jest dość prosta, musimy tylko oszacować maksymalne i minimalne wartości jakie mogą wystąpić na każdym etapie obliczeń (zwykle robi się to analitycznie).

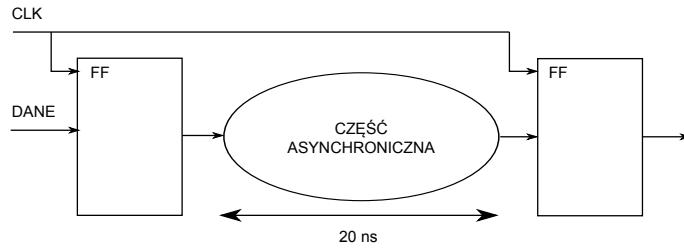
Po drugie, trzeba określić liczbę bitów, które przeznaczymy na część ułamkową. To już jest trudniejsze, gdyż precyzji *double* w ten sposób nigdy nie osiągniemy. Musimy się pogodzić z pewną utratą dokładności. Zatem zwykle tworzy się model stałoprzecinkowy algorytmu i empirycznie sprawdza, jak precyzja 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

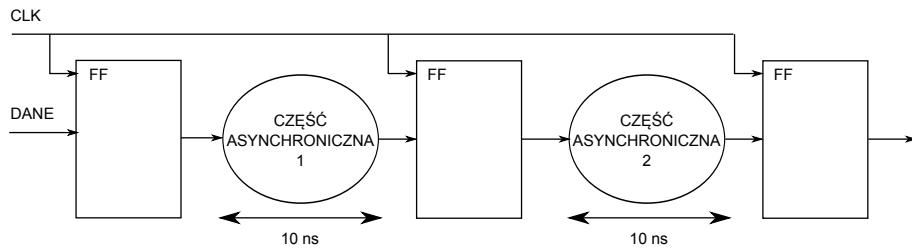
Oprócz formatu słowa, w którym są wykonywane obliczenia, drugim najważniejszym parametrem opisującym elementy wykonujące operacje arytmetyczne jest latencja. **Latencja** to liczba taktów zegara, która upływa od pojawienia się wartości na wejściu do ustalenia się prawidłowego rezultatu na wyjściu.

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.

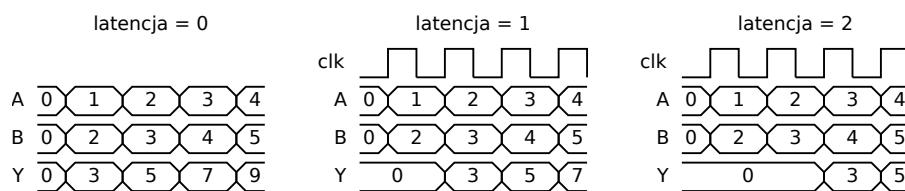


Rysunek 7.2: Logika synchroniczna i asynchroniczna – podział

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 nieskończonym krótkim czasie”. Logika asynchroniczna ma swoje czasy propagacji (czas przejścia sygnału przez element logiczny tj. np. LUT i zasoby połączeniowe). W przypadku, gdy latencja wynosi 1, wartość na wejściu zmienia się dopiero przy następnym narastającym zboczu zegara. W przypadku gdy latencja wynosi 2, wartość na wyjściu jest poprawna dopiero przy wystąpieniu drugiego narastającego zbocza zegara. Przy czym należy zauważać, że w przypadku latencji, opóźnione są wartości wejścia od wyjścia, natomiast poszczególne wyniki nie są od siebie opóźnione.



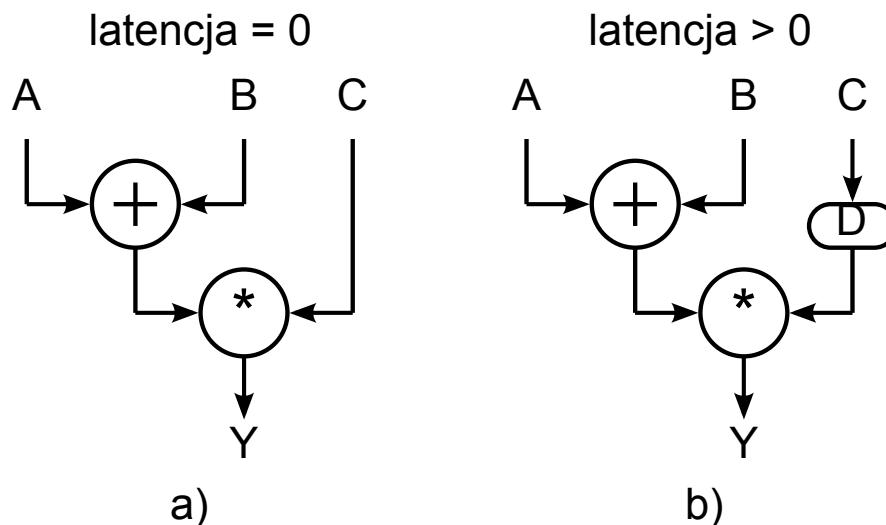
Rysunek 7.3: Wykonywanie obliczeń przez sumator z latencją równą 0, 1 i 2 taktom zegara

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 występuły w tej samej chwili czasu. Rozważmy przykład wykonywania operacji danej równaniem:

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

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

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



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 „natychmiast”, 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. Rozbiecie operacji złożonych na prostsze (podobnie jak w przypadku dodawania dwóch liczb metodą słupkową na kartce) zwiększa latencję, umożliwia jednak dalsze zwiększenie 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.

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

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 uzyskanie modułów pracujących z największą częstotliwością.

## 7.4 Pisanie a generowanie

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

### Kod 7.4.1 — Sumator o zerowej latencji:

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

### Kod 7.4.2 — Sumator o latencji równej jednemu cyklowi zegara:

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

Uzyskanie elementów obliczeniowych o większej latencji, wymaga wykorzystania predefiniowanych bloków logicznych (aplikacja *CORE generator*) i wygenerowania ich jako bloków *IP*

<sup>1</sup>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

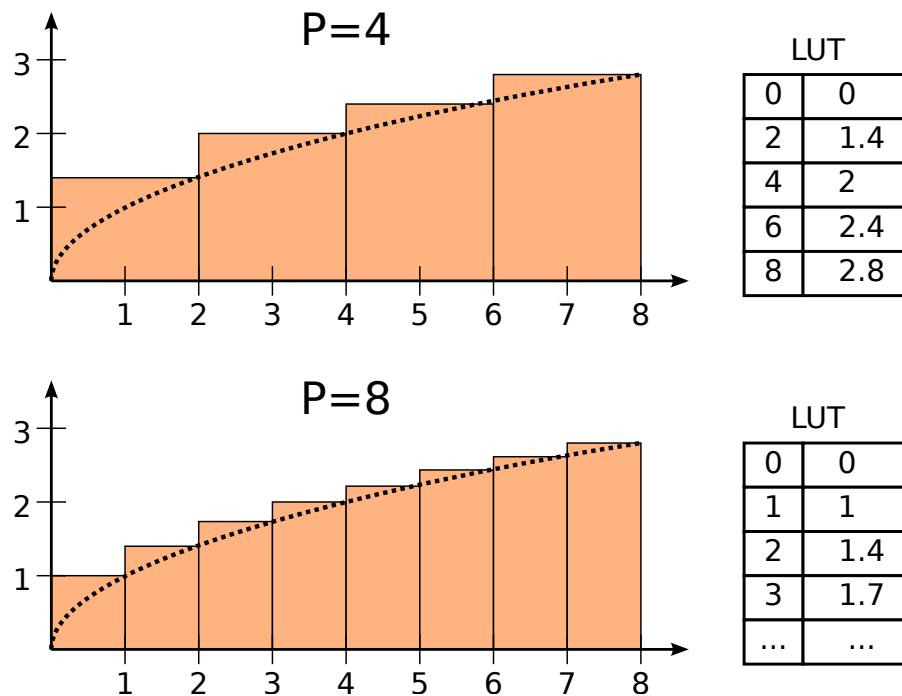
Bezpośrednie obliczanie wartości skomplikowanych funkcji na platformie rekonfigurowalnej jest utrudnione, wymaga bowiem zastosowania dużej liczby elementów logicznych i wielu iteracji metody przybliżonej. W związku z tym, często stosowanym podejściem, umożliwiającym realizację tych obliczeń w prosty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

W metodzie tej, dla pewnych argumentów obliczane są wartości funkcji i umieszczone w tablicy LUT (rysunek 7.5). Następnie podczas pracy systemu, wartość funkcji jest uzyskiwana 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.

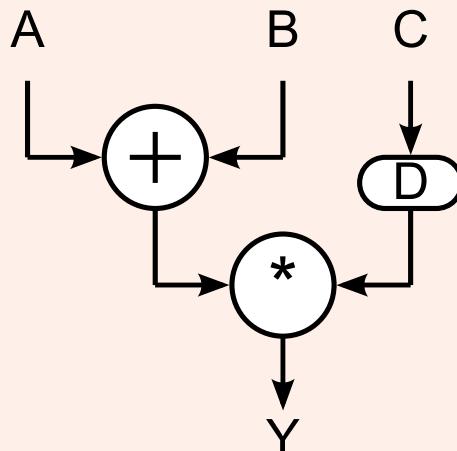
Na rysunku 7.5 przedstawiono przykład dla tablicowania funkcji  $y = \sqrt{x}$  dla dwóch przypadków. W pierwszym z nich, funkcja jest tablicowana dla 4 przedziałów. Tablica jest niewielka, jednak skutkuje to dużym błędem dla wartości bliskich 0. W drugim przypadku funkcja jest tablicowana dla 8 przedziałów, 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

Załącz, że argumenty  $A$ ,  $B$  i  $C$  to liczby rzeczywiste z przedziału  $[-1; 1]$ . W pierwszym kroku, w pakiecie Matlab zaimplementuj zadane równanie na liczbach *double*, a potem na stałoprzecinkowych z „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 – podpowiedź 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 Artix 7.
- Do realizacji obliczeń stałoprzecinkowych wykorzystamy „Fixed-Point Designer” Toolbox z pakietu Matlab. W pierwszym kroku należy stworzyć obiekt, który pozwoli na konwersję liczby zmiennoprzecinkowej na format stałoprzecinkowy zadaną precyzją. W tym celu wykorzystuje się polecenie `fi`:

#### Kod 7.6.1 — Konwersja z liczby zmiennoprzecinkowej na stałoprzecinkową:

```
value=0.32345;
sign=1; %0-unsigned value, 1-signed value           % 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: zmiennu i stałoprzecinkowym. Porównaj uzyskane wyniki – np. wypisz obok siebie na konsoli lub podglądnij w zakładce *Workspace*.

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

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

Na podstawie uzyskanych wyników wybierz precyzję obliczeń. Uwaga 1. Otrzymany wykres pokazuje, że nie z każdym zwiększeniem precyzji wiąże się poprawa dokładności. Osoby dociekliwe mogą przeanalizować przyczynę takiego stanu poprzez analizę 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 założmy, że błąd ma być raczej niewielki.

- Przechodzimy teraz do realizacji obliczeń w układzie FPGA. Utwórz nowy projekt w Vivado. Dodaj moduł nadzędny 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 precyją. 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. Otwórz *IP Catalog* W zakładce *Math Functions* wybierz *Adders & Subtracters*, a następnie *Adder/Subtracter*. Po dwukrotnym kliknięciu na moduł otworzy się okno konfiguratora. Tworzymy moduł w oparciu o mnożarki sprzętowe (*Implement using - DSP 48*), pracujemy na liczbach ze znakiem, szerokość ustawiamy zgodnie z użytą precyją, tryb dodawania, szerokość wyjścia zgodnie z podanymi wcześniej wskazówkami. Ustawiamy latencję na tryb *Automatic*. Zapamiętujemy ile ona wynosi. Warto zwrócić uwagę na:
  - zakładkę *Information*, w której przedstawione jest szacowane użycie zasobów logicznych,
  - zakładkę *Control*, w której możliwe jest ustawienie szeregu dodatkowych opcji

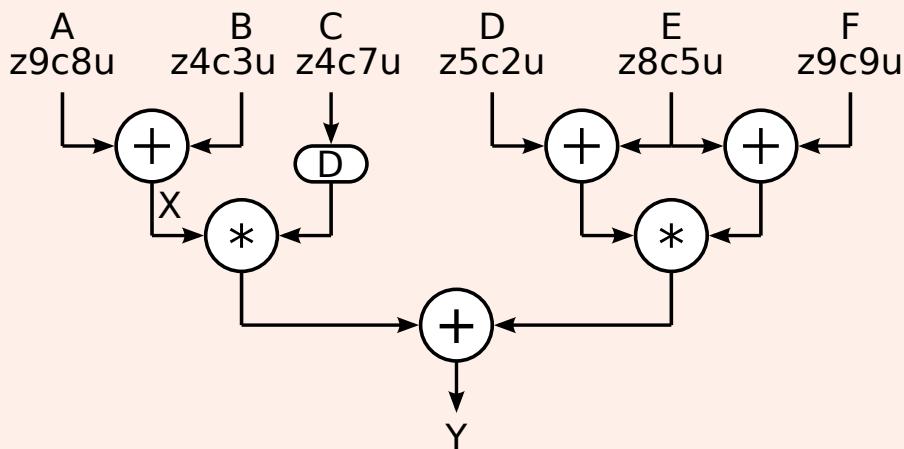
Klikamy *OK*, potem *Generate* i generujemy moduł (trwa chwilkę). Moduł pojawi się w hierarchii projektu. Należy go wstawić do projektu. W tym celu warto otworzyć plik *VHDL* (w hierarchii modułu), aby zobaczyć jak wygląda interfejs (*entity*). Następnie, już w module nadzędnym, umieszczaćmy instancję (jak w poprzednich ćwiczeniach). Uwaga. Proszę nie zapomnieć o zdefiniowaniu `wire` na wynik oraz o wrażliwości na wielkość znaków. Warto zauważyć, że powyższa konstrukcja to de-facto integracja kodu *VHDL* i *Verilog*. 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).
- Trzeci potrzebny moduł to mnożarka. Ponownie *IP Catalog* i dodajemy *Multiplier*. Konfigurujemy go. Wybieramy format danych (Input Options), wykorzystywane zasoby (*Multiplier Construction*). W zakładce *Output and Control* wybieramy szerokość wyjścia (zostawiamy automatyczną) oraz latencję (ustawiamy taką jak sugerowana optymalna), możemy także dodać port *CE*. W zakładce *Information* upewniamy się, że wykorzystujemy jedną mnożarkę. Moduł generujemy i wstawiamy do projektu.
- Na końcu należy wypisać na wyjście modułu „odpowiednie” bity z wyniku mnożenia.
- Następnie tworzymy testbench. Dodajemy generowanie zegara (sekcja *initial* – jak po przednio). Ustalamy początkowe wartości *A, B* i *C* na podstawie modelu programowego

(najprościej w postaci binarnej). Tu dobrze wykorzystać kolejną sekcję *initial*. Sprawdzamy czy uzyskujemy poprawny wynik. Dodatkowo sprawdzamy, czy wyniki pojawia się po tylu taktach zegara, po ilu się go spodziewamy. Uwaga. Do eliminacji błędów bardzo przydaje się model programowy. Dzięki niemu mamy wszystkie wyniki pośrednie i możemy precyjnie zlokalizować na którym etapie występuje nieprawidłowość. Warto pamiętać o możliwościach jakie daje symulator: 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};
```

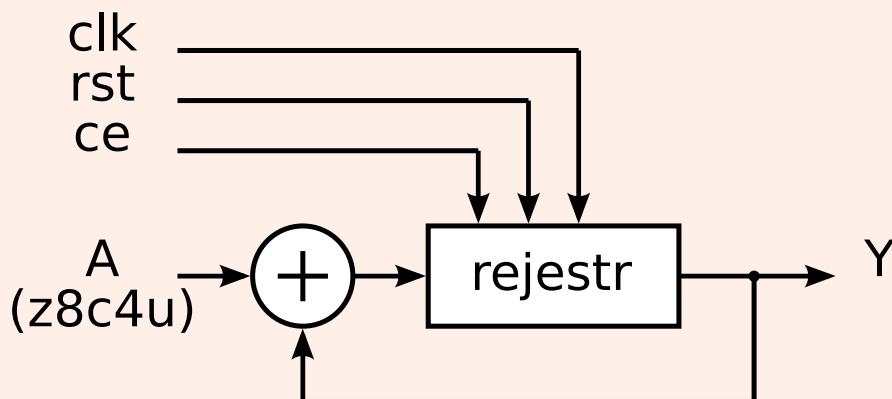
Jak w związku z tym należy zmodyfikować formaty na wejściach i wyjściach do poszczególnych sumatorów (na kartce)? Zmodyfikuj sumator (IP CORE). Sprawdź czy

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 Vivado.
- Proszę odpowiednio zmodyfikować formaty na wejściach i wyjściach poszczególnych mnożarek (na kartce).
- Proszę wygenerować odpowiednie mnożarki i zapisać latencje poszczególnych modułów.
- Czy latencja na wszystkich poziomach jest taka sama? Jeśli nie, to gdzie trzeba dodać moduły opóźniające? Proszę zmodyfikować schemat modułu (na kartce).
- Proszę opisać cały moduł z rysunku 7.7.
- Proszę stworzyć pełny model programowy w pakiecie Matlab. Przyjmijmy, że wartości poszczególnych portów wynoszą: A=-100,34    B=7,367    C=-4,92    D=9,111    E=-99,99    F=134,56  
Jaki jest błąd spowodowany realizacją tych obliczeń na liczbach stałoprzecinkowych?
- Proszę zapisać poszczególne liczby A-F binarnie (polecamie *bin* w programie Matlab).
- Proszę stworzyć środowisko testowe (testbench) umożliwiające weryfikację poprawności działania modułu.
- Proszę zaproponować jeszcze trzy zestawy wartości portów A-F i podać je w testbenchu na odpowiednie wejścia modułu arytmetycznego, w kolejnych taktach zegara.

**Zadanie 7.3** Zaprojektowane podczas laboratoriów moduły działały potokowo i umożliwiał 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 w rejestrze. Wystąpienie sygnału *rst* powinno umożliwić wyzerowanie wartości w rejestrze. Przyjmijmy założenie, że rozmiar rejestrów akumulacyjnych musi umożliwić dodanie maksymalnie 256 wartości z portu A.

Realizacja ćwiczenia – podpowiedź i uwagi:

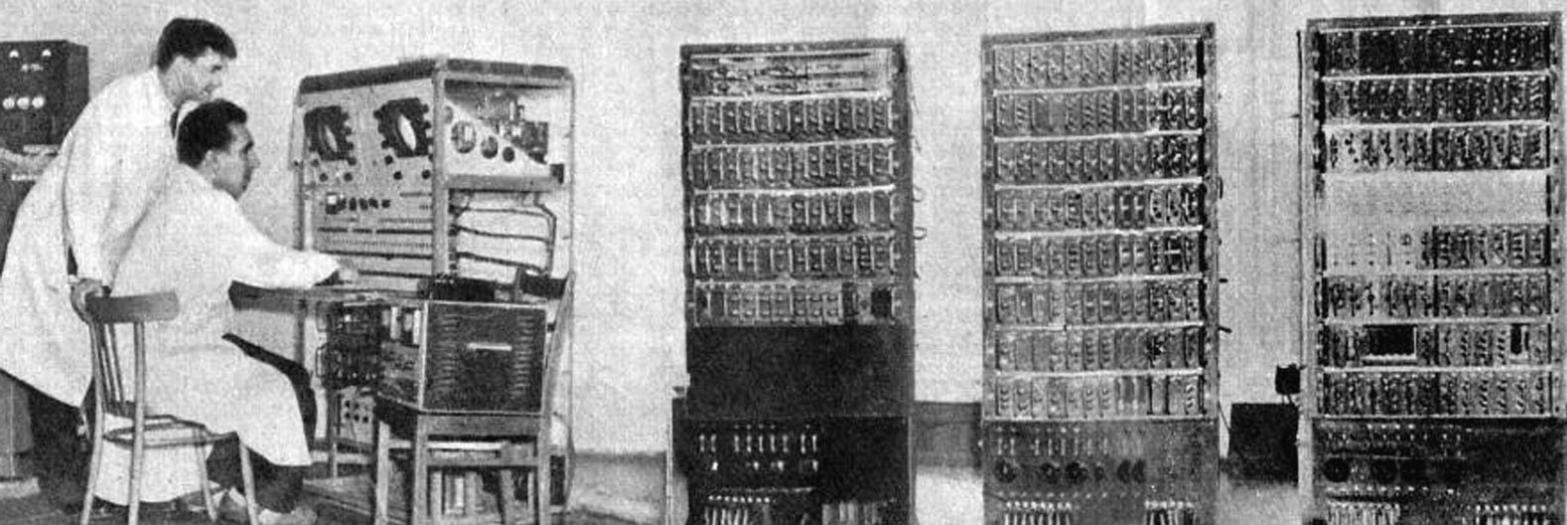
- Proszę zastanowić się jakie powinny być formaty wejściowe i wyjściowe w wykorzystywanym sumatorze. Jak szeroki musi być rejestr?
- Proszę wygenerować odpowiedni sumator. Jaka musi być latencja sumatora, aby możliwa była akumulacja wartości z portu przychodzących w następujących po sobie taktach zegara?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

## 7.8 Zadania dodatkowe

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

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

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



## 8 — Potokowe przetwarzanie i analiza obrazów

### 8.1 Wstęp teoretyczny

Typowy system wizyjny składa się z kamery, elementu realizującego obliczenia oraz ew. urządzenia do wizualizacji wyników lub ich transmisji 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 języku C++ i 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 [?], [?], [?], [?]. 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 odcienni szarości oraz detekcja

krawędzi).

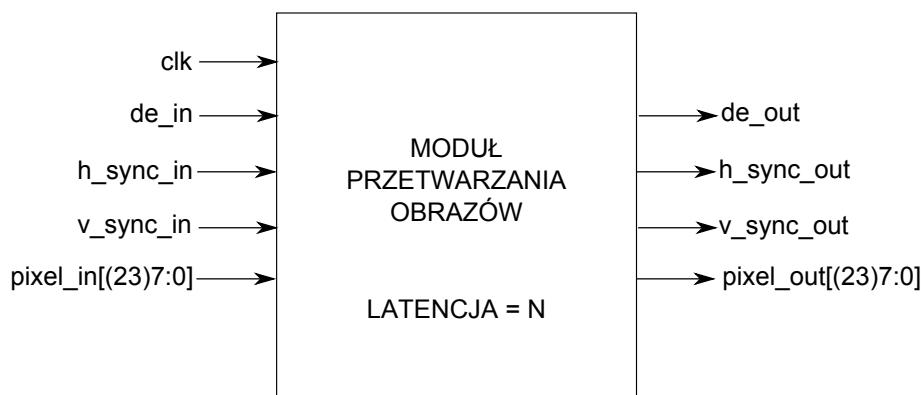
Natomiast analiza obrazu to proces wydobywania z niego istotnych informacji. Przy czym „istotność” ta jest ściśle powiązana z docelową aplikacją. Przykładami są: segmentacja obiektów (podział sceny na poszczególne obiekty: ludzi, samochody itp.), indeksacja (przypisanie do poszczególnych pikseli etykiet) 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 zwykła 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 (rozwiążanie 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.



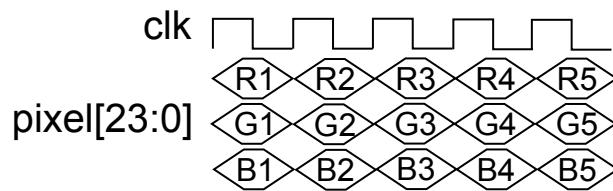
Rysunek 8.2: Schemat typowego modułu do przetwarzania obrazów

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 – przemieszczenie pionowe i poziome), 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 video 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

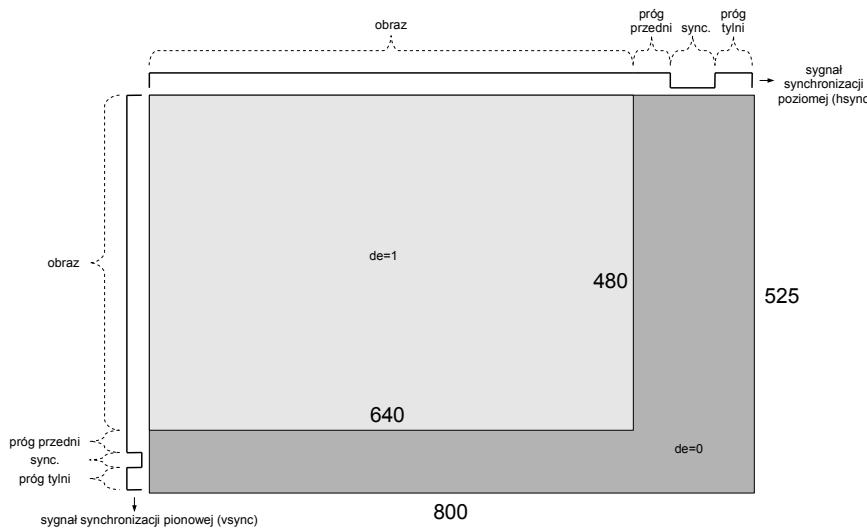
Sygnały sterujące tj. *de*, *hsync*, *vsync* wymagane są do poprawnego wyświetlania obrazu na ekranie monitora, a generowane są przez urządzenie dokonujące akwizycji tj. kamerę. Synchronizacja pionowa i pozioma początek swój wzięła z pierwszych telewizorów analogowych (CRT – ang. *cathode-ray tube*), które działały na zasadzie „ostrzeliwania” ekranu wiązką elektronów. Aby wyświetlić pełny obraz, „dzielko” poruszało się od lewej do prawej (skanowanie poziome) oraz od góry do dołu (skanowanie pionowe). Sygnały synchronizacji sterowały tym procesem. Przerwa w wyświetlanym 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  $640 \times 480$ .

Można zauważyć, że rzeczywista rozdzielcość obrazu jest większa od wyświetlonej ( $800 \times 525$  vs.  $640 \times 480$ ). 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 uwzględnić 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 ?



Rysunek 8.4: Synchronizacja dla obrazu o rozdzielczości  $640 \times 480$

To zależy od implementowanej operacji. Np. w dodawaniu dwóch obrazów czy operacji LUT (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 **sprawą 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 w pakiecie Vivado.

Utwórz nowy projekt w pakiecie Vivado (nazwa *hdmi\_vga\_zybo*, karta Zybo). Dodaj źródła **modelu symulacyjnego** (w archiwum *hdmi\_vga\_zybo\_src*):

- *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*).

Uwaga. Proszę w kreatorze zaznaczyć *Add or create simulation sources* oraz później *Copy sources into project*.

W pliku *hdmi\_in.v* ustaw poprawną ścieżkę do obrazka testowego *geirangerfjord\_64.ppm* (najlepiej bezwzględną). Uruchom symulację (behawioralną) modułu *tb\_hdmi*. Uwaga. Aby oszczędzić czas operujemy na obrazie o rozdzielcości  $64 \times 64$ . Symulacja nawet prostych operacji przetwarzania i analizy obrazu jest dość czasochłonna.

Sprawdź, czy w wyniku symulacji otrzymano poprawny obraz. Znajduje się on w folderze *hdmi\_vga\_zybo.sim/sim\_1/behav/*. Zwróć uwagę na odpowiedni czas trwania symulacji (**min. 20 us**). Zwróć również uwagę na przebieg sygnałów *de*, *hsync* i *vsync*.

Uwaga. Proszę zauważać, że sygnały *hsync* i *vsync* mają odwrotną polaryzację, niż to jest pokazane na rysunku 8.4. Wynika to ze specyfiki układu Zybo i może być istotne przy realizacji własnych projektów.

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

## 8.4 Uruchomienie toru wizyjnego na karcie Zybo

#### Zadanie 8.2 Uruchom tor wizyjny na karcie Zybo. Potrzebne pliki zawarte są w archiwum. W efekcie obraz z kamery wyświetlany będzie bezpośrednio na monitorze LCD.

Sposób podłączenia karty Zybo przedstawiono na rysunku ???. Uwagi:

- na karcie Zybo używamy portów HDMI (w trybie IN) oraz VGA,
- do HDMI podpinamy sygnał wizyjny z komputera PC,
- wyjście VGA łączymy z monitorem LCD,
- wyniki oglądamy po wybraniu odpowiedniego wejścia na monitorze (w tym przypadku VGA).

Przebieg ćwiczenia:

- W oprogramowaniu Vivado, w odróżnieniu od wcześniejszego ISE, dość mocno promowana jest praca ze schematami blokowymi – w sposób podobny do tego, jaki jest dostępny w Simulinku firmy MathWorks. Realizacja toru wizyjnego stanowi „dobrą okazję” do zapoznania się z tym podejściem.
- W tym samym projekcie, w którym uruchamialiśmy symulację, z zakładki *IP Integrator* wybieramy *Create Block Design* oraz nazywamy schemat np. *hdmi\_vga*. Otworzy się okno *Diagram*.

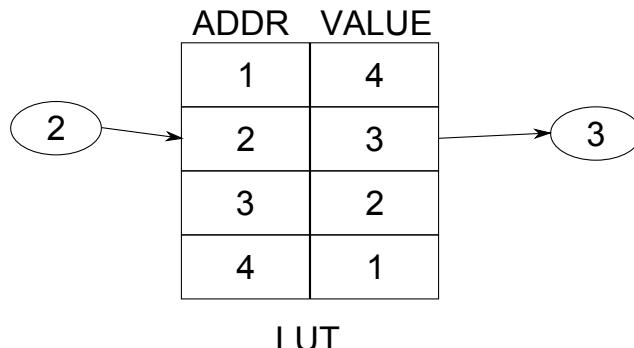
- Do budowy toru wizyjnego potrzebujemy zasadniczo dwóch komponentów: modułu obsługującego wejście HDMI oraz modułu odpowiadającego za wyjście VGA. Ponadto potrzebny będzie pomocniczy generator sygnału zegarowego.
- Ze strony kursu pobierz archiwum *hdmi\_vga\_ip\_repo* i rozpakuj w tym samym folderze co projekt (ale niekoniecznie wewnątrz projektu).
- Dodaj do projektu plik z ograniczeniami użytkownika *Zybo\_HDMI.xdc* (sposób dodawania plików z ograniczeniami przedstawiony był w pierwszym ćwiczeniu). Plik jest dostępny w głównym folderze projektu *hdmi\_vga\_zybo*.
- Wybierz *Project Manager->IP Catalog*. W otwartym oknie kliknij prawym przyciskiem myszy i wybierz *Add Repository*. Podaj ścieżkę do folderu *hdmi\_vga\_ip\_repo*. Otworzy się okno, w którym powinna pojawić się informacja, że dodaliśmy dwa moduły IP oraz jeden interfejs do projektu. Ponadto w hierarchii pojawi się *User Repository*, a wewnątrz dwa moduły (*DVI to RGB Video Decoder (Sink)* oraz *RGB to VGA output*) , które trzeba dodać do projektu poprzez dwukrotne kliknięcie i wybranie *Add IP to Block Desing*. Ewentualnym komunikatem o błędzie proszę się nie przejmować. Druga próba dodania modułu powinna się udać.
- Konfigurację zaczynamy od modułu *dvi2rgb* (czyli wejścia):
  - najeżdżamy kursorem na port TMDS (pojawi się symbol ołówka), klikamy prawy przycisk myszy i wybieramy *Create Interface Port*). Nazywamy go *hdmi\_in* (köniecznie tak, z uwagi na wymóg zachowania zgodności z plikiem *xdc*), w polu VLVN powinno być *diligent...:tmds\_rtl:1.0*, tryb pozostawiamy SLAVE.
  - moduł potrzebuje zegara referencyjnego (port *RefClk*) o częstotliwości 200 MHz tj. większej od spodziewanego zegara piksela . Aby go dostarczyć musimy wykorzystać *Clocking Wizard*. W zakładce *IP Catalog*, w polu *Search* wpisz *Clock* i dodaj do schematu moduł *Clocking Wizard*. Następnie należy stworzyć interfejs dla portu *clk\_in1* – najechać na port (ołówek), prawy przycisk myszy, opcja *Create Port*. Ustawiamy nazwę na *clk* i częstotliwość na 125 MHz.
  - Port *reset* należy na stałe podpiąć do '0' (GND). W tym celu należy wykorzystać moduł *Constant* (odszukać w *IP Catalog* i dodać do schematu). Należy ustawić jego wartości na 0 (domyślnie jest 1). Jego wyjście należy też podłączyć do portów *aRst* i *pRst* modułu *dvi2rgb* – łączenie poprzez naciśnięcie jednego portu i przeciągnięci linii do drugiego.
  - Na końcu należy skonfigurować sam moduł *Clocking Wizard*. Po dwukrotnym kliknięciu otworzy się konfigurator. W zakładce *Clocking Options* ustawiamy: *Primitive* na *PLL* oraz częstotliwość zegara wejściowego na *Auto*, a w zakładce *Output Clocks* na 200 MHz (*Requested*). Wyjście łączymy z wejściem *RefClk* modułu *dvi2rgb*.
  - Następnie konfigurujemy moduł *dvi2rgb* – ustawiamy rozdzielcość  $1280 \times 720$  oraz *TMDS clock range* na  $< 80 \text{ MHz}$  (*720p*).
  - Wyjścia modułu *dvi2rgb* łączymy z wyjściami *rgb2vga - RGB - vid\_in*, *PixelClk - PixelClk* (odpowiednio piksel 24 bity oraz sygnały synchronizacji (można podglądać po rozwinięciu *RGB* – kliknąć na znak „+”) i zegar piksela.
  - Dla pozostałego wyjścia *DDC* należy stworzyć port (*Create Interface Port*) i nazwać go *hdmi\_in\_ddc*. Należy pozostawić domyślny typ: *iic\_rtl* oraz *MASTER*. Warto zauważyc, że jest to interfejs *I<sup>2</sup>C* wchodzący w skład HDMI, który służy do komunikacji pomiędzy poszczególnymi urządzeniami (np. ustalanie dostępnych trybów wyświetlania). Uwaga. Pozostałe porty to resety (nie będziemy ich używać) oraz sygnał *aPixelClkLckd* – flaga ustawiana po synchronizacji modułu z sygnałem video.

- Dla kompletności należy stworzyć jeszcze port wyjściowy *hdmi\_hpdi* i podpiąć go na stałe do VCC (moduł *Constant* z wartością 1) oraz port wyjściowy *hdmi\_out\_en* i podpiąć go na stałe do GND (to wynika z możliwości użycia portu jako wyjściowego).
- Na koniec pozostało podłączenie wyjść z modułu *rgb2vga*. Należy dla każdego z nich wybrać *Create Port* i pozostawić parametry domyślne. Warto jeszcze raz podkreślić, że nazwy portów muszą korespondować z wartościami w pliku *xdc*.
- W celu sprawdzenia poprawności schematu możemy wykorzystać polecenie *Validate Desig* – klawisz F6, a celu automatycznej re-organizacji schematu polecenie *Regenerate Layout*.
- Po ukończeniu schematu w zakładce *Sources* klikamy prawym przyciskiem myszy na plik schematu (*hdmi\_vga.bd*) i wybieramy *Create HDL Wrapper* oraz później *Generate Output Products* – zaznaczamy *Synthesis Options* na *Global* (tj. globalne opcje syntezy). Projekt jest gotowy do syntezy i implementacji (o ile nie popełniliśmy błędów).  
Uwaga. Proces konfigurowania schematu i toru wizyjnego „od zera” jest dość żmudny, jednak wydaje się, że ma wartość dydaktyczną, a sam sposób postępowania może być przydatny w bardziej zaawansowanych projektach.
- Przed uruchomieniem syntezy należy jeszcze naprawić błąd z dostępu do pliku *900p\_edid.txt* (nota bene jest to plik, który definiuje w jaki sposób wejście HDMI karty Zybo będzie widziane przez inne urządzenia np. przez komputer czy kamerę – dostępne rozdzielcości itp.). W hierarchii projektu (*Design Sources*) rozwijamy moduł *dvi2rgb* i odszukujemy plik *dvi2rgb.vhd*. Otwieramy go i patrzymy gdzie się znajduje w projekcie (ścieżka jest wyświetcona nad oknem edytora). Do tego folderu musimy skopiować plik *900p\_edid.txt*, który znajduje się w folderze *ip\_repo/dvi2rgb\_v1\_6/src/*.
- Przechodzimy przez wszystkie fazy implementacji projektu (eliminując potencjalne błędy) i otrzymujemy plik konfiguracyjny.
- Sprawdzenie czy system działa wymaga trochę „gimnastyki”. Zakładamy, że monitor jest podłączony do komputera po kablu DVI. Wyjście HDMI z komputera podpinamy do karty Zybo. Wyjście VGA z Zybo podłączamy do monitora.
- W konsoli wykonujemy polecenie *xrandr*. Wyświetli się lista wspieranych trybów. Upewniamy się, że dla HDMI jest to  $1280 \times 720$ . Ewentualnie można to ustawić polecienniem: *xrandr -output HDMIX -mode 1280x720* (X – odpowiedni numer). Proszę też wykonać takie polecenie: *xrandr -output HDMI1 -set "Broadcast RGBFull"*. Powoduje ono wyświetlanie „pełnego” zakresu wartości RGB.
- Zasadniczo efekt powinien być taki, że pulpit wyświetli nam się na ekranie, przy czym sygnał będzie przechodził przez Zybo.
- W laboratorium dostępna jest kamera IP (tj. umożliwiająca transmisję obrazu po sieci Ethernet). Możliwe jest zatem „emulowanie” przetwarzania rzeczywistego obrazu poprzez odtworzenie strumienia wizyjnego na rozszerzonym pulpicie. Sposób dostępu do kamery jest opisany na stronie kursu.

## 8.5 Realizacja operacji LUT

Operacja LUT (ang. *Look-Up Table*) to jedna z najprostszych operacji punktowych w przetwarzaniu obrazów. Polega na przekształceniu wartości piksela zgodnie z uprzednio zdefiniowaną tablicą przekodowania. W FPGA realizowana jest zwykle z wykorzystaniem modułu ROM (ang. *Read-Only Memory* – pamięci rozproszonej lub blokowej). W tym przypadku operacja LUT działa tak, że jako adres (ADDR) podaje się wartość piksela, a w wyniku zwracana jest wartość zapisana w pamięci pod tym adresem (VAL), która staje się nową wartością piksela. Schematycznie zostało to przedstawione na rysunku 8.5. W tym przypadku piksel o wartości 2

zostaje zastąpiony pikselem o wartości 3.



Rysunek 8.5: 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 Zybo.

Uwagi do realizacji:

- W ramach ćwiczenia zademonstrujemy tworzenie własnych modułów IP (bloczków, które możemy użyć na schemacie), dlatego działanie rozpoczęniemy od stworzenia nowego projektu w Vivado (najlepiej otworzyć drugąinstancję aplikacji). Projekt nazywamy *vp* – od *video processing*. Tworzymy nowy moduł Verilog – *vp* oraz wybieramy kartę Zybo. Ustawiamy interfejs taki jak na rysunku 8.2, w wersji 24 bitowej. Będzie to nasz plik nadzędny, w którym będziemy realizować wszystkie operacje przetwarzania strumienia wizyjnego. Na jego podstawie utworzymy później własny moduł IP.
- W tym samym projekcie tworzymy nowy IP Core o nazwie LUT. W *IP Catalog* odszukujemy element *Distributed Memory Generator*. Nie używamy pamięci blokowej (BRAM), gdyż jest ona zbyt duża dla naszych potrzeb.
- W oknie konfiguratora trzeba wybrać nazwę (LUT) oraz 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 (Pipeline 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 archiwum *hdmi\_vga\_zybo*. Podglądni wczytaną zawartość. Wygeneruj moduł.
- Instancję LUT dodaj do modułu *vp* i odpowiednio połącz. De facto potrzebujemy trzech instancji – przetwarzamy każdą składową barwną niezależnie.
- Poprawność metodologiczna wymaga aby opóźnić sygnały synchronizacji *de*, *hsync* i *vsync*. Wynika to z synchroniczności pamięci (opcja *Registered*). Najprościej do tego wykorzystać trzy rejesty pomocnicze i proces (instrukcja *always*).

**Kod 8.5.1 — Przykład realizacji opóźnienia o jeden takt:**

```
reg r_de      = 0;
reg r_hsync   = 0;
reg r_vsync   = 0;

always @ (posedge clk)
begin
    r_de <= de_in;
    r_hsync <= hsync_in;
    r_vsync <= vsync_in;
end

assign de_out = r_de;
assign hsync_out = r_hsync;
assign vsync_out = r_vsync;
```

- Przeprowadź syntezę modułu – celem sprawdzenia poprawności.
- Przystępujemy do tworzenia modułu. W pierwszym kroku skopiuj cały projekt *vp* do tymczasowego folderu (jakby coś miało się zepsuć...). Z menu *Tools* wybierz *Create and Package IP*. Pozostawiamy opcję *Package your current project*. Po zatwierdzeniu opcji (bez modyfikacji) otworzy się okno *Package IP*. Przechodzimy do zakładki *Review and Package* i wykonujemy *Package IP*.
- Wracamy do „poprzedniego” Vivado. W *IP Catalog*, po dodaniu (*Add Repository*), będzie widoczny nasz moduł.
- Po pierwsze sprawdzamy go symulacyjnie. Wstawiamy instancję *vp* do pliku *tb\_hdmi.v*. Sposób postępowania został opisany wcześniej, jednak warto zwrócić uwagę na podłączenie sygnału piksela. Moduły wejścia i wyjścia mają poszczególne składowe wydzielone, a nasz „zintegrowaną”. Dla przypomnienia operator konkatenacji sygnałów to: *a*, *b*, *c* – należy go tu zastosować. **Uwaga.** Przy przypisywaniu wartości wyjściowych (sekcja *“Output assignment”*) warto raczej komentować poprzednie przypisania, niż je nadpisywać. W ten sposób stworzymy sobie „multiplekser” i łatwiej będzie nam wracać do poprzednich wersji.  
Uruchamiamy symulację behawioralną.  
Uwaga. Wynikowy rysunek jest istotnie inny niż wejściowy. Sprawdź jaką funkcję LUT realizuje stworzony moduł.
- Ostatnim krokiem będzie uruchomienie modułu na karcie Zybo. Mając wygenerowany bloczek IP jest to już dość proste – należy „przerwać” bezpośrednie połączenie pomiędzy *dvi2rgb*, a *rgb2vga* i wstawić tam moduł *vp*. Uwaga. Dostęp do poszczególnych sygnałów po naciśnięciu znaku '+'. Następnie należy przeprowadzić wszystkie niezbędne kroki i uruchomić moduł na karcie Zybo.

## 8.6 Zadania do wykonania w domu

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

Podpowiedzi:

- Klucz to postać pliku \*.coe. Należy wykonać analizę istniejącego lub przeglądając do-

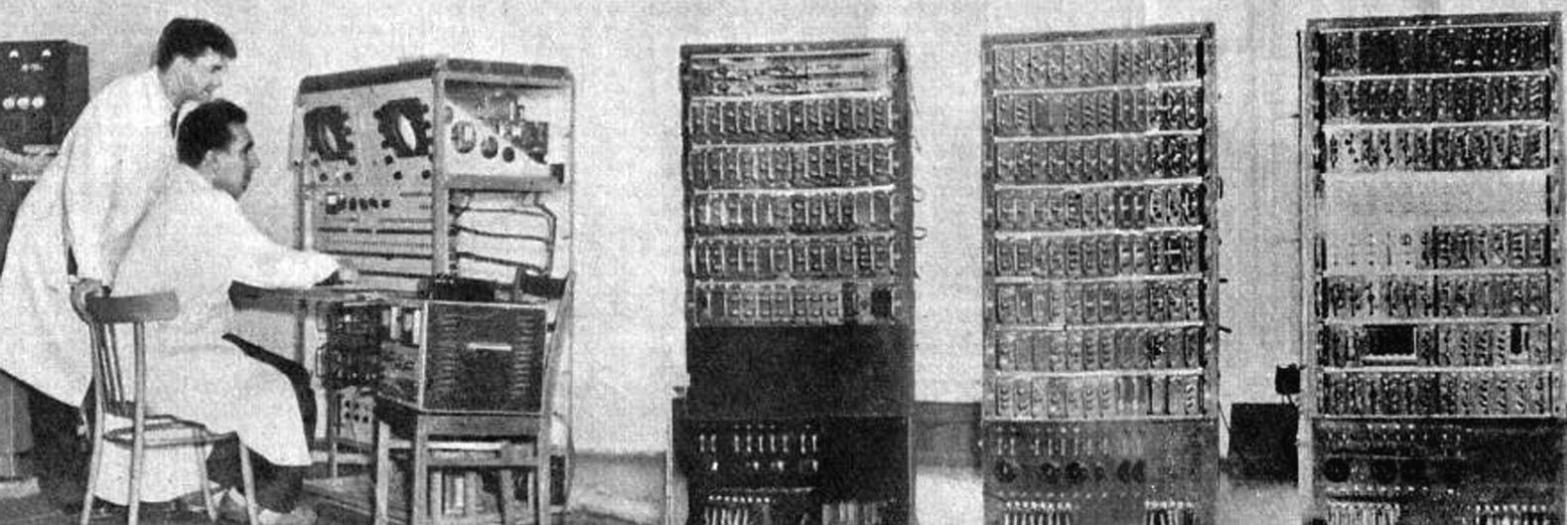
kumentację modułu *Distributed Memory*. Następnie trzeba utworzyć nowy plik \*.coe i podmienić w module LUT.

- 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 Zybo.

**Zadanie 8.5** Wykonaj testy symulacyjne modułu LUT zrealizowanego ma pamięci *Block RAM* dla różnych modeli tj. behawioralnego, syntezie i implementacji. Zaobserwuj różnice.

Przebieg zadania:

- utwórz nowy projekt w Vivado.
- utwórz nowy plik Verilog – *main\_LUT.v*. Jako wejście ustaw: *clk* i *addr* (8 bitów), a wyjście *dout* (8 bitów). Wewnątrz modułu utwórz instancję LUT, ale tym razem opartą nie o pamięć rozproszoną (ang. *distributed*), a blokową. **Uwaga.** Bezpośrednie symulowanie modułu IP Core w Vivado, na poziomie innym niż behawioralny, nie jest możliwe, stąd potrzeba „opakowania” go w dodatkowy plik (wrapper).
- Konfigurator do pamięci blokowej jest dość podobny, do tego dla pamięci rozprozonej. Wybieramy: *Memory Type – Single Port ROM*. W zakładce *Port A Options* ustawiamy odpowiednie parametry (takie jak poprzednio) oraz ustawiamy *Enable Port Type – Always Enabled*.
- **Uwaga.** Upewnij się, że moduł *main\_LUT.v* oznaczony jest jak *Top Module* dla projektu.
- utwórz nowy plik testowy Verilog *tb\_mainLut*, dodaj generowanie sygnału zegarowego (uwaga **okres 10 ns**) – zegar powinien być generowany w osobnej sekcji *initial*.
- dodaj instancję modułu *main\_LUT*.
- ustal też jakąś przykładową wartość wejściową – na początku 0, później jakąś wartość (to będzie potrzebna druga sekcja *initial*).
- uruchom symulację behawioralną.
- na przebiegach symulacji odszukaj miejsce, gdzie ustawiana jest wartość sygnału *addr*. 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 *dout* (wyjście z modułu LUT). Zmierz opóźnienie pomiędzy zboczem zegara, a pojawiением się wartości na wyjściu *dout* (pierwszym poprzedzającym). W tym celu:
  - powiększ wykres,
  - dodaj drugi marker (*Add Marker*),
  - pierwszy ustaw na narastającym zboczu zegara,
  - drugi ustaw w miejscu, gdzie zmienia się wartość *dout*,
  - zmierzone opóźnienie zanotuj.
- powtarzaj opisane wyżej czynności dla kolejnych modeli symulacyjnych: *post synthesis functional simulation*, *post synthesis timing simulation*, *post implementation functional simulation*, *post implementation timing simulation*.
- zastanów się/doczytaj z czego wynikają różnice w uzyskiwanych opóźnieniach.



## 9 — Segmentacja obszarów o kolorze skóry

### 9.1 Wprowadzenie

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

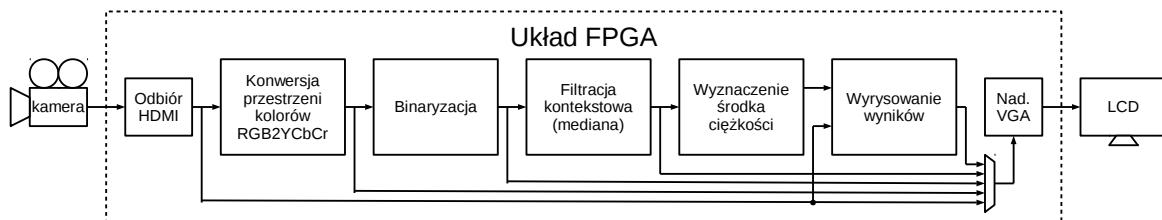
Warto w tym miejscu zaznaczyć, że nie jest to podstawowe podejście do problemu detekcji twarzy na obrazie. Najbardziej rozpowszechniona wydaje się być metoda zaproponowana przez Paula Viola i Michela Jonesa oparta o cechy Haara i algorytm uczący AdaBoost [?]. Dostępna jest ona np. w popularnej bibliotece do przetwarzania obrazów OpenCV ([opencv.org](http://opencv.org)) oraz w pakiecie Matlab. Nota bene, implementacja tego rozwiązania w układzie FPGA stanowi dość poważne wyzwanie. Tym niemniej, kolor może być informacją wspomagającą, która pozwala wykluczać tzw. fałszywe detekcje.

Schemat rozwiązania został przedstawiony na rysunku 9.1. W pierwszej kolejności, obraz z kamery jest odbierany po złączu HDMI i zamieniany na postać równoległą jak omówiono w rozdziale 8. Kolejny blok ma za zadanie dokonać konwersji przestrzeni barw z formatu RGB na YCbCr, w którym łatwiej wykrywać kolor skóry. Skóra ma specyficzną chrominancję (kolor) – wpadającą w czerwień. Właściwość ta jest zachowana (oczywiście w pewnym zakresie) dla przedstawicieli różnych ras. Dlatego też użycie przestrzeni, w której luminancja i chrominancja są odseparowane (np. YCbCr) pozwala na uzyskanie lepszych rezultatów, niż analiza w przestrzeni RGB.

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

monitora LCD na wyjście dowolnego z zaprojektowanych bloków.

W trakcie eksperymentów będziemy używać zielonego tła, co znaczco ułatwia proces segmentacji. Należy tutaj zwrócić uwagę na dwie kwestie. Po pierwsze, jest to bardzo uproszczona metoda określana jako *blue screen* lub *green screen*, a powszechnie stosowana w mediach (programy telewizyjne, czy filmy). Zainteresowane osoby odsyłamy do zasobów Internetu – warto wiedzieć w jak minimalnym stopniu we współczesnych filmach wykorzystuje się „rzeczywistą” scenografię, a jak bardzo nakładanie obrazów. Po drugie, powyżej przedstawiona została teoria na temat segmentacji na podstawie koloru skóry. Praktyka uczy jednak, że nie jest to takie prosta. Cześć osób wykonujących to ćwiczenie doświadczy tego – przykładowo zmiana warunków oświetlenia zewnętrznego (słoneczny, a pochmurny dzień) powoduje, że raz dobrane wartości progów „przestają działać”. Zatem zielone tło ułatwia sprawną realizację ćwiczenia, którego zadaniem jest nauczenie sprzętowej implementacji metod przetwarzania i analizy obrazów, a nie cierpliwości w dobieraniu progów.



Rysunek 9.1: Projektowany system wizyjny

Przybliżony plan pracy będzie zatem następujący:

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

## 9.2 Konwersja RGB do YCbCr – podstawy

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

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



Rysunek 9.2: Oryginalny obraz „lena” oraz poszczególne jego kanały w przestrzeni RGB oraz obraz w przestrzeni YCbCr i poszczególne kanały

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

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

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

**Uwaga.** W literaturze można również spotkać inne wersje tej konwersji (inne współczynniki) – por. <http://en.wikipedia.org/wiki/YCbCr>. Używane w ćwiczeniu wzory zgodne są ze standardem JPEG oraz biblioteką OpenCV. W stosunku do funkcji `rgb2ycbcr` z pakietu Matlab występują pewne różnice (niewielkie i nie mające wpływu na przebieg samego ćwiczenia).

### 9.3 Binaryzacja

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

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

$$maska = \begin{cases} 1 lub 255 & \text{jeśli } Ta < Cb < Tb \text{ i } Tc < Cr < Td \\ 0 & \text{w pozostałych przypadkach} \end{cases} \quad (9.2)$$

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

## 9.4 Filtracja

Uzyskana maska binarna zwykle zawiera szумy, czy to w postaci niewielkich grup pikseli wykrytych jako obszar skóry, czy też dziur wewnętrz „teoretycznie” jednolitych obszarów. Zatem zwykle pożądane jest przeprowadzenie filtracji. Uwaga. W 90 % algorytmów wizyjnych uzyskaną na jakimś etapie obliczeń maskę binarną poddaje się takiej operacji. Do wyboru są dwa podejścia: filtracja medianowa dla obrazów binarnych lub wybrana operacja morfologiczna. W rozważanym systemie zastosujemy medianę z oknem o rozmiarze  $5 \times 5$ , ale jak się później okaże przekształcenie modułu tak, aby realizował operację morfologiczną jest dość proste.

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

Z uwagi na specyfikę późniejszej implementacji sprzętowej warto przypomnieć sobie o problemie obsługi pikseli brzegowych (nie dla każdego piksela na obrazie możemy wyznaczyć kontekst  $5 \times 5$ ). Dla potrzeb niniejszego systemu (i wielu innych praktycznych rozwiązań) założymy, że jeśli nie dysponujemy pełnym kontekstem tj. rozpatrywany piksel leży blisko krawędzi obrazu to wyniki naszej filtracji wynosi '0'. W praktyce oznacza to, że „godzimy” na „czarną” ramkę o szerokości 2 pikseli. Przy rozdzielcości  $1280 \times 720$  nie ma ona większego znaczenia.

## 9.5 Wyznaczanie środka ciężkości

Na podstawie maski binarnej, system wizyjny nie ma możliwości „stwierdzenia” ile obiektów znajduje się na obrazie, albo jakie mają rozmiary. Informacja ta musi zostać wydobыта, poprzez przetworzenie maski przy pomocy algorytmów indeksacji i analizy grup połączony pikseli (ang. *connected component labelling*). W rozważanym przypadku zdecydowano się na jedną z najprostszych operacji analizy – wyznaczenie średniego położenia wszystkich pikseli maski należących do obiektu.

Warto w tym momencie zaznaczyć, że implementacja indeksacji w potokowym systemie wizyjnym nie jest sprawą trywialną. W przypadku tzw. indeksacji jednoprzebiegowej trzeba zrealizować moduł analizujący parametry poszczególnych obiektów i ich łączenie. Natomiast typowy algorytm dwuprzebiegowy wymaga użycia pamięci RAM (zwykle zewnętrznej) i wprowadza do systemu opóźnienie (latencję) powyżej jednej ramki. Oczywiście w obu przypadkach możliwa jest realizacja w pełni potokowa, ale zadanie należy uznać za trudne (i czasochłonne – czytaj niewykonalne w ramach laboratorium).

Do określania środka ciężkości pikseli należących do poszukiwanego obiektu, wykorzystywane są wzory bazujące na momentach geometrycznych:

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

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

oraz

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

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

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

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

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

## 9.6 Przykład działania

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



## 9.7 Zadanie do wykonania w domu

**Uwaga** Wykonanie niniejszego zadania jest **warunkiem dopuszczenia** do realizacji laboratorium.

### 9.7.1 Model programowy

**Zadanie 9.1** Proszę wykonać cyfrowe zdjęcie twarzy lub ręki. Obiekt powinien znajdować się w środku kadru i zajmować minimum 20% powierzchni obrazu. Dla ułatwienia, proszę zadbać o to, aby w kadrze nie było obiektów o kolorze zbliżonym do koloru skóry. Zdjęcie proszę przeskalać do rozdzielczości  $64 \times 64$ .

Proszę napisać skrypt w programie Matlab/Octave, który:

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

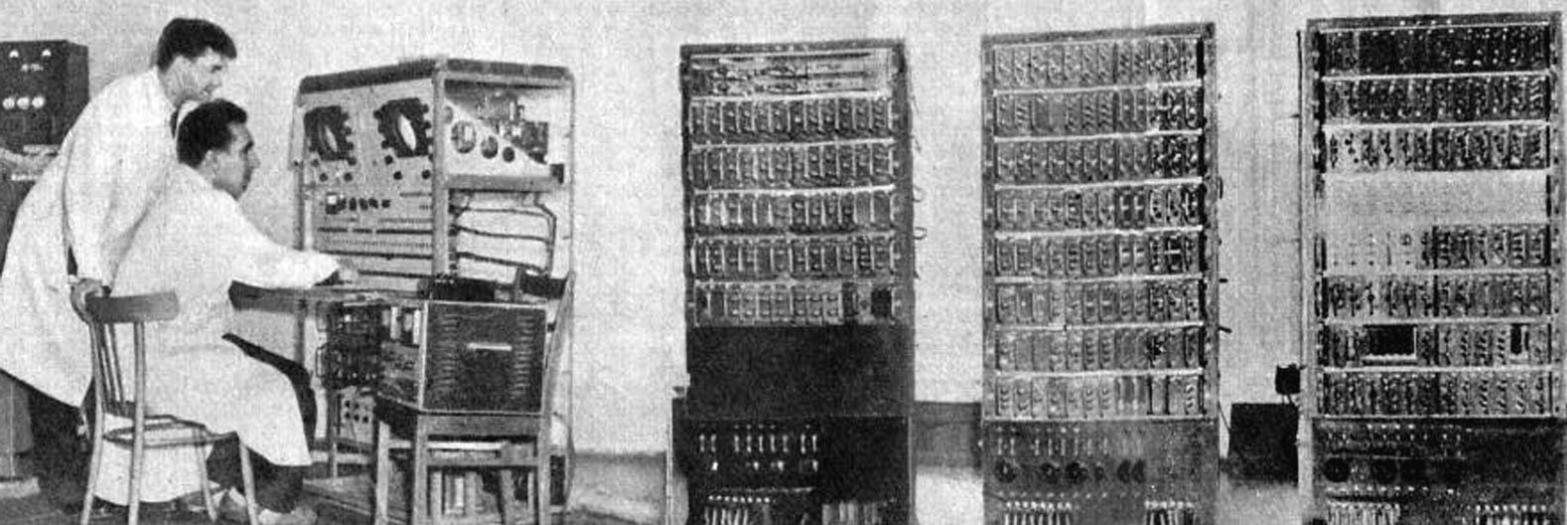
Zasadniczo efekt końcowy powinien być zbliżony do zaprezentowanego na rysunku 9.6. Proszę też przygotować wersję obrazu o rozdzielcości  $64 \times 64$  w formacie ppm. ■

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

P6LF  
64 64LF  
255LF

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

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



## 10 — Konwersja RGB do YCbCr

### 10.1 Model programowy

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

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

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

Wykorzystując metodologię wprowadzoną w zadaniu 7.6 stwórz stałoprzecinkowy model programowy. Ustal precyzję dla współczynników (dla mnożenia na 17+1 bitów), dla reszty (8+1 bitów – zakres [0:255]). Uwaga. Precyza 18 bitów ze znakiem dla współczynników wynika z maksymalnego rozmiaru słowa obsługiwanej przez moduł DSP48 w układzie Zynq (de-facto mnożarka wspiera obliczenia na argumentach 25 i 18). W praktyce raczej nie opłaca się tworzyć wielu ściśle dedykowanych modułów (o precyzyjnie dobranej szerokości argumentów). Znacznie utrudnia to ew. migrację na inny układ, gdyż trzeba przegenerować wiele modułów IP Core, a nie pozwala zaoszczędzić zasobów (mnożarka i tak będzie użyta). Należy jednak zaznaczyć, że powyższa uwaga nie dotyczy operacji implementowanych bezpośrednio w logice.

Uwaga. Zakładamy, że po operacji mnożenia **dokonujemy pominięcia części ułamkowej**. W Matlab'ie należy zastosować polecenie `quantize`. Uwaga! Do polecenia trzeba dodać argumenty – tj. podać oczekiwana precyzę. Kwantyzację należy przeprowadzić **osobno na każdym wyniku mnożenia**. Można to zrealizować np. za pomocą pętli `fors`.

Uwaga. Na liczbach w standardzie `fixed integer` można stosować operacje macierzowe – zatem istnieje możliwość stworzenia „zwartej” implementacji konwersji. Obsługiwane są funkcje (`fi`, `quantizeze`) oraz operatory arytmetyczne (również mnożenie macierzowe). Jednakże należy zwrócić uwagę na opisaną powyżej kwantyzację na wyjściu z każdego mnożenia. Zastosowanie jej po sumowaniu wyników (jak to się dzieje przy mnożeniu macierzy) może powodować drobne niezgodności pomiędzy wersją programową, a sprzętową.

Wybierz przykładowy wektor testowy RGB. Zrealizuj konwersję RGB->YCbCr na liczbach

*double*, a następnie stałoprzecinkowych. Wyniki porównaj. Z czego wynikają różnice?

## 10.2 Implementacja sprzętowa

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

Wskazówki:

1. Rozwijamy projekt zrealizowany w ramach poprzedniego ćwiczenia (tor wizyjny HDMI – por. rozdział 8).
2. Jednakże zaczniemy od stworzenia nowego projektu o nazwie *rgb2ycbcr* z modułem nadzorującym *rgb2ycbcr.v*. Jego wejścia to zegar, sygnały synchronizacji i piksel (RGB), a wyjścia to odpowiednio opóźnione sygnały synchronizacji oraz piksel YCbCr.
3. Następnie trzeba zaimplementować mnożenia. Realizujemy je z użyciem modułów DSP48 dostępnych w układzie Zynq (część FPGA). Należy wygenerować IP CORE typu *Multiplier*. W konfiguratorze ustawiamy:
  - *Input options* – rozmiar na 18 bitów i typ *Signed*,
  - *Multiplier Construction* – *Use Mults*,
  - *Pipeline Stages* – tak jak pokazuje *Optimum Pipeline Stages*.

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

Wstawiamy 9 instancji mnożarki. Uwaga. Wielkość liter w nazwach portów ma znaczenie! Do każdej mnożarki trzeba doprowadzić wartość R,G lub B. Ponieważ wejściem do modułu jest wektor 24 bitowy warto stworzyć pomocnicze 18-bitowe połączenia (R,G,B) typu *wire signed* (co oznacza, że wektory reprezentują liczby ze znakiem). Pomocna może być składnia *assign R = {10'b0, pixel[23:16]}*, czyli konkatenacja dwóch wektorów bitowych. Przypomnienie. Oba argumenty mnożenia są 18-bitowe. Drugi argument to stała. Najprościej ją uzyskać z modelu w pakiecie Matlab używając polecenia *bin(X)*, gdzie X to liczba stałoprzecinkowa (w odpowiedniej precyzyji). Wyjście z modułu należy zapisać do uprzednio stworzonych połączeń.

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

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

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

Uwaga. Warto rozważyć zaprojektowanie takiej samej struktury dla trzech składowych YCbCr, przy czym dla Y argument C będzie miał wartość 0.

5. W ostatnim kroku trzeba dodać opóźnienie sygnałów synchronizacji. Można ponownie

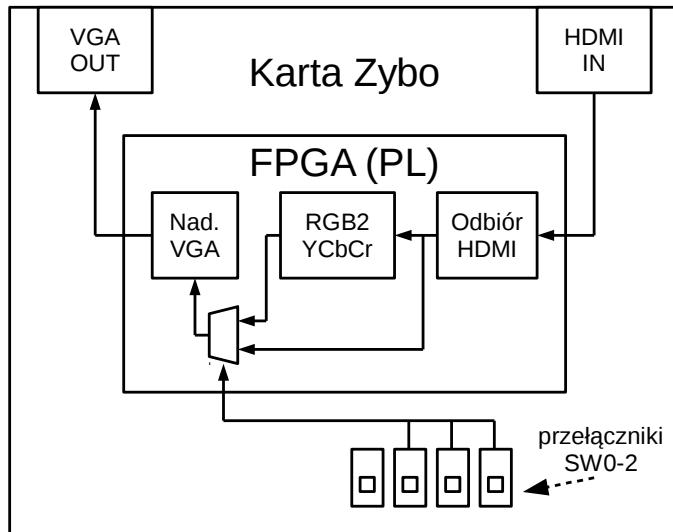
wykorzystać stworzony wcześniej moduł opóźniający. Trzeba tylko wyznaczyć globalną latencję modułu konwersji. Warto zastosować konkatenację sygnałów (wejściowych i wyjściowych) – {de,hsync,vsync}. Powoli to zastosować jeden, a nie trzy moduły i zaoszczędzić nieco pisania (kopiowania) kodu.

6. Po zbudowaniu modułu warto otworzyć jego schemat – *RTL Analysis -> Elaborated Design -> Schematic*. Jego analiza umożliwia eliminację części potencjalnych błędów.
7. Kolejny krok to prosty testbench. Po opisaniu modułu warto zrobić szybki test, dla tej samej wartości co w modelu programowym. Na tym etapie należy wyeliminować wszystkie błędy – wynik musi być identyczny.
8. Mając gotowy moduł możemy stworzyć z niego pakiet i dodać do pozostałą części projektu. Wybieramy *Tools->Create and Package New IP*. Dobrze jest wybrać jeden folder na moduły IP np. *ip\_repo*. Pakujemy cały projekt. Obecnie używane Vivado można zamknąć.
9. Otwieramy projekt *hdmi\_vga\_zybo*. Dodajemy w *IP Catalog* nowy folder lub odświeżamy (*Refresh All Repositories*), jeśli utworzyliśmy moduł we wcześniej używanym folderze – np. *ip\_repo*.
10. Symulacja toru wizyjnego. Wykonany moduł należy wstawić do modelu toru wizyjnego. Należy także dodać do folderu własny plik *ppm* i ustawić jego nazwę oraz ścieżkę w module *file\_input hdmi\_in.v*. Uwaga. Jeśli coś nie działa – np. występuje jakieś tajemnicze przesunięcie to proszę sprawdzić poprawność pliku *ppm* – tj. czy nie ma tam np. komentarza. Funkcja, która odczytuje obraz jest bardzo prosta (można sprawdzić) i sprowadza się do pominięcie 13 bajtów nagłówka oraz odczytu kolejnych pikseli.
11. Ostatnim etapem będzie porównanie wyników modelu programowego i symulacji dla całego obrazka. W tym celu należy zmodyfikować model stałoprzecinkowy (Matlab'owy) w taki sposób, aby można było przetworzyć obraz. Oczywiście ma to być własny plik *ppm* ze zdjęciem ręki lub twarzy.  
Zasadniczo, zrealizowane przetwarzanie należy obudować pętlą po wszystkich pikselach na obrazie. Następnie należy wczytać wyniki symulacji – plik *out\_*. Do porównania można użyć funkcji *imabsdiff* (suma z modułów różnic). Oczywiście oczekujemy, że nie będzie żadnych różnic.  
Uwaga. Obliczenia, nawet dla tak niewielkiego obrazka ( $64 \times 64$ ) mogą trwać chwilę. Jeśli będziemy mieli pewność, że wyniki modelu i symulacji Vivado są zgodne, to warto zapisać obraz w przestrzeni YCbCr z Matlab'a (funkcja *imwrite*). Wykorzystamy go później.  
Uwaga. Jeśli dla pojedynczych liczb (prosty test) wynik jest poprawny, a dla całego obrazu już nie, to prawdopodobną przyczyną błędów są kwestie związane z synchronizacją względem siebie poszczególnych wyników obliczeń.

### 10.3 Uruchomienie na karcie Zybo

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

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



Rysunek 10.1: Proponowany system wizyjny

**Zadanie 10.4** Uruchom zaprojektowany moduł na karcie Zybo. Dodaj multiplekser. ■

Podpowiedzi:

- Wstawienie modułu należy przeprowadzić analogicznie jak w przypadku LUT (por. rozdział ??).
- W projekcie *hdmi\_vga\_zybo* na schemacie zaznaczamy moduł *vp* i wybieramy opcję *Edit in IP Packager* (dostępna pod prawym przyciskiem myszy).
- Dodajemy do głównego pliku *vp* moduł *rgb2ycbcr* (podobnie jak przy symulacji). Na razie nie przypisujemy do niego sygnałów.
- Multiplekser jest trochę bardziej złożony. Na początku należy zdefiniować tablicę dla sygnałów R,G,B oraz synchronizacji:

```
wire [23:0] rgb_mux[7:0];
wire de_mux[7:0];
wire hsync_mux[7:0];
wire vsync_mux[7:0];
```

W ten sposób możemy np. do *r\_mux* przypisać 8 różnych wartości.

Następnie do wybranych pozycji w tablicy przypisujemy odpowiednie sygnały (składowe i synchronizację). W naszym przypadku mamy dwa (RGB (0) i YCbCr (1)). W ramach dalszych prac multiplekser rozbudujemy.

Zatem przykładowo stosujemy polecenie: `assign rgb_mux[0] = pixel_in;`

W tym momencie możemy już podpiąć nasz moduł *rgb2ycbcr*. Uwaga. Zarówno dla wejścia jak i wyjścia możemy stosować sygnały z multipleksera (to nawet wprowadza większy porządek do kodu).

Przy końcowym przepisaniu modułów wykorzystujemy stworzone tablice, przy czym wybór elementu uzależniamy od układu przełączników. Ponieważ maksymalnie zakładamy 8 możliwości to wystarczą 3 przełączniki.

Do interfejsu modułu należy dodać port wejściowy *sw* o rozmiarze 3 bitów].

- W celu eliminacji potencjalnych błędów zaleca się przeprowadzić syntezę modułu. Następnie przechodzimy na zakładkę *Package IP*. W oznaczonych miejscach w kolumnie

*Packaging Steps* musimy wybrać opcję *merge*. Na końcu dajemy *Re-Package IP* i polecamy zamknąć tymczasowe Vivado.

- Wracamy do projektu *hdmi\_vga\_zybo*. W górnej części ekranu, na żółtym tle, pojawi się informacja, że należy uaktualnić blok *vp* (po wprowadzaniu przez nas zmian). Wybieramy opcję *Show IP Status*. W dolnej części wyświetli się okno *IP Status*, w którym pokazane są moduły, które uległy modyfikacji. Wybieramy opcję *Upgrade Selected*. Możemy również dać opcję *re-run* i upewnić się, że wszystko jest poprawnie zaktualizowane.
- Proszę zwrócić uwagę, że na naszym module *vp* pojawił się nowy port *sw*. Należy dla niego wybrać opcję *Create Port* (prawy przycisk myszy). Należy również odkomentować stosowne linie w pliku *xdc*.
- Projekt należy, zaimplementować, uruchomić i sprawdzić na karcie Zybo.

## 10.4 Implementacja progowania

Następnym etapem jest progowanie obrazu w przestrzeni YCbCr – z wykorzystaniem tylko chrominancji. Powinno ono realizować następujące równanie:

$$P = \begin{cases} 255 & (Cb > T_a \&\& Cb < T_b \&\& Cr > T_c \&\& Cr < T_d) \\ 0 & \text{w p.p.} \end{cases}$$

Realizacja powyższego kodu w Verilog jest dość prosta:

```
assign bin = (Cb > Ta && Cb < Tb && Cr > Tc && Cr < Td) ? 8'd255
: 0;
```

Wartość *bin* należy przypisać do trzech składowych wyjściowych. Ponieważ „moduł” nie wprowadza opóźnienia, nie istnieje konieczność opóźniania synchronizacji.

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

Podpowiedzi:

- W modelu symulacyjnym należy wykorzystać zapisany poprzednio (rozdział 10.2) obraz YCbCr. Tylko w ten sposób będziemy mieli gwarancję, że uzyskana maska będzie identyczna zarówno w modelu symulacyjnym, jak i w sprzęcie. Jest to istotne w dalszych pracach, przy wyliczaniu środka ciężkości.
- Parametry (*Ta*, *Tb*, *Tc*, *Td*) należy ustawić „na sztywno” (jako parametry lokalne – *localparam*). Można sobie wyobrazić rozbudowę aplikacji o zmienianie tych parametrów podczas działania systemu (tj. on-line) przykładowo poprzez UART.
- Przygotowany moduł należy przetestować symulacyjnie. Warto sobie stworzyć „multiplekser z komentarzy” w module *tb\_hdmi*, czyli zakomentować, a nie kasować poprzednie przypisania (YCbCr) (sekcja *Output assignment*).
- Następnie moduł należy wstawić w tor przetwarzania obrazów (do modułu *vp* – *Edit in IP Packager*). Należy też podłączyć nowe wyjście do multipleksera.
- System należy zaimplementować i zweryfikować na karcie Zybo. Możliwe, że będzie potrzebne „dostrojenie” progów.

## 10.5 Zadania dodatkowe

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

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

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

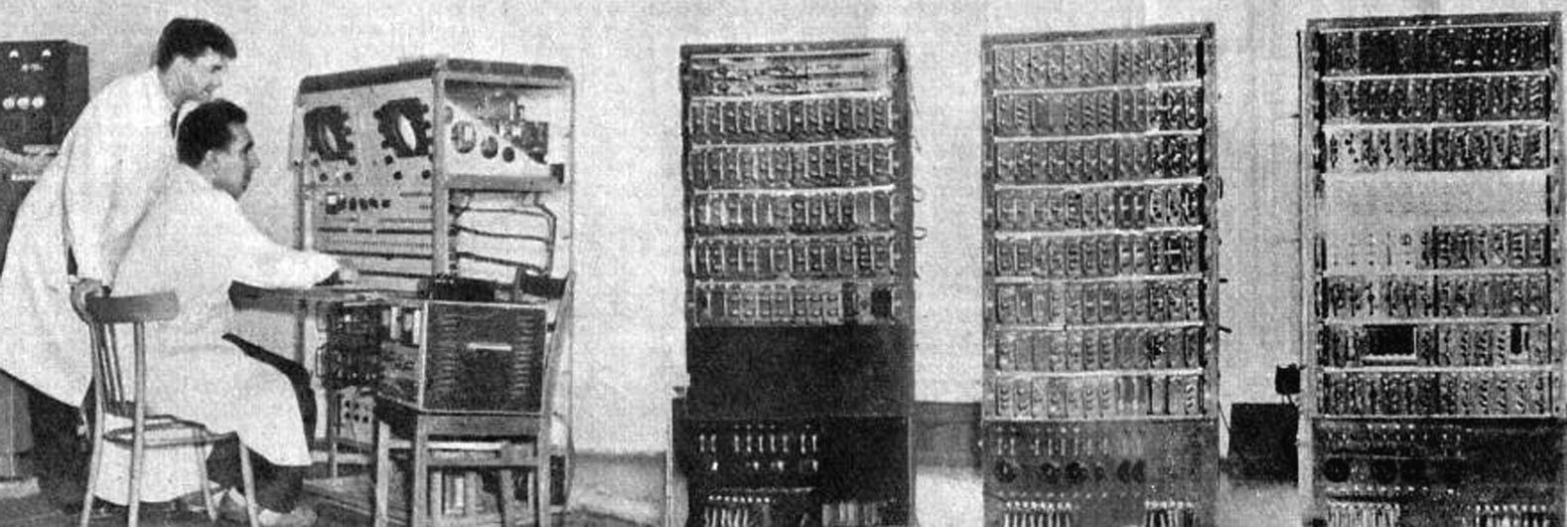
$$V = \max(R, G, B) \quad (10.1)$$

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

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

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

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



## 11 — Środek ciężkości

### 11.1 Wyznaczanie środka ciężkości

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

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

Wykorzystywane są trzy liczniki: kolumn, rzędów i końca ramki, sterowane na podstawie sygnałów synchronizacji (*de*, *hsync*, *vsync*). Pozwalają one określić współrzędne aktualnie rozważanego piksela na obrazie (*x,y*).

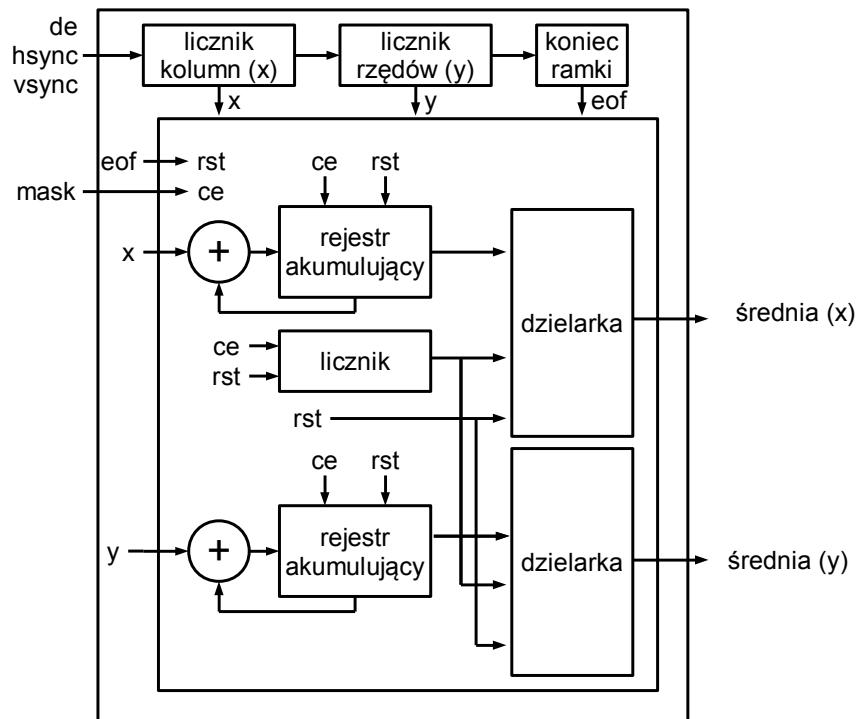
Wartości *x*, *y*, *eof* oraz maska binarna są wykorzystywane jako sygnały sterujące modułami wyznaczającymi współrzędne środka ciężkości. Wartość maski binarnej jest podpięta do wejścia *ce* w liczniku i rejestrze akumulującym. Dzięki temu, będą one pracowały jedynie w przypadku, gdy maska binarna ma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

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

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

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

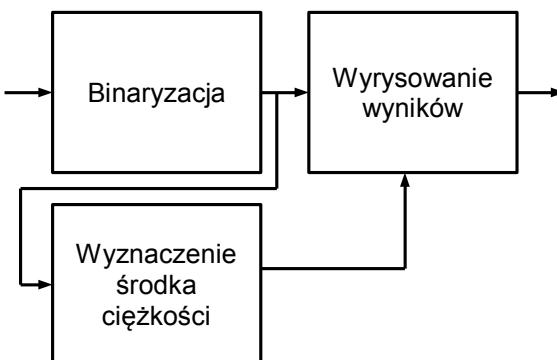
Warto się jeszcze zastanowić jak należy połączyć moduły wyznaczania środka ciężkości i wizualizacji. Prosta realizacja koncepcji przedstawionej na rysunku 10.1 ma pewną wadę.



Rysunek 11.1: Schemat modułu do wyznaczania środka ciężkości

Wyznaczanie środka ciężkości trwa w zasadzie całą ramkę – konieczne jest przetworzenie wszystkich pikseli obrazu zbinaryzowanego. Zachowanie potokowe wymagałoby opóźniania maski oraz sygnałów synchronizacji o niemal jedną ramkę. Jest to oczywiście możliwe (przy wykorzystaniu zewnętrznej pamięci RAM), ale niekonieczne sensowne i ekonomiczne.

Można za to zastosować pewne drobne „oszustwo” polegające na tym, że środek ciężkości wyznaczony dla poprzedniej ramki będzie wyświetlany na obrazie bieżącej. Przy częstotliwości 50/60 ramek na sekundę raczej nie będziemy w stanie zauważać tej drobnej rozbieżności. Koncepcję ilustruje rysunek 11.2.



Rysunek 11.2: Współpraca modułu obliczania środka ciężkości i wizualizacji.

## 11.2 Zadania do wykonania na laboratorium

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

Wskazówki:

- Otwieramy dotąd rozwijany projekt w Vivado (*hdmi\_vga\_zybo*). Moduł obliczania środka ciężkości dodamy wewnątrz modułu *vp*. Zatem klikamy na niego prawym przyciskiem myszy i wybieramy *Edit in IP Packager*. Uruchomi się drugie Vivado.
- Jednakże, aby stworzyć nowy moduł IP potrzebujemy tymczasowo trzeciego Vivado – tak jak działaliśmy poprzednio przy module *rgb2ycbcr*. Tworzymy nowy projekt – *centroid*, a także nowy plik źródłowy (*centroid.v*). Wejścia: *clk*, *ce*, *rst*, *de*, *hsync*, *vsync*, *mask* (bitowe). Wyjścia: współrzędna *x* i *y* środka ciężkości. Zakładamy, że maksymalna rozdzielcość przetwarzanego obrazu będzie wynosić  $1280 \times 720$ . Wykorzystując tę informację należy dobrać rozmiar wektora. Ponadto potrzebne są dwa parametry: *IMG\_H* i *IMG\_W* – odpowiednio wysokość (ang. *height*) i szerokość (ang. *width*) obrazu. Dodawanie parametrów zostało pokazane np. w zadaniu 5.1.
- Na początku stworzymy liczniki, które pozwolą określić aktualną pozycję na obrazie. Po pierwsze definiujemy niezbędne rejesty – np. *x\_pos* i *y\_pos* oraz je zerujemy. Wewnątrz procesu (*always*) sprawdzamy czy sygnał *vsync* wynosi 1 – jeśli tak to liczniki zerujemy, jeśli nie to sprawdzamy czy sygnał *de* (tj. poprawność piksela) jest równy 1. Jeśli tak to inkrementujemy licznik horyzontalny. Sprawdzamy czy nie osiągnął on wartości szerokości obrazu (*IMG\_W - 1* z uwagi na specyfikę działania przypisania  $\leq$ ). Jeśli tak to go zerujemy i inkrementujemy licznik wertykalny. Jeśli licznik wertykalny osiągnął wartość wysokości obrazu (*IMG\_H - 1* z uwagi na specyfikę działania przypisania  $\leq$ ) to go zerujemy.

Uwaga. Zaprezentowana została tylko jedna przykładowa możliwość implementacji. Równie dobrze można oprzeć się tylko i wyłączenie na przebiegach sygnałów synchronizacji – wtedy nie jest potrzebne określanie rozmiarów obrazu. Przy czym w tym przypadku należy pamiętać, że dla Zybo obowiązuje odwrotna logika dla sygnałów *hsync* i *vsync*.

- Wykrycie końca ramki (*eof – end of frame*) można zrealizować albo na podstawie powyższych liczników, albo na postawie sygnału *vsync*. W tym drugim przypadku należy wykryć jego narastające zbocze np. rejestrując poprzednią wartość (rejestrowanie ma odbywać się w każdym taktie zegara). Ponieważ potrzebujemy impulsu (sygnał trwający jeden takt) to możemy wykorzystać następującą składnię:

```
assign eof=(prev_vsync==0'b1&vsync==1'b1)?1'b1:1'b0;
```

- Następny element to obliczanie wartości *m00* – do wykonania za pomocą prostego licznika zerowanego sygnałem *eof* (*rst = eof*). Rozmiar rejestru należy określić na podstawie maksymalnej rozdzielcości obrazu.
- Implementacja obliczania momentów *m01* i *m10* opiera się o moduł akumulatora z zadania 7.3. Składa się on z sumatora o latencji 0 oraz procesu synchronizującego dodawanie. Szczegóły w opisie zadania. Jedyną modyfikacją jest rozmiar akumulatora. Należy się zastanowić jaki on powinien być. Szczególnie, jeśli chcemy wykorzystać moduł stworzony w ramach zadania 7.3. Można zastosować taki sam moduł dla obu momentów, gdyż oszczędność zasobów w przypadku realizacji o różnych szerokościach będzie niewspółmierna do potencjalnej oszczędności zasobów logicznych.
- Do dzielenia należy wykorzystać dostępny na stronie kursu moduł *divider\_32\_20*. Jest to implementacja dzielenia za pomocą kolejnych aproksymacji. Moduł składa się z mnożarki i maszyny stanowej. W skrócie jego działanie opiera się na następującej zasadzie. Na

początku wynik (*sar*) jest zerowany. Następnie, w kolejnych iteracjach (ich liczba zależy od długości dzielnika), na '1' ustawiany jest odpowiedni bit rejestru *sar* – zaczynając od najstarszego. Jest on mnożony przez dzielną (*divisor*), a rezultat operacji porównywany z dzielnikiem. Jeśli jest od niego mniejszy to '1' na tej pozycji zostaje. Jeśli nie, co oznacza, że proponowana liczba jest za duża, analizowany bit jest zerowany. Przechodząc, w ten sposób przez cały wektor *sar* uzyskuje się wynik dzielenia.

Niewątpliwą zaletą metody jest jej prostota i stosunkowo niewielkie zużycie zasobów logicznych. Wada to iteracyjność – jedno dzielenie wymaga: liczba bitów wyniku  $\times (2+4)$  taktów zegara, gdzie 4 oznacza latencję użytej mnożarki. Przy czym w naszym przypadku, kiedy dzielnie potrzebne jest raz na ramkę, takie podejście wydaje się być najlepszym wyborem.

Dzielarka domyślnie ma ustawioną szerokość dzielnej i wyniku na 32 bitów, a dzielnika na 20 bitów. Oczywiście można to zmienić, ew. trzeba tylko przegenerować IP Core. Jeśli nowy moduł miałby inną latencję niż 4, trzeba ustawić odpowiedni parametr. Jednakże dla potrzeb niniejszego ćwiczenia nic nie trzeba modyfikować.

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

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

Uwaga. Przy tworzeniu instancji modułu należy zadbać o to, aby ew. różnice w długości sygnałów uzupełnić zerami. W przeciwnym przypadku nie uda się uzyskać poprawnie działającej symulacji.

- Stworzony i spakowany moduł *centroid* należy dodać do modułu *vp*. Po odpowiednim podłączeniu i przeprowadzaniu „próbnej syntezы” należy moduł *vp* ponownie spakować (*Re-Package IP*) i przejść do projektu nadzawanego.
- Testowanie symulacyjne modułu jest dość proste. Po dodaniu go do pliku *tb\_hdmi* i odpowiednim podłączeniu, uruchamiamy symulację i monitorujemy sygnały *m00*, *m01*, *m10* oraz wyniki dzielenia. Powinny one być w 100 % zgodne z modelem programowym. Oczywiście przy założeniu, że mamy uzgodniony wynik po etapie binaryzacji. Uwaga. Należy odpowiednio ustawić parametry związane z rozdzielcością obrazu. Wykonuje się to poprzez dwukrotne kliknięcie na moduł *centroid* w hierarchii projektu. W symulacji używamy obrazów:  $64 \times 64$ , a na karcie  $1280 \times 720$ .
- Realizacja modułu wizualizacji (*vis\_centroid*) jest, wbrew pozorom, dość prosta. Należy zacząć od nowego projektu i stworzyć z niego IP Core. Podstawą są liczniki określające aktualną pozycję na ekranie. Należy wykorzystać stworzony wcześniej kod. Następnie sprawdzamy, czy położenie piksela odpowiada położeniu środka ciężkości i jeśli tak to „podmieniamy” kolor rozważanego piksela. Uwaga. Proszę pamiętać, że rysujemy dwie prostopadłe linie, a nie punkt – pojedynczy piksel byłby trudno widoczny. Przykładowy kod:

```
assign o_red= ( (x_cnt [9:0]==x || y_cnt [9:0]==y) ?8'hff:i_red);
```

- Moduł wizualizacji należy przetestować symulacyjnie (podłączyć do toru wizyjnego w pliku *tb\_hdmi*). Proszę pamiętać o sposobie jego podłączenia, w stosunku do modułu obliczania środka ciężkości.
- W ostatnim kroku należy oba moduły dodać do pliku *vp.v* i przetestować na karcie Zybo. Proszę pamiętać o ustawieniu parametrów *IMG\_W* i *IMG\_H* na odpowiednio 1280 i 720. Uwaga. W modułach *centroid* oraz *vis\_centroid*.

### 11.3 Zadania do wykonania w domu

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

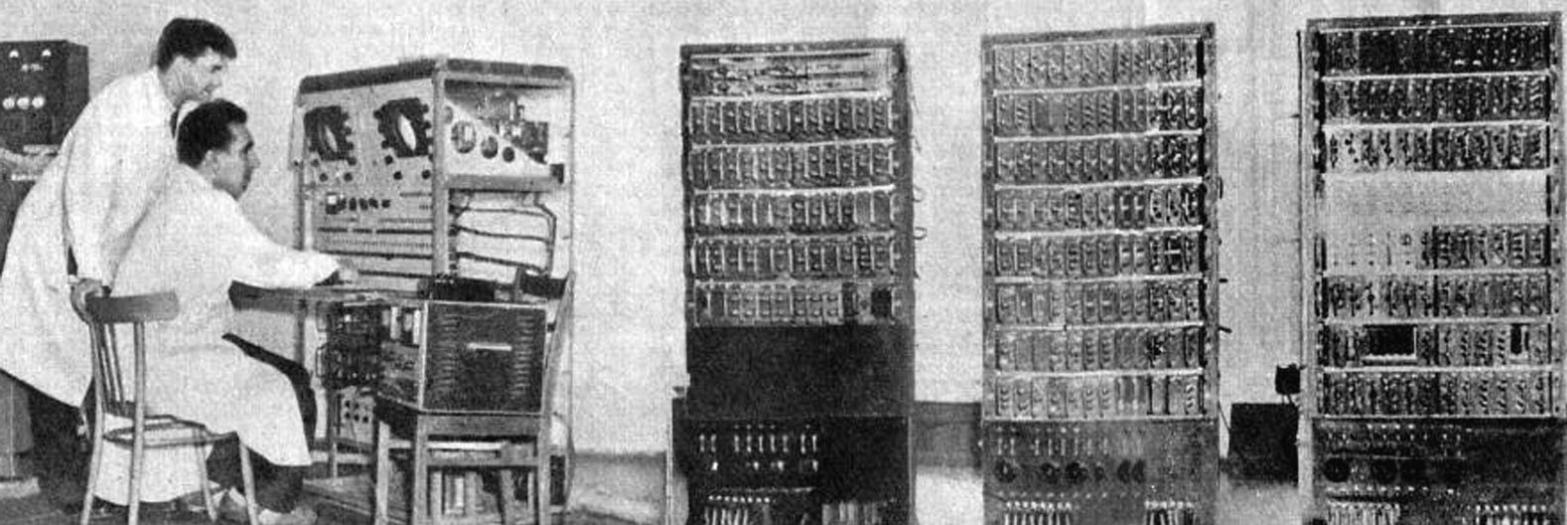
### 11.4 Zadania dodatkowe

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

Wskazówki:

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





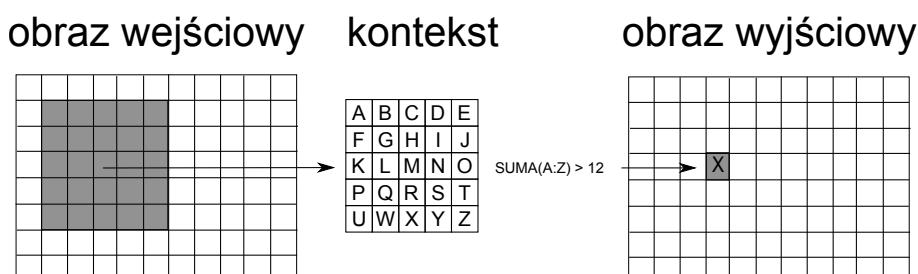
## 12 — Potokowa realizacja operacji kontekstowych

### 12.1 Koncepcja realizacji operacji kontekstowych w potokowym systemie wizyjnym

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

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

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



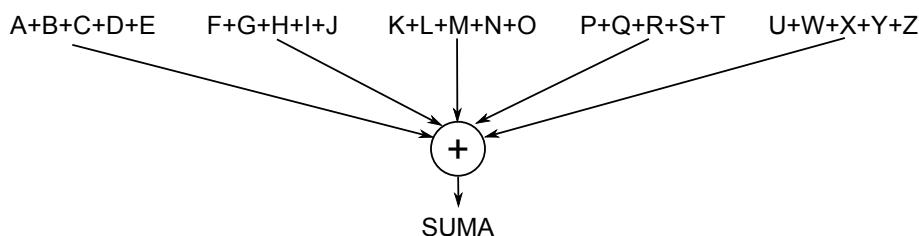
Rysunek 12.1: Koncepcja realizacji operacji medianowej z oknem  $5 \times 5$

Mamy zatem obraz wejściowy, z którego wycinamy kolejne konteksty o danym rozmiarze (w tym przypadku  $5 \times 5$ ). Warto zauważyć, że dla pikseli brzegowych nie da się wyznaczyć pełnego kontekstu (w rozważanym przypadku dokładnie dla obwódki o szerokości dwóch pikseli). W literaturze spotyka się wiele możliwości „obsługi” tego przypadku:

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

Należy zauważyć, że o ile implementacja każdej z tych metod programowo (Matlab, C++, Java) jest dość prosta, o tyle ich implementacja sprzętowa (i w dodatku potokowa) nastręcza pewnych trudności. Dla ułatwienia, w rozważanym projekcie wykorzystana zostanie metoda ostatnia, czyli piksele, które nie mają pełnego kontekstu, uzyskają na obrazie wynikowym wartość 0. Warto także zwrócić uwagę, że z praktycznego punktu widzenia obsługa pikseli brzegowych nie jest zbyt istotna, gdyż pominięcie filtracji ramki o szerokości 1 lub 2 piksele (odpowiednio filtr  $3 \times 3$  i  $5 \times 5$ ) ma zwykle marginalny wpływ na działanie całego systemu wizyjnego o rozdzielczości np.  $720 \times 576$ , czy też  $1280 \times 720$  pikseli.

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



Rysunek 12.2: Koncepcja realizacji drzewa sumacyjnego

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

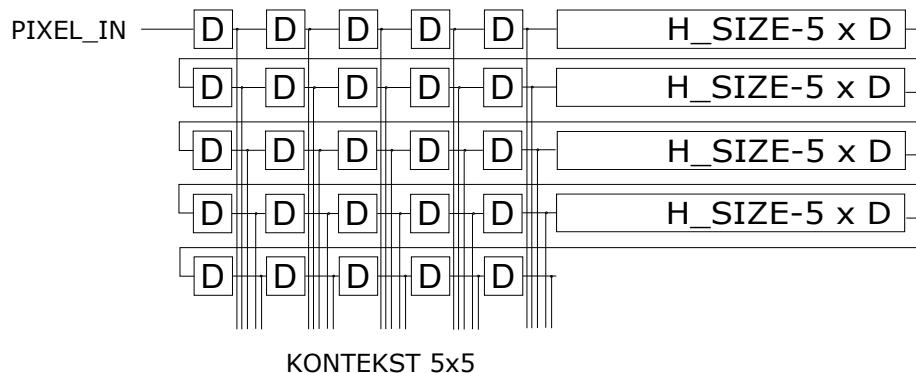
Otrzymaną sumę należy porównać z połową rozmiaru maski (12) i zwrócić wynik filtracji.

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

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

Zaleca się „wyobrażenie” sobie sposobu, w jaki przetwarzany jest pojedynczy piksel (jego „przepływ” przez schemat z rysunku 12.3) i dlaczego taki mechanizm ma prawo działać.

## 12.2 Zadania do wykonania na laboratorium



Rysunek 12.3: Schemat linii opóźniających dla kontekstu  $5 \times 5$

**Zadanie 12.1** Proszę zaimplementować i przetestować symulacyjne binarną filtrację medianową z maską o rozmiarze  $5 \times 5$ .

Wskazówki:

- Tradycyjnie zaczynamy od stworzenia nowego projektu – *median5x5* – później wykonamy na jego podstawie moduł. W kreatorze tworzymy plik *median5x5.v*. Ma mieć on typowy interfejs wizyjny, przy czym dla uproszczenia obliczeń zakładamy, że na wejście podajemy maskę bitową (w praktyce najmłodszy bit z maski uzyskanej z poprzedniego modułu), a na wyjściu wypisujemy już wektor 8-bitowy (z zakresu 0-255 – do wizualizacji). Moduł powinien posiadać jeden parametr – *H\_SIZE*, który opisuje szerokość obrazu (sekcja *parameter*). Jako domyślą wartość należy ustawić 83 – odpowiada to przetwarzaniu obrazu o rozdzielczości  $64 \times 64$  (dla potrzeb testowania symulacyjnego).
- W pierwszym kroku tworzymy moduł długiej linii opóźniającej. Dodaj nowy IP Core *IP Catalog -> Block Memory Generator*, nazwij go *delayLineBRAM* (**dokładnie tak**). Uwaga. Do realizacji linii opóźniających idealnie nadaje się pamięć blokowa (Block RAM) dostępna w układzie FPGA – zapewnia ona odpowiednią długość i szerokość oraz wspiera operacje FIFO (ang. *First In First Out*). Ustaw następujące opcje:
  - Interface Type -> Native,
  - Memory Type -> Single Port RAM,
  - Write Width -> 17 (wartość maksymalna dla pojedynczego modułu),
  - Write Depth -> 2048 (to gwarantuje przetwarzanie obrazu  $1280 \times 720$  z dostępnej kamery HDMI),
  - Operating Mode -> Read first,
  - Enable Port Type -> Always Enabled
  - zauważ ile bloków pamięci wykorzystujemy.

Dodaj do projektu moduł *delayLineBRAM\_WP* (dostępny na stronie kursu). Przeanalizuj jego implementację. Czy zgadzisz się ze stwierdzeniem, że działa on jak FIFO zbudowane na tablicy (tu module Block RAM)?

- Jak wiemy należy również zapewnić poprawne opóźnienie sygnałów synchronizacji: *de*, *hsync*, *vsync*. Najprostszą możliwością jest ich “doklejenie” do przetwarzanego piksela. Wykorzystamy to w realizowanym module. Zatem pojedyncza porcja danych będzie składać się z 4 bitów:  $\{mask, de, hsync, vsync\}$ .
- Trzeba teraz przygotować i opisać strukturę zgodną z rysunkiem 12.3: 25 krótkich linii opóźniających (w postaci 25 rejestrów o odpowiedniej długości) i cztery długie linie opóźniające. Wszystkie przypisania należy zrealizować w ramach jednego procesu. Uwagi:
  - skoro pojedyncza porcja danych składa się z 4-bitów to wymagane cztery linie

opóźniające możemy zrealizować z wykorzystaniem pojedynczego modułu BRAM. Trzeba tylko zadbać o odpowiednie przypisanie sygnałów wejściowych i wyjściowych. Użyteczny będzie operator konkatenacji – np. {x,y,z}.

- sygnał *rst* i *ce* w długiej linii opóźniającej należy ustawić na odpowiednio '0' i '1'.
- port *h\_size* modułu długiej linii opóźniającej należy ustawić na *H\_SIZE* - 5 – z uwagi na opóźnienie w pojedynczych rejestrach.
- szerokość danych dla modułu *delayLineBRAM\_WP* została ustawiona na 16 tj. 4 porcje danych o szerokości 4 bitów (odpowiada za to parametr *WIDTH*).
- do pierwszego z 25 rejestrów należy przypisać sygnały *mask* oraz *de*, *hsync*, *vsync*.
- należy stworzyć sygnał na dane wyjściowe z długiej linii opóźniającej i wykorzystać go do przypisania wewnątrz procesu (jak pamiętamy rejestr nie może być argumentem wyjściowym).
- kluczem do szybkiego sukcesu jest dobre nazewnictwo rejestrów – sugeruje się numerowanie wierszami i kolumnami np. *D11* itp.

- Realizacja drzewa sumacyjnego jest prosta. Należy tylko zdefiniować rejesty na sumy częściowe i ostateczną oraz dobrać ich rozmiary. Samo dodawanie można wykonać w ramach procesu (tego samego co opóźnienie) – nie jest potrzebny osobny IP Core.
- Żeby całość działała poprawnie należy jeszcze odpowiednio opóźnić sygnały synchronizacji dla piksela centralnego oraz flagę *context\_valid* (iloczyn sygnału *de* wszystkich pikseli kontekstu – np. *D11[2]* & *D12[2]* .... *D55[2]*, która określa, czy rozpatrywany kontekst jest poprawny). W przypadku dwupoziomowego drzewa sumacyjnego (latencja=2) opóźniamy dokładnie o 2. Uwaga. Dla operacji kontekstowych zwykle działamy w odniesieniu do piksela centralnego tj. środkowego dla danej maski. Do opóźnienia można zastosować używaną wcześniej linię opóźniającą zbudowaną na przerzutnikach (w odróżnieniu od długich linii opóźniających opartych o pamięć blokową).
- Na samym końcu należy wypisać wynik operacji tj. wartość piksela i trzy sygnały synchronizacji. Wykorzystujemy instrukcję *assing*. Uwaga. Na tym etapie należy także sprawdzić, czy uzyskana suma jest większa, czy mniejsza od połowy liczby pikseli w masce. Można to np. zrealizować na pomocą następującej instrukcji:  
`assign mask_new = sum > 5'd12 ? 255 : 0;`. Należy też sprawdzić poprawność kontekstu – składnia zbliżona. Uwaga. Należy zadbać, aby sygnały synchronizacji były zsynchronizowane z pikselem.
- Ostatnim krokiem będzie stworzenie nowego IP – *Create and Package IP*. Postępujemy podobnie jak przy wcześniejszych modułach. Uwaga. Wcześniej warto przeprowadzić próbную syntezę modułu, co powinno pozwolić na eliminację przynajmniej części potencjalnych błędów.
- Testowanie symulacyjne. Tradycyjnie jest to ważny element realizacji projektu. W przypadku tak „subtelnej” operacji jak mediana, ocena poprawności implementacji „na oko” nie jest najlepszym sposobem. Zatem wstawiamy stworzony moduł do pliku *tb\_hdmi* (w naszym głównym projekcie) – wcześniej dodając go repozytorium. Jeśli mamy pewność, że nasz model jest 100 % zgodny z FPGA to po prostu sprawdzamy wyniki. Powinny one być zgodne z funkcją *medfilt2* pakietu Matlab z dokładnością do krawędzi. Można je „wyzerować”:  
`m(1:2,:) = 0; m(63:64,:) = 0; m(:,1:2) = 0; m(:,63:64) = 0;` Wtedy powinniśmytrzymać pełną zgodność. Inną opcją jest zapisanie wyniku binaryzacji z modelu symulacyjnego. Stosując go jako wejście do modelu programowego możemy sprawdzić samą medianę.
- Testowanie w sprzecie. Stworzony moduł należy dodać do modułu *vp* i odpowiednio podłączyć. Bardzo ważne jest ustawienie parametru ***H\_SIZE* na 1664**. Przypominam, że

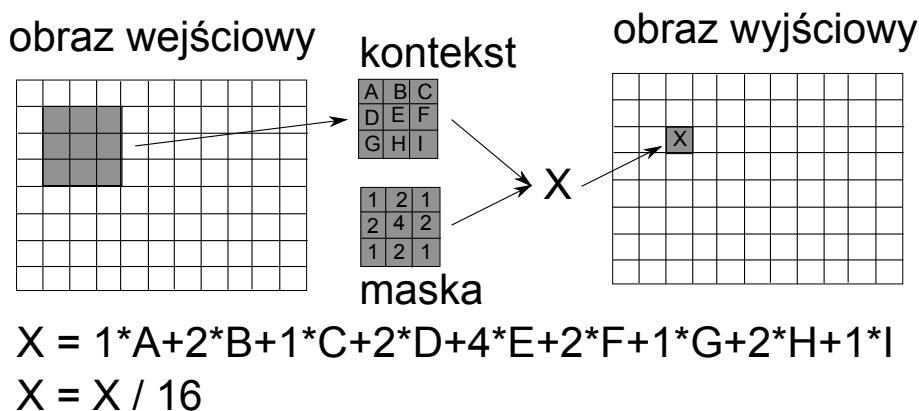
trzeba to zrobić poprzez polecenie *Re-Customize IP* .... Wartość wynika oczywiście ze specyfiki sygnału HDMI i rozdzielczości sygnału z kamery:  $1280 \times 720 @ 60$  fps (Uwaga. Np. dla 50 fps, tj. sygnału z typowej europejskiej kamery, wartość ta będzie już inna).

### 12.3 Zadania do wykonania w domu

**Zadanie 12.2** Wykorzystując zaimplementowany moduł zrealizuj moduł morfologicznego otwarcia lub zamknięcia (czyli kombinacji erozji i dylatacji). Rozwiążanie przetestuj symulacyjnie i w sprzęcie. ■

### 12.4 Zadania dodatkowe

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



Rysunek 12.4: Koncepcja realizacji filtracji uśredniającej  $3 \times 3$

Wskazówki:

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

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

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

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

## 13 — Procesor w układzie FPGA

### 13.1 Czym jest procesor

Główną różnicą pomiędzy układami reprogramowalnymi FPGA a układami programowalnymi (mikroprocesory czy CPU) jest to, że w tych pierwszych architektura przetwarzająca dane może być dowolnie zmieniana (rekonfigurowana). W procesorach już na etapie projektowania określa się liczbę rejestrów, szerokość magistrali danych oraz operacje które będą realizowane w jednostce arytmetyczno-logicznej (ALU). W etapie produkcji bloki te zostają zrealizowane na stałe w krzemie jako grupy połączonych ze sobą bramek logicznych. W związku z tym, istnieje określona, z góry ograniczona liczba podstawowych instrukcji, które mogą być wykonywane. Użytkownik nie ma możliwości zmiany tej architektury, może natomiast zmieniać program, który będzie na niej wykonywany. To od sekwencji instrukcji dostarczonych przez programistę zależy, które dane są pobierane z pamięci do rejestrów, które operacje na tych danych są wykonywane w jednostce arytmetyczno-logicznej i gdzie zapisywane są wyniki. Program jest pewną sekwencją instrukcji, która w zależności od ich kolejności umożliwia rozwiązywanie różnych problemów obliczeniowych.

Skoro jednak w układzie FPGA można realizować dowolne architektury sprzętowe, możliwe jest także połączenie zasobów rekonfigurowalnych w ten sposób, aby realizowały one funkcjonalności procesora. Takie podejście, w którym procesor jest zrealizowany nie w postaci na stałe ustalonych połączeń w krzemie, ale w postaci odpowiednio skonfigurowanych zasobów układu FPGA jest określone mianem soft-procesora.

Rozwiązywanie to ma wiele zastosowań praktycznych. Jest wykorzystywane przez producentów procesorów do testowania i prototypowania nowych architektur. W przypadku rozwijania urządzeń typu *embedded* (tj. wbudowanych), pozwala na umieszczenie mikrokontrolera i układu FPGA w jednej obudowie, co prowadzi do oszczędności miejsca na płytce PCB oraz kosztów produkcji (przykładem takie architektury, zwanej *System on Chip*, są układy Zynq firmy Xilinx). Soft-procesory są również wykorzystywane w przypadku, gdy część algorytmu jest typowo sekwencyjna i jej realizacja w formie „czysto sprzętowej” wymagałyby projektowania skomplikowanych maszyn stanów. Warto w tym miejscu wspomnieć, że obaj wiodący producenci układów FPGA Xilinx i Intel (Altera) dostarczają własne moduły soft-procesorów. Dla Xilinx’a są to MicroBlaze i PicoBlaze, a dla Intel Nios.

Główną wadą soft-procesorów jest relatywnie niska częstotliwość pracy, w porównaniu do procesorów zrealizowanych bezpośrednio w krzemie. Wynika to bezpośrednio z uniwersal-

nego charakteru układów reprogramowalnych i jest związane z faktem, że zarówno elementy połączeniowe jak i konfigurowalne elementy obliczeniowe nie mogą działać tak samo szybko jak dedykowane, wysoce zoptymalizowane i wykonane w krzemie stałe bramki i połączenia. Dla najnowszych dostępnych obecnie układów reprogramowalnych maksymalna częstotliwość wynosi teoretycznie około 740 MHz (Virtex 7) podczas gdy procesory osiągają częstotliwości powyżej 3 GHz (niektóre nawet 4 GHz).

Na niniejszych laboratoriach zaprojektujemy i wykonamy soft-procesor w układzie FPGA.

### 13.2 Instrukcje, asembler, kod maszynowy

Pisząc program komputerowy w języku wysokiego poziomu najczęściej nie zastanawiamy się, jak dokładnie jest on wykonywany przez procesor. Wywołanie prostej funkcji wypisującej tekst na ekranie w trybie tekstowym np. `printf`, wymaga wykonania setek instrukcji. Program użytkownika wysyła odpowiedni tekst do warstwy systemu odpowiedzialnej za komunikację modułami wejścia/wyjścia (I/O), a następnie sterownik karty graficznej musi wyrysować go w odpowiednim miejscu w konsoli użytkownika.

Wiemy, że instrukcje wysokiego poziomu są zamieniane na instrukcje w języku asembler. Język maszynowy odpowiada prostym operacjom realizowanym przez daną architekturę procesora. Są to operacje podstawowe, takie jak pobranie czy zapisanie danych do odpowiednich komórek pamięci, wykonywanie operacji arytmetyczno-logicznych i zapisy danych do rejestrów. Jednak program w języku asembler (w postaci tekstuowej) również musi zostać skompilowany do postaci kodu maszynowego. Kod maszynowy to binarnie zapisane instrukcje, które sterują poszczególnymi częściami architektury w ten sposób, aby dane były przetwarzane zgodnie z intencją programisty.

### 13.3 Opis proponowanego procesora

Na laboratorium wykonamy procesor, którego architektura została zaprezentowana na rysunku 13.1. Posiada on pamięć danych i rozkazów, zestaw 8 rejestrów do przechowywania danych oraz jednostkę arytmetyczno-logiczną. Dodatkowo 7 multiplekserów jest wykorzystywanych do odpowiedniego sterowania przepływem danych pomiędzy tymi elementami. W pamięci rozkazów przechowywane są 32-bitowe instrukcje. Poszczególne bity są wykorzystywane jako instrukcje sterujące (oznaczone jako `_op`) oraz wartość bezpośrednią (`immediate – im`), co zostało przedstawione na rysunku 13.1.

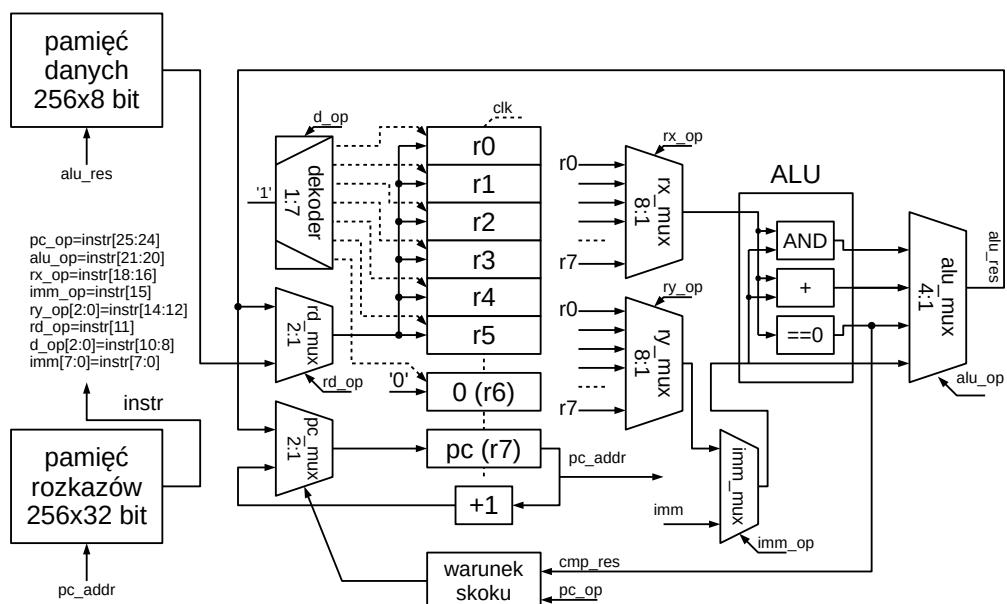
Bank 8-bitowych rejestrów składa się z 6 rejestrów ogólnego przeznaczenia (R0–R5). Rejestr R6 jest rejestrem specjalnym, który na stałe ma wpisaną wartość 0. Rejestr R7 jest tak zwany licznikiem rozkazów (PC – *program counter*) i jego wartość jest wykorzystywana jako adres, spod którego następuje odczytanie następnej instrukcji. W normalnym trybie, jest on inkrementowany o 1, jeśli jednak podano instrukcję skoku, multiplekser (oznaczony jako `pc_mux`) jest wykorzystywany do zapisania innej wartości adresu. Wybór trybu jest dokonywany przez blok nazwany `warunek skoku`. W zaprojektowanym procesorze, instrukcja skoku (`jump`) może odbywać się na trzy sposoby (sterowanie przy pomocy `pc_op`):

- `jump` – skok bezwarunkowy do adresu danego przez wartość na linii `alu_res`,
- `jnz` – skok do adresu danego przez wartość na linii `alu_res`, w przypadku gdy wartość porównywana na komparatorze nie wynosi零 (*jump if not zero*),
- `jz` – skok do adresu danego przez wartość na linii `alu_res`, w przypadku gdy wartość porównywana na komparatorze wynosi零 (*jump if zero*),

Instrukcje warunkowe pozwalają na realizację takich operacji jak *if* oraz *case*. Instrukcje bezwarunkowe odpowiadają wywołaniu funkcji.

Zapis do rejestrów R0–R5 jest sterowany poprzez dekoder, natomiast zapis do rejestru R7 (PC) odbywa się w każdym taktie zegara. Wszystkie elementy oprócz banku rejestrów pracują w trybie asynchronicznym (bez zegara – z zerową latencją).

Jednostka arytmetyczno-logiczna może wykonać trzy operacje. Dodawanie dwóch liczb 8-bitowych, operację logiczną AND oraz przyrównanie jednej z wartości do 0. Dodatkowo, multipleksery alu\_res zawierają czwarte wejście – bezpośrednie przypisanie wyjścia z multipleksera imm\_mux. Wszystkie bloki działają z zerową latencją.



Rysunek 13.1: Architektura procesora

### 13.4 Przykład realizacji instrukcji mov i movi

W niniejszym rozdziale zostanie przedstawiona metodologia translacji instrukcji asemblera na kod maszynowy. Wykorzystany będzie przykład jednej z podstawowych instrukcji w każdym procesorze, która umożliwia przeniesienie lub zapisanie wartości do odpowiedniego rejestru. Instrukcje te najczęściej oznacza się **mov** (ang. *move*) i **movi** (ang. *move immediate*).

W języku wysokiego poziomu (np. C lub C++) odpowiadają jej instrukcje przypisania:

**Kod 13.4.1 — ustawienie wartości zmiennej:**

```
int main()
{
    int x=1;
    ...
}
```

**Kod 13.4.2 — przepisanie wartości zmiennej:**

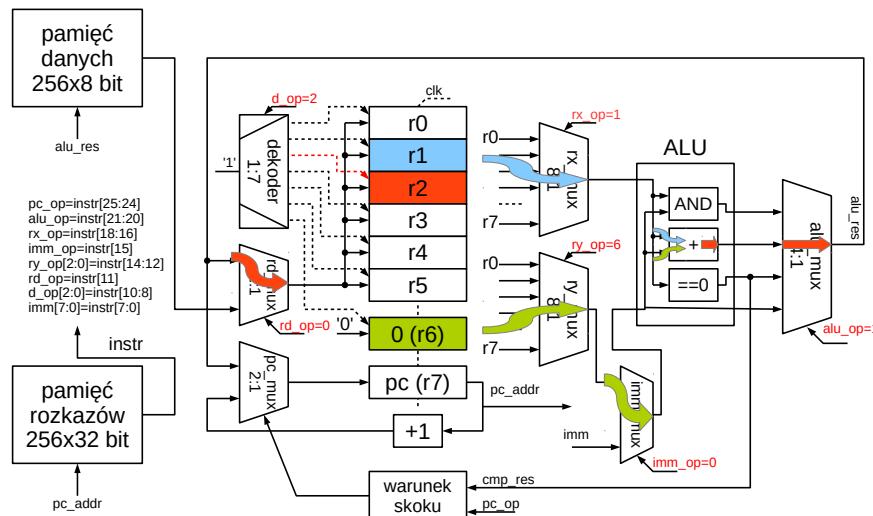
```
int main()
{
    int x=1;
    int y;
    y=x;
}
```

(przy czym zakłada się, że zmienne są przechowywane w rejestrach, a nie w pamięci oraz że wartość jest inicjowana bezpośrednio, a nie z pamięci danych).

W języku asembler instrukcję przepisania wartości rejestru oznacza się następująco:  
**mov Rd, Rx** – gdzie Rd jest rejestrem do którego ma nastąpić zapis, a Rx rejestrem źródłowym,

np. **mov R2, R1** – spowoduje przepisanie wartości z rejestru R1 do rejestru R2.

Zastanówmy się teraz, jak należy skonfigurować multipleksery (instrukcje sterujące `_op`), aby uzyskać odpowiedni przepływ danych przez architekturę zaprojektowanego procesora (rysunek 13.1). Jedno z możliwych rozwiązań zostało zaprezentowane na rysunku 13.2. Ponieważ, w proponowanej architekturze, nie ma możliwości bezpośredniego połączenia wyjścia rejestru R1 z wejściem rejestru R2, konieczne jest użycie ALU. Aby operacja nie zmieniała danych, wykorzystana zostanie wartość 0 dostępna w rejestrze R6. W tym celu odpowiednio skonfigurowano multipleksery `rx_mux` i `ry_mux`. Obie wartości są podawane do jednostki arytmetyczno-logicznej (ALU), gdzie jest wykonywane sumowanie wartości z rejestru R1 z 0 (z rejestru R6). Dzięki temu na wyjściu z ALU uzyskujemy wartość z rejestru R1. Wartość z sumatora jest ustawiana na linię `alu_res` dzięki odpowiedniej konfiguracji multipleksera `alu_mux`. Następnie jest ona podawana na wejście rejestrów dzięki multiplekserowi `rd_mux`. Dekoder ustawiwe wejście `ce` rejestrów R2 w stan wysoki, co powoduje, że w następnym taktie zegara w rejestrze R2 zostanie zapisana żądana wartość.

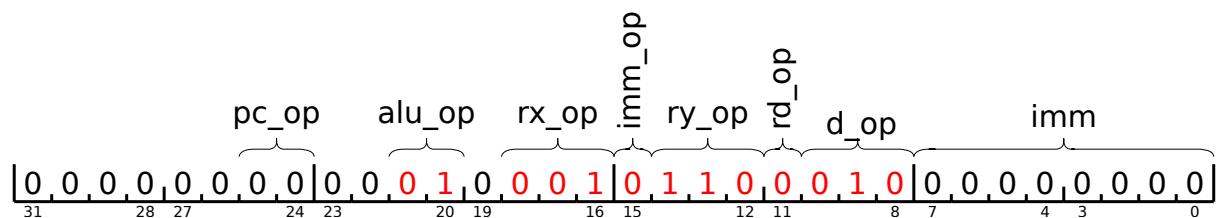


Rysunek 13.2: Wykonanie instrukcji **mov R2, R1**

Jeśli wstawimy odpowiednie wartości instrukcji sterujących do formatu instrukcji (rysunek 13.4), to okaże się, że instrukcji:

**mov R2, R1** odpowiada kod maszynowy 0x00116200,  
można również zauważyc, że dla dowolnych Rd i Rx:

**mov Rd, Rx** – odpowiedni kod maszynowy będzie przyjmował wartość 0x{001, Rx, 6, Rd, 00}.



Rysunek 13.3: Format instrukcji **mov R2, R1**

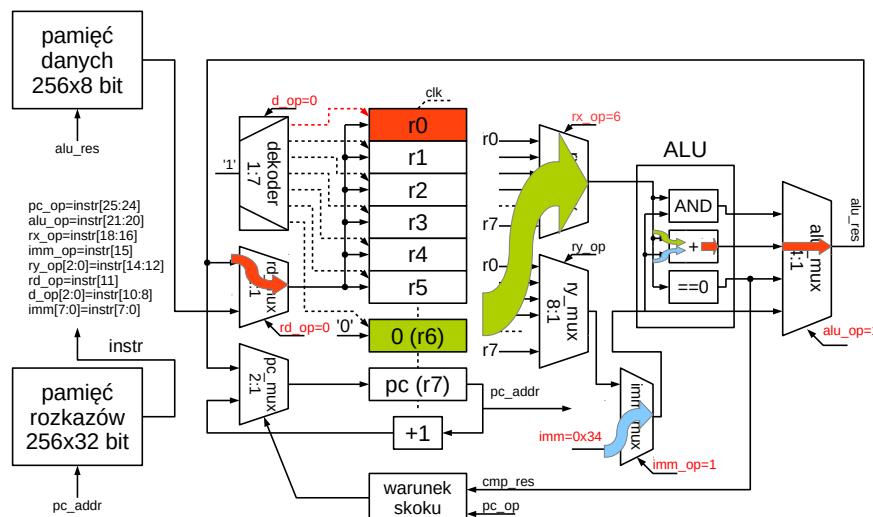
W języku asembler instrukcję ustawienia wartości bezpośredniej w rejestrze oznacza się następująco:

**movi Rd, imm** – gdzie Rd jest rejestrem do którego ma nastąpić zapis, a imm wartością, która ma być do niego wpisana,

np. **mov R0, 0x34** – spowoduje zapisanie wartości 0x34 (decymalnie 52) do rejestrów R0.

Zastanówmy się teraz, jak należy skonfigurować multipleksery (instrukcje sterujące \_op), aby uzyskać odpowiedni przepływ danych przez architekturę zaprojektowanego procesora (rysunek 13.1). Jedno z możliwych rozwiązań zostało zaprezentowane na rysunku 13.4. Ponieważ nie ma w niej możliwości bezpośredniego połączenia wyjścia imm z wejściem rejestrów R0, konieczne jest ponowne wykorzystanie wartości 0 z rejestrów R6. W tym celu odpowiednio skonfigurowano multiplekser rx\_mux. Wartość bezpośrednią jest podawana na drugie wejście jednostki arytmetyczno-logicznej (ALU) dzięki multiplekserowi imm\_op, gdzie jest wykonywane sumowanie wartości z rejestrów 0 (z rejestrów R6) z wartością bezpośrednią imm. Wartość z sumatora jest ustawiana na linię alu\_res dzięki odpowiedniej konfiguracji multipleksera alu\_mux. Następnie jest ona podawana na wejścia rejestrów dzięki multiplekserowi rd\_mux. Dekoder ustawi wejście ce rejestrów R0 w stan wysoki, co powoduje, że w następnym taktie zegara w rejestrze R0 zostanie zapisana żądana wartość.

Alternatywnym rozwiązań może być wykorzystanie bezpośredniego wyjścia z imm\_mux dostępnego z alu\_mux.



Rysunek 13.4: Wykonanie instrukcji **movi R0, 0x34**

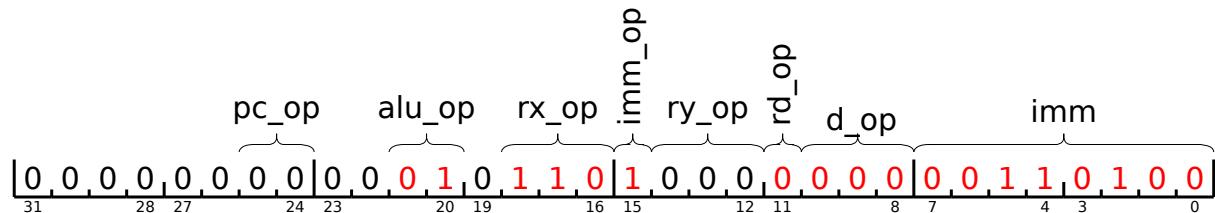
Jeśli wstawimy odpowiednie wartości instrukcji sterujących do formatu instrukcji (rysunek 13.4), to okaże się, że instrukcji:

**movi R0, 0x34** odpowiada kod maszynowy 0x00168034,

można również zauważyc, że dla dowolnych Rd i imm:

**movi Rd, imm** odpowiedni kod maszynowy będzie przyjmował wartość  $0x\{00168, Rd, imm\}$  – przy czym imm musi być wyrażone jako bajt np. 0x02.

W zadaniu 13.3 zaprezentowano więcej możliwych instrukcji języka asembler, które mogą być wykonywane na zaproponowanym procesorze i umożliwiają napisanie kompletnego programu.

Rysunek 13.5: Format instrukcji **movi R0, 0x34**

### 13.5 Zadania do wykonania na laboratorium

**Zadanie 13.1** Proszę opisać w języku Verilog procesor przedstawiony na schemacie 13.1. Moduły pamięci należy pobrać ze strony kursu (proszę je dodać do projektu). Cały procesor powinien zostać umieszczony w module, który posiada tylko jedno wejście: `clk`.

Podpowiedzi:

- zaczynamy od nowego projektu w Vivado,
- wbrew pozorom zadanie nie jest trudne. Wymaga jedynie nieco cierpliwości kończąc kurs z projektowania logiki FPGA należy zauważać, że cierpliwość to podstawowa cecha charakteru, niezbędna do realizacji większych projektów. Trzeba stworzyć wszystkie multipleksery, moduł ALU oraz rejestr, a także dodać pamięci,
- można, ale nie trzeba, realizować wszystko w osobnych modułach – proszę pamiętać, że w jednym pliku Verilog można zdefiniować kilka modułów.

**Zadanie 13.2** Proszę napisać moduł testowy, który będzie zawierał moduł procesora i generował sygnał zegarowy – `clk`. W tym celu można wykorzystać poniższy kod:

**Kod 13.5.1 — Wymuszenie zegara w module testowym:**

```
reg clk=1'b0;

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

W ramach pierwszych testów proszę zrealizować opisane powyżej dwie instrukcje **mov** i **movi**. Po pierwsze, proszę wpisać jakąś wartość do rejestru **R0**, a później przepisać ją do **R1**. Uwaga. Instrukcje możliwe zapisywać binarnie lub heksadecymalnie – w zależności od tego, co dla kogo jest bardziej czytelne.

**Zadanie 13.3** Proszę zaproponować kod maszynowy, który wymusi ustawienie odpowiednich multiplekserów i rejestrów w ten sposób, aby możliwe było wykonanie następujących instrukcji. Każdy przypadek proszę przetestować dla co najmniej dwóch argumentów wejściowych. Uwaga. Proszę dla każdego przypadku testowego zapisać sobie realizowaną instrukcję (przykład **mov R1, R0**). Będzie to później bardzo pomocne w realizacji zadania

domowego.

- **nop**  
pusty przebieg, stan rejestrów (R0–R6) nie jest zmieniany, pobierana jest kolejna instrukcja,
- **jump Rx**  
wartość licznika instrukcji (PC) jest ustawiana na wartość z rejestru Rx, kolejna instrukcja jest pobierana z lokacji o adresie Rx,
- **jumpi imm**  
wartość licznika instrukcji (PC) jest ustawiana na wartość bezpośrednią imm, kolejna instrukcja jest pobierana z lokacji o adresie imm,
- **jz Rx, imm**  
w przypadku, gdy register Rx ma wartość 0, wartość licznika instrukcji (PC) jest ustawiana na wartość bezpośrednią imm,
- **jnz Rx, imm**  
w przypadku, gdy register Rx ma wartość inną niż 0, wartość licznika instrukcji (PC) jest ustawiana na wartość bezpośrednią imm,
- **add Rd, Rx, Ry**  
wynik sumowania wartości z rejestrów Rx i Ry jest zapisywany do rejestru Rd,
- **addi Rd, Rx, imm**  
wynik sumowania wartości z rejestru Rx i wartości bezpośredniej imm jest zapisywany do rejestru Rd,
- **and Rd, Rx, Ry**  
wynik operacji logicznej AND wartości z rejestrów Rx i Ry jest zapisywany do rejestru Rd,
- **andi Rd, Rx, imm**  
wynik operacji logicznej AND wartości z rejestru Rx i wartości bezpośredniej imm jest zapisywany do rejestru Rd,
- **load Rd, Rx**  
załadowanie do rejestru Rd wartości z pamięci danych spod adresu o wartości z rejestru Rx,
- **loadi Rd, imm**  
załadowanie do rejestru Rd wartości z pamięci danych spod adresu o wartości bezpośredniej imm.

## 13.6 Zadania do wykonania w domu

**Zadanie 13.4** Napisać skrypt w programie Matlab/Octave (lub w czym kto lubi), który umożliwi przekształcenie instrukcji asemblera z zadania 13.3 do postaci kodu maszynowego.  
np. **mov R0, R2** → 0x00116200  
lub **mov R0, R3** → 0x00116300

Kod asemblera ma być pobierany z pliku tekstowego (np. *program.asm*) i parsowany linia po linii. Wynik ma być zapisywany do pliku tekstowego (np. *program.mc*).

Porada 1: Do parsowania tekstu w programie Matlab przydatna jest funkcja `strtok`, podanie jako delimiter ' ' (spacja i przecinek) powoduje usunięcie tych znaków z tekstu i zwrot kolejnych tokenów (więcej szczegółów w dokumentacji funkcji).

Porada 2: Do porównywania tokenów zaleca się stosować funkcję `strcmp`.

Porada 3: Do wczytywania linii z pliku tekstowego najlepiej nadają się funkcje: `fopen`, `fgets`, `fclose`.

Porada 4: Do konwersji z liczby dziesiętnej na binarną, albo hesksadecymalną można wykorzystać funkcje: `dec2bin` i `dec2hex`.

Uwaga: Zadanie, wbrew początkowemu wrażeniu, jest dość proste i sprowadza się do „umiejętnego” kopiowania wzorca instrukcji. Zaleca się wykorzystanie osobnych funkcji do parsowania adresów rejestrów, czy wartości `imm`.

**Zadanie 13.5** Napisać program w zaproponowanym języku asembler, który wykona następujące operacje:

1. zapisanie do rejestru R0 wartości 0 (instrukcja 0),
2. zapisanie do rejestru R1 wartości 4 (instrukcja 1),
3. dodanie wartości 1 do poprzedniej wartości rejestru R0 (instrukcja 2),
4. zapisanie do rejestru R2 wartości operacja AND na rejestrach R0 i R1 (instrukcja 3),
5. skok do instrukcji 2 w przypadku, gdy w rejestrze R2 jest wartość 0 (instrukcja 4),
6. zapisanie do rejestru R3 wartości 1 (instrukcja 5),

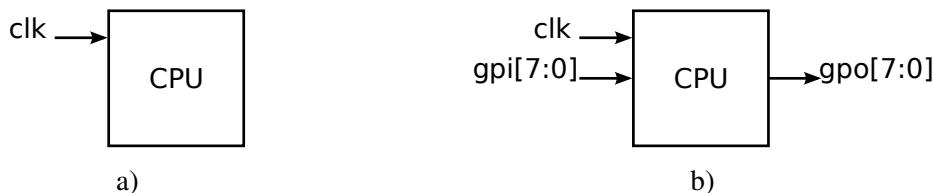
Proszę skompilować program, przy pomocy skryptu z zadania 13.4 i uruchomić go na procesorze (symulacja). Uwaga. Jest dość prawdopodobne, że na tym etapie pojawią się błędy w parserze, które się wcześniej nie ujawniły.

Jaką operację realizuje powyższy program ? Proszę napisać odpowiadający mu kod w języku C/C++;

## 14 — Procesor - testowanie i weryfikacja

Na poprzednich laboratoriach zaprojektowaliśmy procesor i zaproponowaliśmy listę instrukcji asemblera, które umożliwiają wykonywanie na nim programu. Moduł posiadał tylko jedno wejście (`clk`) i żadnego wyjścia (rysunek 14.1 a), co w pewnym stopniu utrudnia weryfikację poprawności stworzonego rozwiązania i uniemożliwia wykorzystanie procesora do wykonania jakiejkolwiek interakcji z otoczeniem. Narzędzia symulacyjne pozwalają na podglądanie rejestrów i połączeń wewnętrz modułu, jednak zaprojektowanie automatycznego środowiska testowego jest w takim przypadku mocno utrudnione.

W ramach niniejszego laboratorium zajmiemy się interakcją z otoczeniem przy pomocy portów I/O (rysunek 14.1 b). Umożliwią one następnie przetestowanie zaprojektowanego rozwiązania zarówno symulacyjnie jak i na wykorzystywanej w pracowni karcie ewaluacyjnej Zybo.



Rysunek 14.1: Procesor z poprzedniego laboratorium, oraz procesor z portami wejścia wyjścia (gpi, gpo - *general purpose input/output*)

### 14.1 Realizacja portów I/O

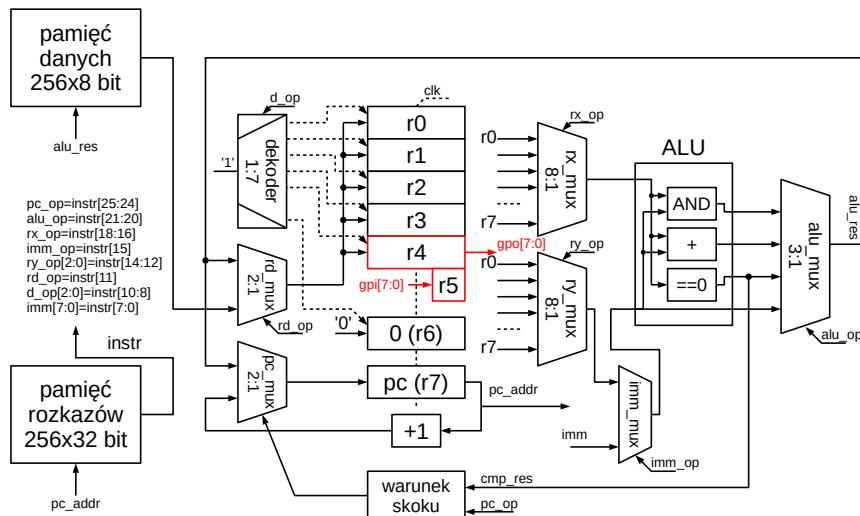
Zaprojektowany w rozdziale 13 procesor jest bardzo prosty i nie posiada żadnej wyspecjalizowanej magistrali, która umożliwiałaby podłączanie do niego urządzeń peryferyjnych. Dlatego, aby umożliwić procesorowi dostęp do portów I/O, proponowane jest zmodyfikowanie dwóch rejestrów (np. R4 i R5). Rejestry te nie będą od tej pory mogły zostać wykorzystane jako ogólnego przeznaczenia, ale jako tzw. specjalne – przeznaczone do obsługi portów I/O.

W pierwszej kolejności omówmy modyfikację rejestru, która umożliwia sterowanie portem wyjściowym (gpo – *general purpose output*). W tym celu w module nadzorowanym przez procesora

definiowany jest port `gpo` i przy pomocy polecenia `assign` należy dołączyć do niego na stałe wartość jednego z rejestrów (np. `R4`). Każdy zapis nowej wartości do rejestru `R4` spowoduje, że na porcie `gpo`, zostanie ustawiona odpowiednia wartość.

Nieco więcej modyfikacji wymaga dodanie funkcjonalności umożliwiającej odczyt wartości portów wejściowych (`gpi` - *general purpose input*). W tym celu, należy usunąć rejestr `R5` i zastąpić go połączeniem „na stałe” z wejściem `gpi`. W ten sposób „rejestr” `R5` stał się rejestrem tylko do odczytu, a jego wartość odpowiada stanowi portu `gpi` w momencie, w którym nastąpił odczyt. Schematycznie wykonane modyfikacje zostały przedstawione na rysunku 14.1.

Zaproponowane rozwiązanie nie jest jedyną możliwością obsługi w procesorze portów `gpio`. Możliwe byłoby na przykład poszerzenie niektórych multiplekserów i dodanie nowych pól sterujących w słowie instrukcji, które umożliwiłyby odpowiednie ich przełączanie i sterowanie portami I/O. Wydaje się jednak, że zaproponowane rozwiązanie w minimalnym stopniu ingeruje w strukturę zaprojektowanego procesora i umożliwia wykorzystanie wcześniej zdefiniowanych instrukcji `mov` i `movi` do obsługi tej funkcjonalności.

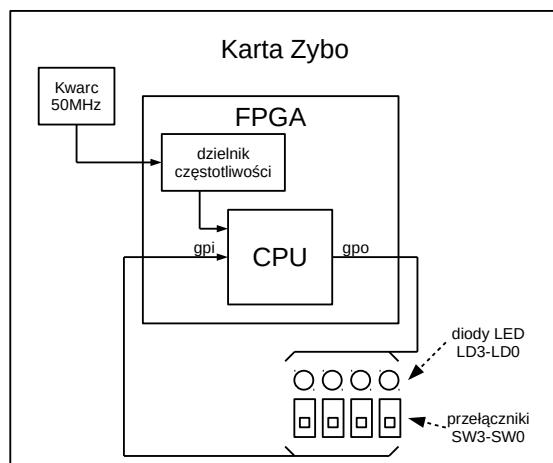


Rysunek 14.2: Modyfikacja procesora do obsługi portów `gpi` i `gpo`

## 14.2 Weryfikacja sprzętowa

Schemat systemu sprzętowego, który umożliwia przetestowanie procesora na karcie Zybo, został przedstawiony na rysunku 14.2. Zaproponowano podłączenie portów wejścia (`gpi`) oraz wyjścia (`gpo`) bezpośrednio do przełączników i diod led, które znajdują się na karcie. Ponieważ ich liczba w każdym przypadku wynosi 4, pozostałe cztery bitów wejść I/O muszą zostać na stałe ustawione na 0.

Jako źródło zegara proponowany jest znajdujący się na karcie Zybo generator kwarcowy o częstotliwości 50 MHz. Ponieważ częstotliwość zegara jest dość wysoka, do weryfikacji sprzętowej proponuje się jej zmniejszenie przy wykorzystaniu dzielnika częstotliwości (1:1000000). Zasadniczo ułatwia to „obserwowanie” zmian bez zbytniego wydłużania rejestrów. Przykładowo, jeśli chcemy włączyć diodę na 1 sekundę, to przy zegarze 50 MHz musimy doliczyć do 50000000, a przy 50 Hz do 50. Zatem, można wykorzystać rejstry 8-bitowe do realizacji „obserwowalnych” opóźnień na diodach LED.



Rysunek 14.3: Weryfikacja sprzętowa procesora na karcie Zybo

### 14.3 Zadania do wykonania na laboratorium

**Zadanie 14.1** Proszę zmodyfikować moduł procesora w języku Verilog tak, aby posiadał on porty gpi i gpo, podłączone zgodnie z opisem z rozdziału 14.1.

**Zadanie 14.2** Proszę napisać program w języku asembler, który umożliwi wykonanie następującej interakcji z otoczeniem:

1. na 1 sekundę zapali diodę led0,
2. zapali diodę led1 i będzie oczekiwany na przełączenie przełącznika SW0,
3. na 1 sekundę zapali diodę led2,
4. zapali diodę led3 i będzie oczekiwany na wcisnięcie przycisku SW1,
5. wróci do początku programu

**Zadanie 14.3** Proszę zmodyfikować środowisko testowe zaprojektowane w zadaniu 13.5.1 w ten sposób, aby na porcie gpi pojawiły się sygnały umożliwiające przetestowanie poprawności działania procesora z wgranym programem z zadania 14.2.

Uwagi:

- należy stosować testowanie sekwencyjne. Na początku sprawdzamy, czy obsługa diody led0 jest poprawna. Jeśli tak, to analizujemy działanie przełącznika SW0 – musimy dodać odpowiednie wymuszenie. Pozostałe kroki analogicznie.
- nie jest wykluczone, że na tym etapie niezbędne będzie wprowadzenie pewnych poprawek do projektu – zarówno do modułu procesora, jak i do parsera.

**Zadanie 14.4** Proszę napisać moduł, który zostanie uruchomiony na karcie Zybo, zgodnie z opisem z rozdziału 14.2. Proszę doprowadzić do uzyskania pliku bit i weryfikacji sprzętowej procesora.

Podpowiedzi:

- do projektu należy dodać plik top, który będzie zawierałinstancję modułu procesora i dzielnika częstotliwości. Powinien on mieć dwa wejścia: clk50 oraz sw oraz wyjście led (o odpowiednich szerokościach).

- następnie należy wykonać dzielnicę częstotliwości. Standardowo stosuje się *IP Core Clocking Wizard* – konfigurator modułu zarządzania zegarem CMT. Jednak w naszym przypadku nie jest możliwe osiągnięcie wymaganej częstotliwości (limit wynosi 3.125 MHz). Zatem musimy zastosować rozwiązanie „ręczne” w postaci licznika. Jednakże proszę zauważyć, że jest to postępowanie niezalecane w typowych sytuacjach (zla praktyka projektowa), gdyż powoduje wykorzystanie do propagacji sygnały zegarowego zasobów innych niż dedykowane. Tym niemniej, dla tak prostego układu i częstotliwości 50 Hz raczej nie ma to większego znaczenia.
- w ostatnim kroku należy dodać i podłączyć instancję modułu procesora oraz dołączyć plik *xdc* (przystosować używany w projekcie wizualnym).

#### 14.4 Zadania dodatkowe

Uwaga. Poniższe zadania należy traktować jako „wyzwania intelektualne”. Nie będą one w żaden sposób oceniane.

**Zadanie 14.5** Napisać program/skrypt umożliwiający komplikację kodu w języku C do asemblera używanego przez zaprojektowany procesor. ■

**Zadanie 14.6** Porównać architekturę i instrukcje asemblera procesora zaprojektowanego na zajęciach do jednej z rodzin procesorów: 8051/AVR/ARM/MIPS. ■

**Zadanie 14.7** W procesorze zaprojektowanym na laboratorium wszystkie elementy wykonawcze i pamięci oprócz rejestrów pracują z latencją równą 0. Jak wiemy nie jest to korzystne. Jakie modyfikacje należałyby wprowadzić, żeby jednostka ALU mogła pracować z latencją równą 1. Jakie problemy pojawią się w przypadku gdy pamięć danych i rozkazów będzie miała latencję równą 1 (podpowiedź: proszę przeanalizować instrukcje skoków). Proszę zapoznać się z zagadnieniem potokowości wykonywania instrukcji w procesorze (ang. *instruction pipeline*). ■

**Zadanie 14.8** Proszę zapoznać się z pojęciem „Kompletność Turinga”. Czy procesor zaprojektowany na laboratorium jest kompletny w tym sensie ? Dlaczego tak/nie ? ■

**Zadanie 14.9** Opisać w języku Verilog architekturę procesora 8051/AVR/ARM/MIPS. ■