

# 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	Linia opóźniająca .....	43
<b>5.2</b>	<b>Zadania do wykonania w domu</b>	<b>44</b>
5.2.1	Liczniak dzielący modulo N .....	44
5.2.2	Złożony moduł logiczny .....	45

# 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”).

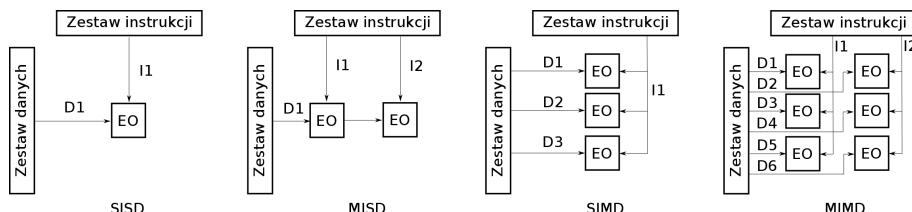
W programowaniu równoległym, na algorytm nie patrzy się jak na „sekwencję instrukcji”, ale na „zbiór elementów obliczeniowych” przez które „przepływa” strumień danych. Ponadto, aby wykorzystać możliwości jakie oferuje równoległość należy dość dobrze znać wykorzystywanyą platformę sprzętową (np. GPGPU, 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 ?? . Zadanie jakie stoi przed nami nie jest najłatwiejsze, ale pozwala w odmienny sposób spojrzeć na „programowanie”, co jest na pewno bardzo rozwijające.

Na koniec warto podkreślić, że na kurs należy patrzeć raczej jako wyzwanie intelektualne i możliwość zapoznania się z nietypową metodologią programowania, niż nabycie *strictę* praktycznych umiejętności projektowania w języku opisu sprzętu Verilog (na zajęciach wykorzystany zostanie tylko pewien niewielki podzbior możliwości tego języka).

## 1.2 Od lampy elektronowej do układu FPGA – rys historyczny

Zasada działania większości maszyn cyfrowych jest podobna. Wykorzystują one odpowiednio połączone podstawowe elementy takie jak bramki logiczne (element realizujący obliczenia) i rejesty (element pamiętający) w celu realizacji bardziej złożonych funkcji. W pierwszych komputerach wykorzystywano przekaźniki i lampy elektronowe. Wraz z rozwojem elektroniki zastosowano tranzystory, a później układy scalone. Te ostatnie uległy miniaturyzacji – od dużych obudów, które zawierały kilka bramek logicznych, po nowoczesne procesory wykorzystywane w aplikacjach i urządzeniach mobilnych, które charakteryzują się niskim zużyciem energii i dużą wydajnością obliczeniową (przykład to porównanie możliwości przeciętnego smartfona i komputera PC z początku lat 90 XX w.).

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

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

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

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

### 1.3 Budowa układów FPGA serii Artix 7 firmy Xilinx

Jak już zostało wspominane, aby dobrze projektować logikę dla układów FPGA należy poznać podstawy ich budowy. W niniejszym rozdziale zostaną zatem omówione podstawowe zasoby logiczne dostępne w układach Artix-7 firmy Xilinx. W tym miejscu należy wyjaśnić, że w trakcie kursu wykorzystywana będzie karta Zybo firmy Digilent wyposażona w układ Zynq SoC (ang. *System on Chip*). Układ Zynq określa się mianem „heterogeniczny”, gdyż w jednej obudowie zawiera dwa rodzaje zasobów sprzętowych: logikę reprogramowalną (układ FPGA) oraz system procesorowy (dwurdzeniowy procesor ARM Cortex-A9). W ramach niniejszych zajęć będzie wykorzystywana tylko część reprogramowalna, która dla układu na karcie Zybo oparta jest o logikę serii Artix-7 firmy Xilinx. Wykorzystanie możliwości tworzenia systemów sprzętowo-programowych z wykorzystaniem układów Zynq jest przedmiotem dalszego kursu pt. „Programowo-Sprzętowa Realizacja Algorytmów”. Ogólny opis rodziny Zynq można odnaleźć w dokumencie [?].

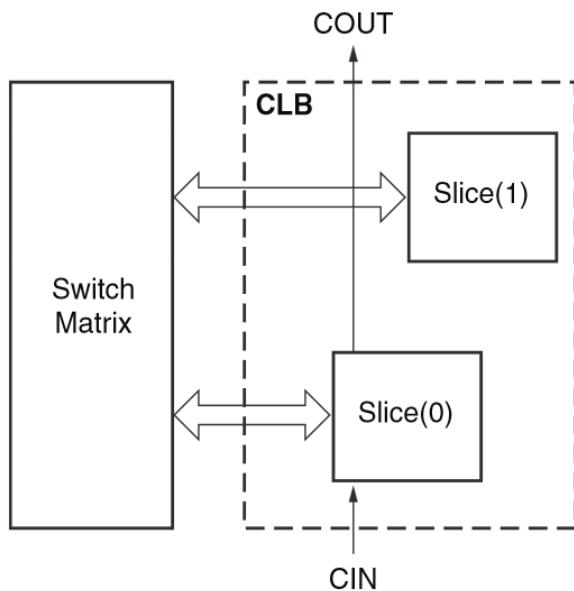
#### 1.3.1 Blok CLB

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



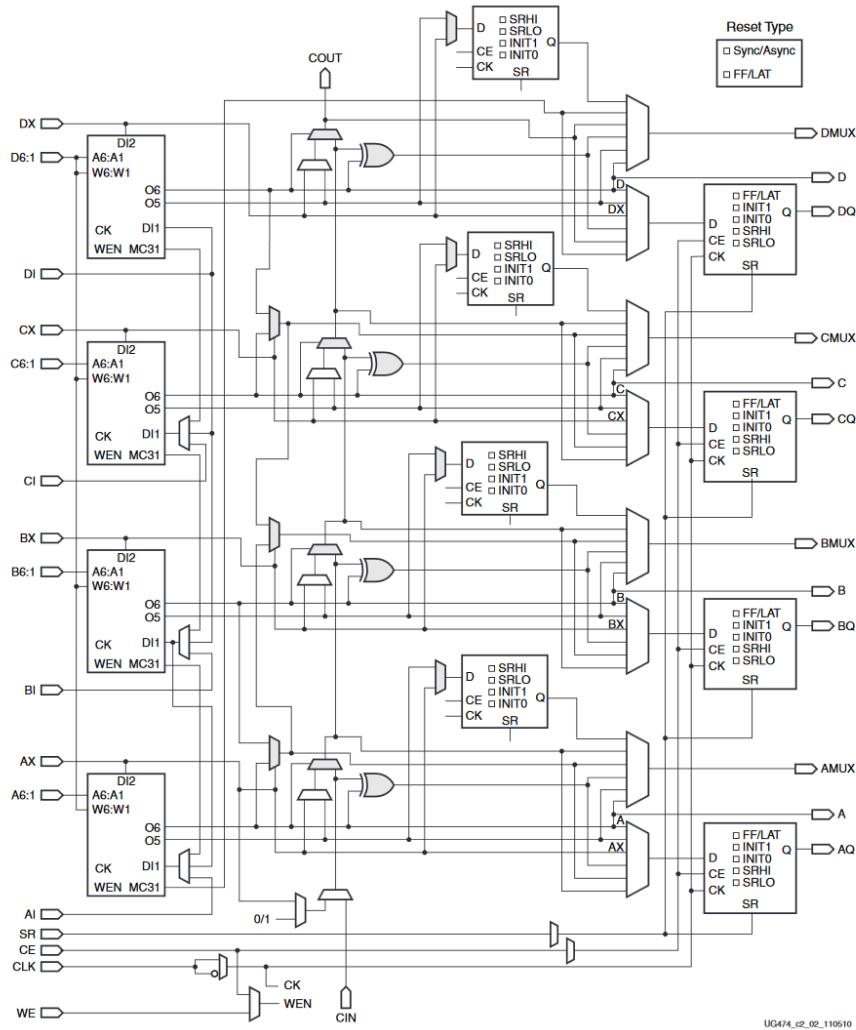
Rysunek 1.2: Schemat budowy bloku CLB. Źródło: [?]

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

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

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

### 2.2 Język opisu sprzętu Verilog

Na laboratoriach będziemy wykorzystywać język opisu sprzętu Verilog, który został zaproponowany 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

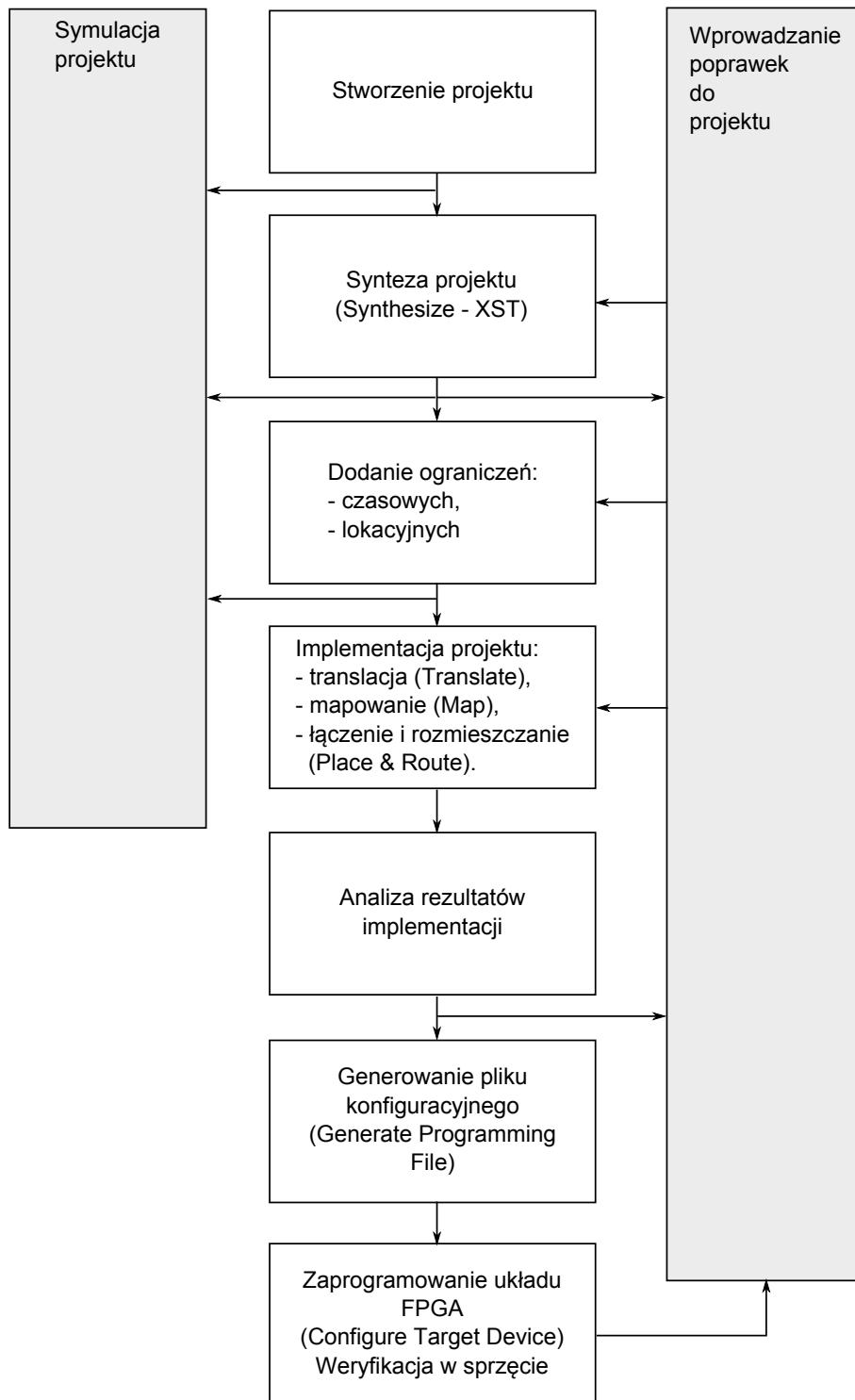
Na laboratoriach będziemy korzystać w układów FPGA serii 7 firmy Xilinx. Dedykowane dla nich środowisko programistyczne to Vivado. W pracach używać będziemy wersji 2017.4. Środowisko to nie różni się znacząco od typowych IDE np. Visual Studio lub Eclipse (choć w pewnych aspektach jest może nieco „toporne”).

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).

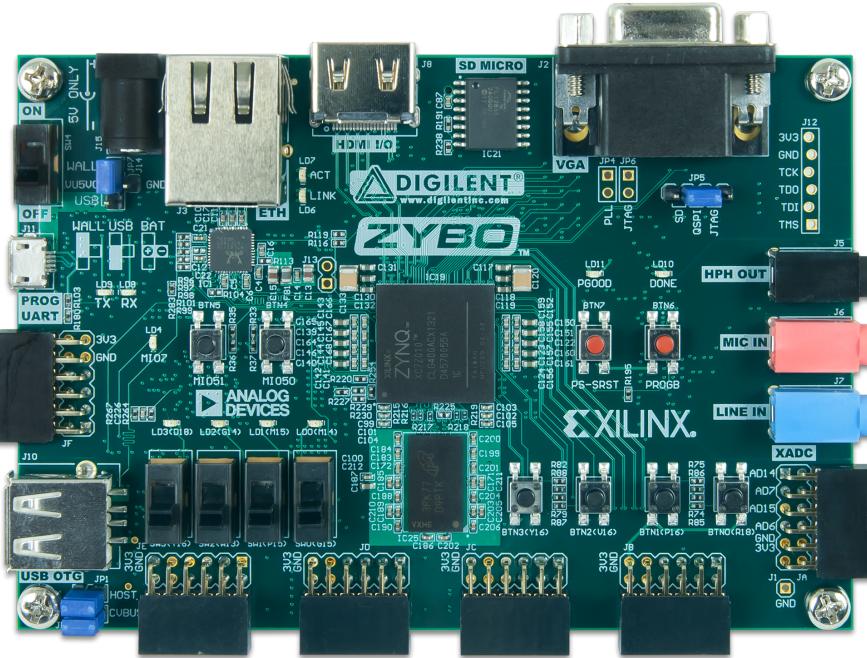
## 2.4 Zybo – platforma sprzętowa

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.

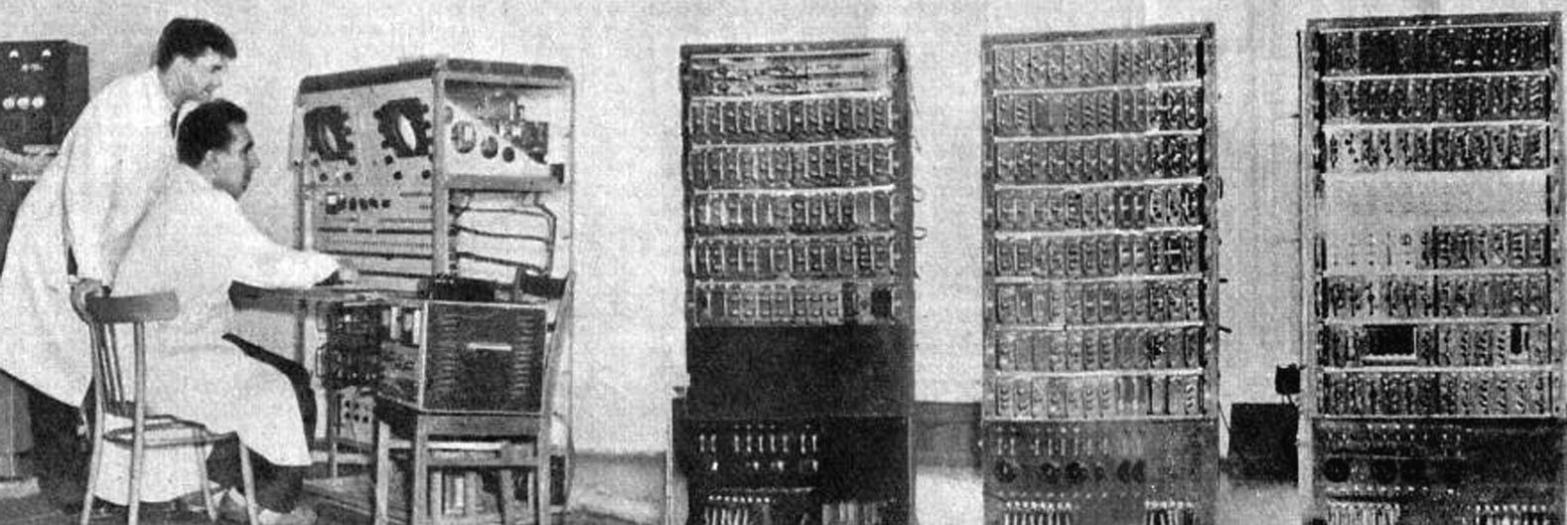
**Zadanie 2.3** Proszę zapoznać się z podstawowymi informacjami o budowie układów FPGA – rozdział 1 niniejszego skryptu. ■

## 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.





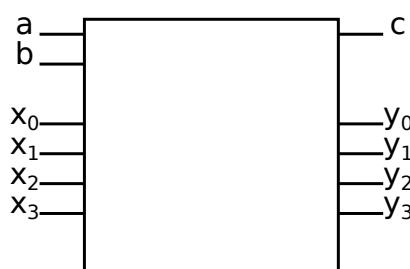
## 3 — Wstęp do projektowania struktury FPGA

### 3.1 Język Verilog – wprowadzenie

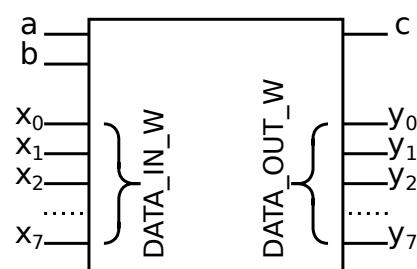
W niniejszym rozdziale zostaną przedstawione podstawowe elementy strukturalne występujące w języku Verilog, które umożliwiają projektowanie logiki w układach rekonfigurowalnych. Warto podkreślić, że jest to wariant „minumum”. Dalszych informacji należy szukać w szeroko rozumianym Internecie oraz w wielu 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.

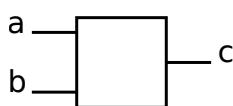
Np. jeśli portowi ma zostać przypisana liczba 123 zapisana na 8 bitach można tego dokonać na kilka sposobów:

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

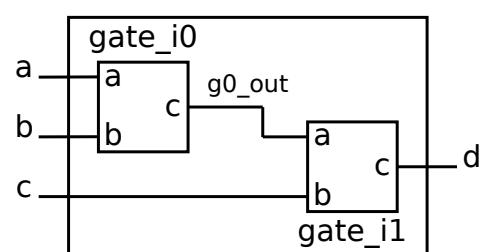
```
wire [7:0] value;
assign value=8'd123;           //decimal
assign value=8'h7b;            //hexadecimal
assign value=8'b01111011; //binary
```

### 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:



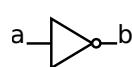
a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

#### 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:



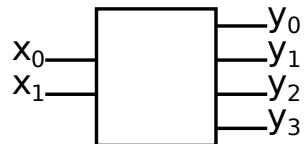
a	b
0	1
1	0

#### 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

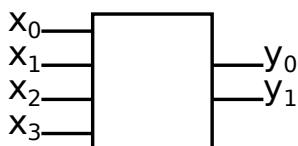
**Kod 3.1.11 — Dekoder:**

```
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:**

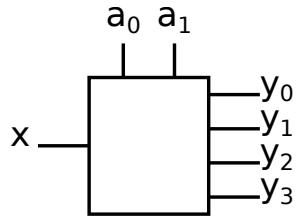
```
module encoder
(
    input [3:0]x,
    output [1:0]y
);

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:



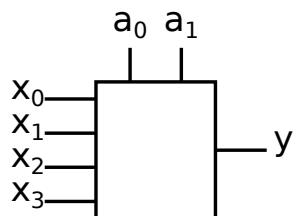
x	a <sub>1</sub>	a <sub>0</sub>	y <sub>3</sub>	y <sub>2</sub>	y <sub>1</sub>	y <sub>0</sub>
v	0	0	0	0	0	v
v	0	1	0	0	v	0
v	1	0	0	v	0	0
v	1	1	v	0	0	0

### 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:



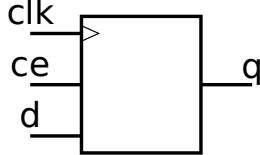
x <sub>3</sub>	x <sub>2</sub>	x <sub>1</sub>	x <sub>0</sub>	a <sub>1</sub>	a <sub>0</sub>	y
v <sub>3</sub>	v <sub>2</sub>	v <sub>1</sub>	v <sub>0</sub>	0	0	v <sub>0</sub>
v <sub>3</sub>	v <sub>2</sub>	v <sub>1</sub>	v <sub>0</sub>	0	1	v <sub>1</sub>
v <sub>3</sub>	v <sub>2</sub>	v <sub>1</sub>	v <sub>0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

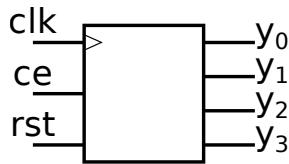
assign q=val;

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;

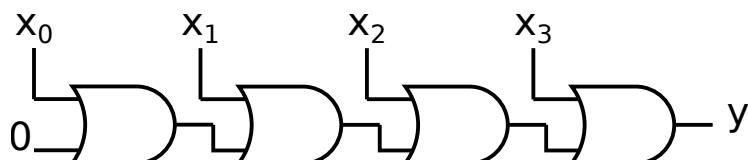
endmodule
```

### 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 zadaną strukturę:



Rysunek 3.5: Bramka OR o czterech wejściach

**Kod 3.1.17 — Bramka OR lub AND:**

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

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

**Kod 3.1.18 — Łańcuch bramek OR:**

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

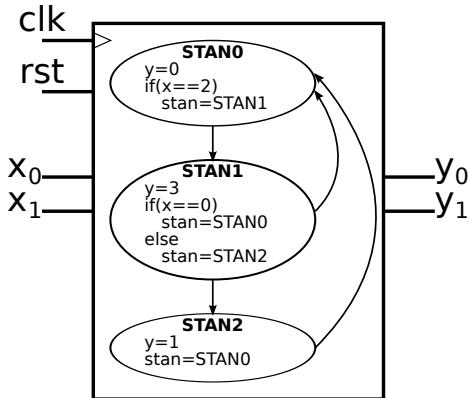
wire [LENGTH:0] chain;
assign chain[0]=1'b0;

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;

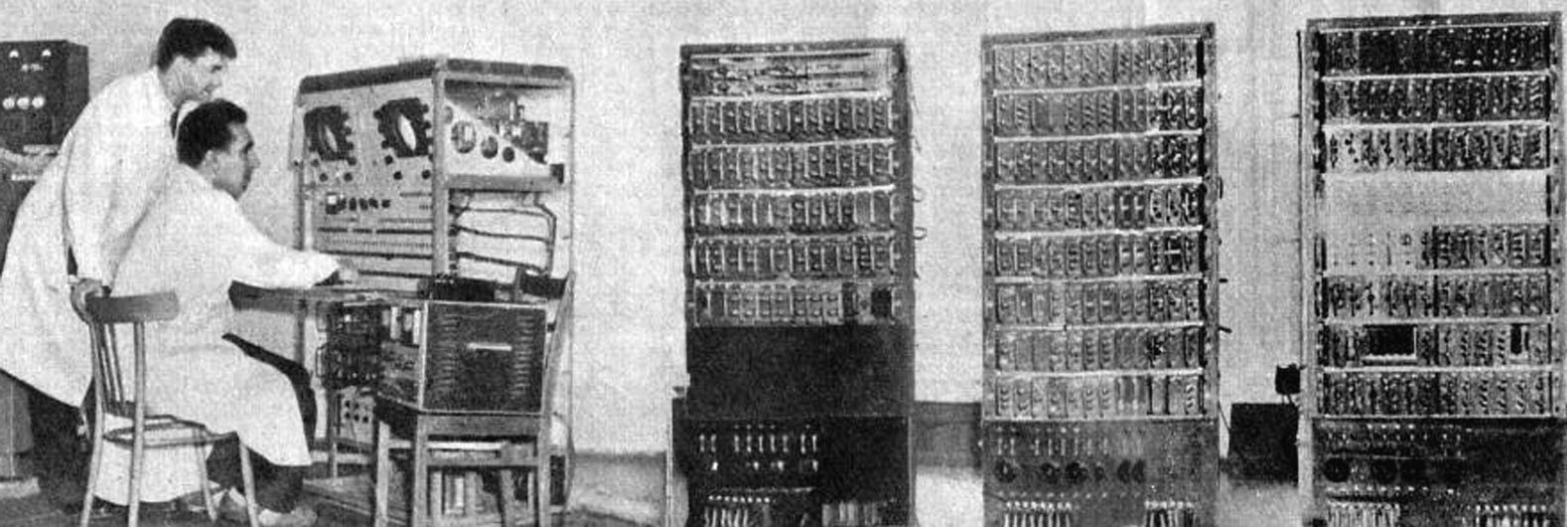
reg [1:0] state=STATE0;
reg [1:0] r_y;

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

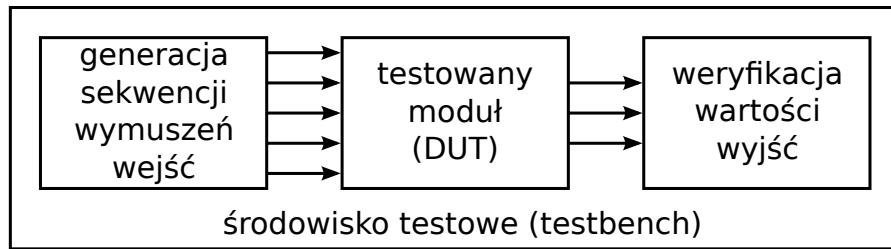
Testowanie i weryfikacja zaprojektowanych modułów sprzętowych jest bardzo **ważnym i złożonym** zagadnieniem. Wynika to bezpośrednio z zasygnalizowanego wcześniej problemu z oceną poprawności zaimplementowanego modułu sprzętowego. Po pierwsze, po uruchomieniu naszej logiki w układzie FPGA, zwykle dostajemy 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.

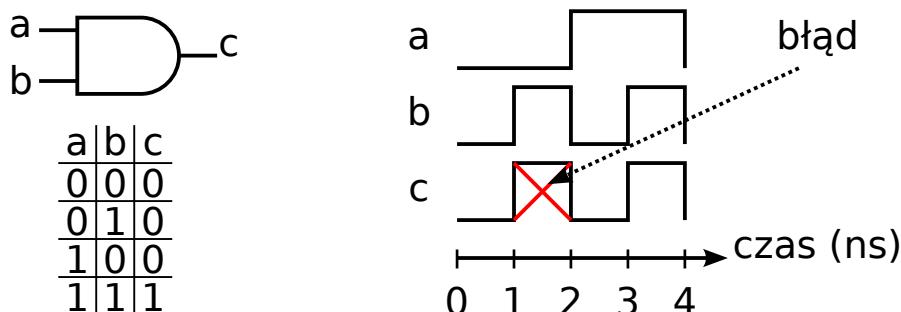
Ponieważ podstawowym elementem wykorzystywanym do opisu struktury układów scalonych w języku Verilog jest moduł, weryfikacja zaprojektowanego rozwiązania również opiera się o weryfikację działania poszczególnych modułów. W tym celu wykorzystywana jest metodologia przedstawiona na rysunku 4.1. Tworzone 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 synchronicznej (tzw. proces),
- **assign** – dla logiki asynchronicznej

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

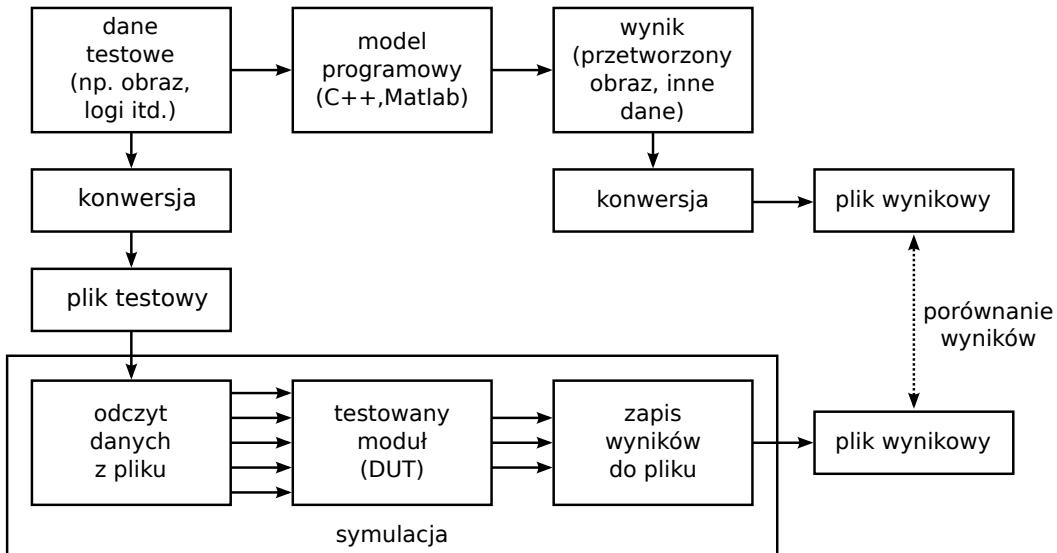
endmodule
```

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 4.2.1. Natomiast moduł, który umożliwia zapisanie wartości portu *c* do pliku wynikowego zaprezentowano w kodzie 4.2.2.

##### Kod 4.2.1 — Odczyt:

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

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

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

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

endmodule
```

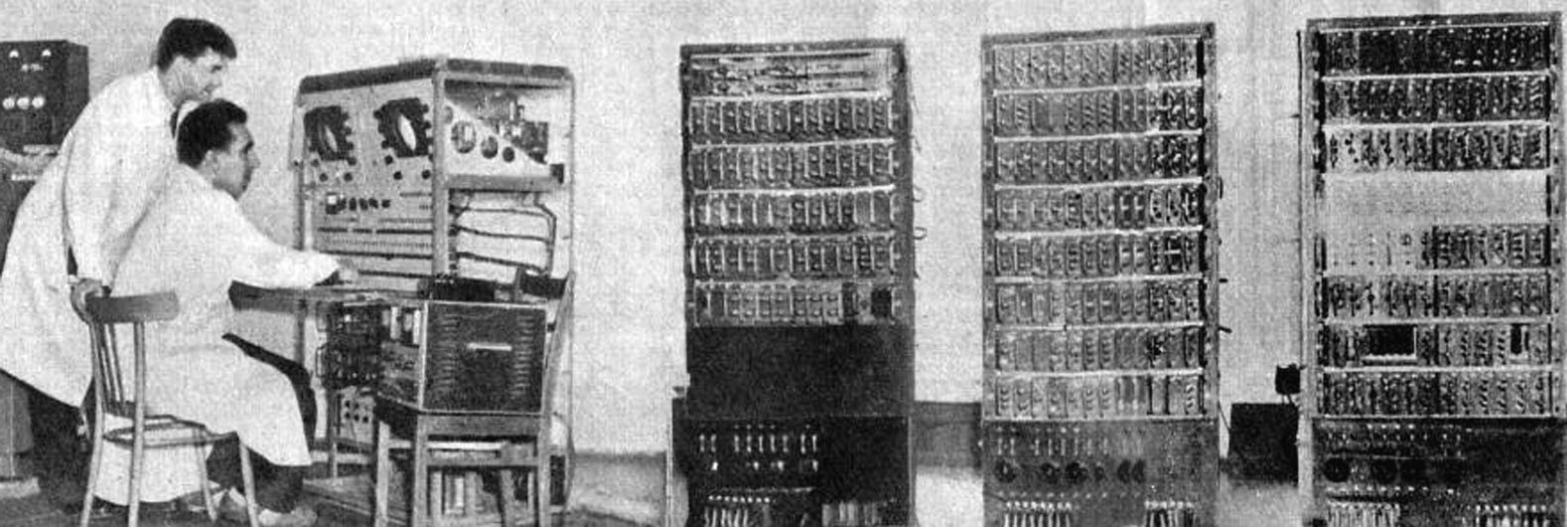
##### Kod 4.2.2 — Zapis:

```
module save_file
(
    input c
);

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

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

endmodule
```



## 5 — Verilog i weryfikacja – praktyka

W rozdziale zamieszczono zadania, które stanowią podsumowanie rozdziałów 3 oraz 4. Układ taki podyktowany jest przekonaniem, że zaimplementowany moduł (nawet najprostszy) powinien od razu zostać poddany weryfikacji. Jest to również dobra praktyka inżynierska – testujemy (dokładnie) kolejne fragmenty większej aplikacji. W ten sposób zyskujemy pewność, że „budujemy” 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 (duże 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 Linia opóźniająca

**Zadanie 5.2** Jak powiedziano w rozdziale 3.1.13, szeregowo połączone rejesty mogą zostać wykorzystane do realizacji linii opóźniającej. Rozwiązanie takie zaprezentowano na rysunku 5.1.

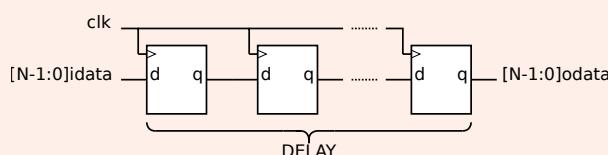
Proszę zaprojektować moduł, który posiada dwa parametry:

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

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

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

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



Rysunek 5.1: Linia opóźniająca

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

Uwaga. Moduł będzie wykorzystywany w ramach dalszej części kursu.

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

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

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

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

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

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

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

[P] Zademonstruj prowadzącemu zajęcia uzyskane wyniki.

## 5.2 Zadania do wykonania w domu

### 5.2.1 Licznik dzielący modulo N

**Zadanie 5.3** 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 5.2.1 — 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.

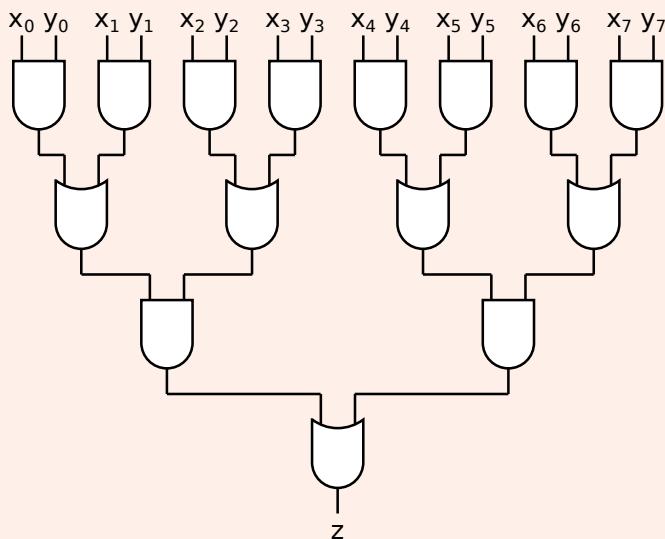
- możliwość „podglądzania” wartości zmiennych – po ustawieniu na niej kurSORA,

- pracę krokową – F8.

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

### 5.2.2 Złożony moduł logiczny

**Zadanie 5.4** 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.