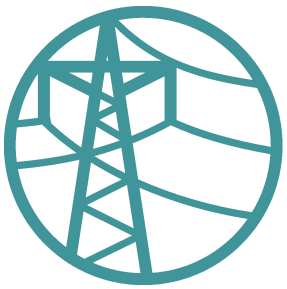


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Grzegorz Rafał Dec

Sprzętowa implementacja sieci LSTM

Praca dyplomowa doktorska

Promotor pracy:
dr hab. inż. Zbigniew Hajduk, prof. PRz

Rzeszów, 2024

Spis treści

Wykaz skrótów

ACAP - Adaptive Compute Acceleration Platform

ACP - Accelerator Coherency Port

ADAS - Advanced Driver Assistance Systems

ADC - Analog to Digital Converter

AI - Artificial Inteligence

ANN - Artificial Neural Network

APU - Application Processing Unit

ARM - Advanced RISC Machine

ASIC - Application Specific Integrated Circuit

AXI - Advanced eXtensible Interface

BPTT - Back Propagation Through Time

BRAM - Block Random Access Memory

CNN - Convolutional Neural Network

CORDIC - COordinate Rotation DIgital Computer

CPLD - Complex Programmable Logic Device

CPU - Central Processing Unit

CUDA - Compute Unified Device Architecture

DCTIF - Discrete Cosine Transform Interpolation Filter

DMA - Direct Memory Access

DSP - Digital Signal Processor

EEPROM - Electrically Erasable Programmable Read-Only Memory

EML - Extreme Machine Learning

EMG - Elektromiografia

FF - Flip-Flop

FIFO - First In First Out

FPGA - Field-Programmable Gate Array

GAL - Generic Array of Logic

GCC - GNU Compiler Collection

GPU - Graphics Processing Unit

GP - General Purpose

GPIO - General Purpose Input/Output

GWO - Gray Wolf Optimizer

HIL - Hardware In the Loop

HP - High Performance

IEEE - Institute of Electrical and Electronics Engineers

IO - Input-Output

IoT - Internet of Things

IP - Intelectual Property

IMU - Inertial Measurement Unit

kNN - k Nearest Neighbors

LDA - Linear Discriminant Analysis

LSTM - Long Short-Term Memory

LUT - Look-Up Table

MIL - Model In the Loop

ML - Machine Learning

MLP - Multi-Layer Perceptron

NR - New Radio

OCM - On-Chip Memory

PAL - Programmable Array of Logic

PBL - Programowalny Blok Logiczny

PC - Personal Computer

PL - Programmable Logic

PLA - Programmable Logic Array

PLD - Programmable Logic Device

PROM - Programmable Read-Only Memory

PS - Processing System

PNN - Probabilistic Neural Network

RAM - Random Access Memory

RBF - Radial Basis Function

ReLU - Rectified Linear Unit

RNN - Recursive Neural Network

ROM - Read-only memory

RPU - Real-time Processing Unit

RTL - Register Transfer Level

SCPL - Simplicial Canonical Piecewise Linear

SDT - Single Decision Tree

SGD - Stochastic Gradient Descent

SIMD - Single Instruction Multiple Data

SoC - System on Chip

SPLD - Simple Programmable Logic Device

SRAM - Static Random Access Memory

SVM - Support Vector Machine

TPU - Tensor Processing Unit

VLIW - Very Long Instruction Word

VHSIC - Very High Speed Integrated Circuit

VOIP - Voice Over IP

WSS - Wideband Spectrum Sensing

1. Wprowadzenie

Koncept sztucznej inteligencji został wprowadzony w połowie XX. wieku, lecz na popularności zaczął zyskiwać zaledwie w ostatnich dwóch dekadach. Związane to było z postępującą komputeryzacją, rosnącą mocą obliczeniową poszczególnych jednostek obliczeniowych oraz dostępem do dużej ilości różnorodnych danych. Rozwiązania oparte o sztuczną inteligencję są obecnie często spotykane w codziennym życiu i wiążą się z automatyzacją procesu myślowego standardowo wykonywanego przez ludzi. Pierwotnie dominującym było podejście określane mianem "symbolicznej sztucznej inteligencji", w którym reguły postępowania były definiowane programistycznie i konieczne było zgromadzenie dużej ilości tychże do poprawnego działania systemu eksperckiego [1]. Z biegiem lat symboliczna sztuczna inteligencja została wyparta przez uczenie maszynowe. W odróżnieniu od klasycznego programowania, w którym reguły wprowadzane są przez programistów, w uczeniu maszynowym wprowadza się dane i oczekiwane odpowiedzi systemu, zakładając, że system sam stworzy reguły i będzie dalej postępował zgodnie z nimi przy analizowaniu nowych danych. Można to rozumieć jako trenowanie lub uczenie systemu i zwykle odbywa się na dużych zbiorach danych, w których zwykłe wnioskowanie statystyczne bywa nieefektywne. Tego typu algorytmy zyskały popularność dopiero niedawno, ponieważ wymagają dużych mocy obliczeniowych oraz dostępności danych. Rozwiązania wykorzystujące uczenie maszynowe są obecnie wykorzystywane w wielu aplikacjach od sektora przemysłowego [2], przez opiekę zdrowotną [3], aż do zwykłych urządzeń codziennego użytku, takich jak telefony komórkowe, gdzie realizują zadania takie jak rozpoznawanie mowy [4], czy tłumaczenie [5]. Obecnie występująca tendencja do Smart Manufacturing oraz Industry 4.0 skutkuje dużym zainteresowaniem uczeniem maszynowym i wprowadza nowy paradygmat sterowania, który wykorzystuje tzw. techniki Big Data oraz IoT[6, 7].

Wyróżnić można kilka rodzajów uczenia. Jednym z nich jest uczenie nadzorowane, w którym dane wykorzystywane w trakcie uczenia są opisane np. przyporządkowane do klasy lub przyporządkowana jest im oczekiwana wartość liczbowa. W trakcie uczenia tego typu, zgromadzone, przygotowane i opracowane dane podawane są na wejście modelu, który następnie oblicza na ich podstawie wartość wyjściową. Wartość ta jest porównywana z odpowiadającą im klasą lub oczekiwaną wartością liczbową, następnie obliczany jest błąd według wybranej funkcji straty. Błąd jest wykorzystywany

następnie do korekcji parametrów modelu w celu jego minimalizacji, zgodnie z przyjętą strategią optymalizacyjną.

Kolejnym rodzajem uczenia sieci jest uczenie pół-nadzorowane, w którym wykorzystuje się część danych etykietowanych i część danych nieetykietowanych. Przykładem może być tu analiza tekstu, w której nie wszystkie dane uczące posiadają odpowiednie etykiety.

Pozostałym rodzajem jest uczenie nienadzorowane, czego przykładem mogą być auto-enkodery, które w trakcie uczenia mają na wejściu i wyjściu takie same dane, przez co uczą się odwzorowywać informacje z danych uczących. Takie modele można wykorzystać w systemach rekomendujących, w których na podstawie zaobserwowanych wzorów zachowań sugeruje się produkt lub usługę. Proces uczenia często wspomagany jest przez mechanizmy automatycznego przekształcania danych do reprezentacji, która ułatwiłaby wykonanie danego zadania.

Historycznie pierwszym wykorzystywanym rodzajem modelowania rozwiązywanego problemu było modelowanie probabilistyczne, oparte na zastosowaniu zasad statystyki. Na podstawie danych uczono, np. model regresji logistycznej, regresji liniowej, czy naiwny klasyfikator bayesowski, i dokonywano predykcji, często z bardzo dobrym rezultatem, co tłumaczy wciąż niesłabnącą popularność takiego sposobu modelowania. W latach 90-tych dużą popularność zyskały maszyny wektorów nośnych (SVM) opracowane w Bell Labs, które w przypadku prostych problemów uczenia charakteryzowały się dużą wydajnością. Metoda ta opiera się na stworzeniu granic decyzyjnych pomiędzy kategoriami. Późniejsza klasyfikacja polega na sprawdzeniu, po której stronie granicy decyzyjnej znajduje się element. Granice decyzyjne oblicza się nie na podstawie współrzędnych punktów z danych uczących, lecz, stosując funkcję jądra, na podstawie ich odległości od siebie. Niestety, w przypadku dużych zbiorów danych modele SVM są trudne do skalowania i nie są efektywne w zadaniach typu rozpoznawanie obrazu [1]. Na początku lat dwutysięcznych duże zainteresowanie wzbudziły drzewa decyzyjne, w których klasyfikacja dokonywana jest w oparciu o zestaw reguł decyzyjnych, tworzonych w procesie uczenia. Podczas tworzenia reguł atrybuty, które wydają się nie być istotne w klasyfikacji, są pomijane. Występowanie takich reguł umożliwia łatwe przeanalizowanie logiki działania drzewa. Pochodnym algorytmem są tzw. lasy losowe, w których predykcja wykonywana jest na podstawie wielu wyspecjalizowanych drzew decyzyjnych. Algorytm niweluje możliwość przesadnego dopasowania, które czę-

sto dotyczy pojedynczych drzew decyzyjnych, uniemożliwia za to łatwą możliwość interpretacji działania modelu. Pomysł łączenia ze sobą wielu słabych modeli w większy system zaowocował powstaniem maszyn wzmacnianych gradientowo, gdzie iteracyjnie trenowane są nowe modele specjalizujące się w rozwiązaniu sytuacji, w których poprzedni model radził sobie nieadekwatnie.

Obecnie na popularności zyskują sztuczne sieci neuronowe (ANN), które znajdują zastosowania m.in. w rozpoznawaniu wzorców, analizie obrazu, sterowaniu silnikami elektrycznych lub rozpoznawaniu mowy. Pozwalają poprawić efektywność procesów zwykle wykonywanych przez ludzi, wykluczając m.in. błędy spowodowane zmęczeniem [8, 9, 10, 11]. Początkowym problemem badaczy było znalezienie wydajnego sposobu trenowania sieci. Dopiero w 1989 r. Yann LeCun połączył algorytmy konwolucyjnych sieci neuronowych i propagację wsteczną w celu rozwiązania problemu klasyfikacji cyfr zapisanych ręcznie. Utworzony model był później wykorzystywany przez pocztę w USA [1]. Przez wiele lat sieci neuronowe ustępowały popularności wspomnianym SVM-om, drzewom decyzyjnym, lasom i maszynom wzmacnianym gradientowo. Zainteresowanie wzbudziły ponownie dopiero wraz ze zwiększeniem się dostępnej mocy obliczeniowej, która pozwalała na wydajne przeprowadzanie procesu uczenia sieci.

Podtypem sieci neuronowych są głębokie sieci neuronowe, działające w oparciu o tzw. uczenie głębokie, które cechuje duża dokładność oraz uproszczenie procesu rozwiązywania problemów poprzez automatyzację etapu obróbki cech. Głębokość owych sieci odnosi się do wielowarstwowości i kładzenia nacisku na uczenie kolejnych warstw. Każda kolejna warstwa jest utożsamiana z lepszą reprezentacją danych wejściowych. Warstw w modelu może być kilkadziesiąt lub kilkaset, wliczając warstwy przekształcające dane, co można często spotkać np. w sieciach CNN. Warstwy te obliczają wypadkową wartość z kilku wartości, łącząc je w jedną i redukując tym samym rozmiar. Nie ma jasnego podziału od ilu warstw można już mówić o uczeniu głębokim, ale najczęściej spotyka się to określenie w odniesieniu do modeli mających co najmniej 2 warstwy - w przeciwnym wypadku mówi się o uczeniu płytkim. Wspomniana już sieć typu CNN to konwolucyjna sieć neuronowa, która jest przykładem głębokiej sieci neuronowej, często wykorzystywanej w zadaniach rozpoznawania obrazów [12, 13]. Innym bardzo popularnym rodzajem głębokich sieci neuronowych jest LSTM, typu rekurencyjnego, przeznaczona do pracy z danymi sekwencyjnymi np. podczas przetwarzania języka naturalnego [14]. Modele uczenia maszynowego są zwykle częścią

większego programu lub systemu, w którym zasoby sprzętowe muszą być współdzielone, co pogarsza wydajność. Obecnie popularne jest wykorzystywanie GPU lub TPU do wykonywania obliczeń związanych z uczeniem maszynowym. Dobrej jakości GPU są powszechnym elementem komputerów prywatnych, ze względu na wysokie wymagania sprzętowe w grach komputerowych o rozbudowanej warstwie graficznej. GPU wyposażone w technologię CUDA potrafi znacząco przyspieszyć obliczenia. W 2016 firma Google zaprezentowała czipy zoptymalizowane pod kątem sieci neuronowych, operujące na tensorach, zwane TPU. Nie są one jednak otwarcie dostępne, chociaż wersja Edge jest montowana w telefonach komórkowych Pixel 4 firmy Google [15]. Wykorzystywanie TPU odbywa się głównie przez chmurę [16], co uniemożliwia wykorzystanie go w niektórych aplikacjach np. gdy operuje się na danych wrażliwych lub bez dostępu do Internetu.

Opracowując systemy wbudowane oraz mając na uwadze, m.in. ograniczenia energetyczne, rozmiar, czy koszt, zastosowanie wydajnego GPU wydaje się nie być optymalnym rozwiązaniem, a w przypadku TPU istotny jest również problem ogólnej dostępności. Systemy oparte jedynie na CPU mogą zostać wyposażone w odpowiednio skonfigurowany moduł FPGA, który może realizować obliczenia związane z klasyfikacją w oparciu o sieci neuronowe i dostarczać informacji o wykonanej klasyfikacji lub dokonywać decyzji samodzielnie. FPGA jest rodzajem programowalnego układu logicznego o dużym stopniu scalenia. Umożliwia realizowanie funkcji logicznych, a także złożonych systemów cyfrowych, poprzez konfigurację wewnętrzną układu. Implementacja na FPGA ma możliwość równoległego przeprowadzania obliczeń w obrębie modelu przy zachowaniu elastyczności, podatności systemu na zmiany oraz zminimalizowaniu operacji niezwiązanych z obliczeniami, przez co może znacząco odciążyć procesor lub w ogóle go wyeliminować z projektowanego urządzenia. Według [17] motywacją do wykorzystania FPGA w zastosowaniach uczenia maszynowego może być znacząca redukcja kosztów projektowanego systemu. W [18] autorzy zauważają, że są zastosowania dla których ogromne możliwości GPU nie są wystarczające. Wskazują jednocześnie na FPGA, które według autorów jest wartościową alternatywą dla zadań związanych z uczeniem głębokim. Przedstawiona w artykule architektura realizuje zadanie analizy obrazu przy użyciu wielowymiarowej sieci LSTM i okazuje się być 50 razy szybsza od GPU (NVIDIA K80) oraz 746 razy wydajniejsza energetycznie. Obecnie podejmowane są próby wykorzystania FPGA w rozwiązywaniu rzeczywistych problemów, przykła-

dem jest tutaj praca [19], która przedstawia zastosowanie FPGA w zadaniu rozpoznawania obrazu w trakcie autonomicznej jazdy samochodem. Niezależnie od wybranej jednostki obliczeniowej, najbardziej kosztownym, w sensie zasobów i czasu obliczeniowego, elementem modeli uczenia maszynowego są funkcje aktywacji [20, 17, 9]. W wielu modelach funkcja aktywacji została zoptymalizowana i uproszczona np. do ReLU, natomiast wciąż jest wiele aplikacji, takich jak sieci LSTM, w których istnieje konieczność zastosowania złożonych funkcji aktywacji, typu tangens hiperboliczny, czy funkcja sigmoidalna [14].

Poza rozwojem samych układów FPGA, w ostatnim czasie można zaobserwować działania producentów układów scalonych prowadzące do połączenia w jednym układzie procesora oraz logiki konfigurowalnej. Przykładem może być tutaj rodzina układów ZYNQ oraz Versal [21, 22]. Zastosowanie takiego podejścia może znacząco uprościć obwód elektroniczny poprzez redukcję elementów, a także otwiera nowe możliwości wykonywania obliczeń z wykorzystaniem jednocześnie procesora i logiki konfigurowalnej, korzystając z ich zalet w zależności od potrzeb projektowych.

Cele pracy:

1) Opracowanie nowej metody implementacji rekurencyjnych sieci neuronowych typu LSTM na układach FPGA, w tym na układach hybrydowych z wbudowanym mikroprocesorem oraz matrycą programowalną, umożliwiającej uzyskanie porównywalnych lub lepszych czasów obliczeń względem GPU lub znaczącego poprawienia wydajności obliczeniowej w sytuacjach, w których zastosowanie GPU lub TPU jest niemożliwe.

2) Weryfikacja zaproponowanej metody poprzez wykonanie układu realizującego zadanie klasyfikacji w wybranym procesie przemysłowym.

3) Opracowanie układu cyfrowego w strukturach FPGA wspierającego proces uczenia sieci LSTM dla wybranego procesu przemysłowego.

Zakres pracy:

1) Analiza różnych metod implementacji funkcji aktywacji, ze szczególnym uwzględnieniem dokładności i szybkości działania; zaprezentowanie nowego podejścia

implementacyjnego; wyselekcjonowanie na drodze eksperymentów czterech różnych wariantów realizacji funkcji aktywacji, wykorzystanych w dalszych rozważaniach.

2) Implementacja modułu komórki LSTM oraz prezentacja sposobu wykorzystania omawianego modułu do celu budowy dowolnej sieci z warstwą LSTM.

3) Projekt i implementacja sieci LSTM w strukturach FPGA realizującej zadanie klasyfikacji w wybranym procesie przemysłowym z uwzględnieniem dwóch różnych platform uruchomieniowych, tj. platformy całkowicie sprzętowej z układem FPGA oraz rozwiązania z podziałem na sprzęt i oprogramowanie tj. z wykorzystaniem układu Xilinx ZYNQ.

4) Projekt i implementacja układu cyfrowego wspierającego uczenie sieci składającej się z warstwy LSTM dla wybranego procesu przemysłowego, z uwzględnieniem dwóch różnych platform uruchomieniowych tj. zarówno na układzie FPGA oraz z wykorzystaniem układu ZYNQ.

5) Analiza prezentowanych podejść implementacyjnych dla wybranych układów FPGA, uwzględniająca analizę kosztów oraz ilość dostępnych zasobów sprzętowych.

6) Analiza wpływu dokładności aproksymacji funkcji aktywacji zarówno na proces klasyfikacji, jak i na proces uczenia sieci LSTM.

Tezy pracy:

1) Sprzętowa realizacja sieci neuronowych typu LSTM w strukturach FPGA lub realizacja sprzętowo-programowa pozwala uzyskać znacząco większą szybkość działania w odniesieniu do realizacji całkowicie programowych, w tym z wykorzystaniem GPU.

2) Możliwa jest sprzętowa realizacja procesu uczenia sieci LSTM umożliwiająca istotne skrócenie czasu uczenia sieci.

Pracę podzielono tematycznie na rozdziały. Rozdział 2 prezentuje wykorzystane w ramach pracy technologie i narzędzia, przedstawia ideę układów programowalnych, ze szczególnym uwzględnieniem układów FPGA, charakteryzuje ich elementy składowe oraz przedstawia metodologię używaną przy projektach wykorzystujących układy FPGA.

W rozdziale 3 przedstawiono tematykę sieci LSTM. Zaprezentowano budowę komórki LSTM, rozważając różne podejścia do implementacji funkcji aktywacji na FPGA, które dalej zostały podzielone na cztery warianty implementacyjne w celu lepszego zbadania wpływu wyboru algorytmu obliczającego wartość funkcji aktywacji na parametry i pracę sieci. Dla wydzielonych wariantów wykonano i przeanalizowano implementacje komórki LSTM, a na ich podstawie wykonano różne warianty sieci LSTM do realizacji zadania klasyfikacji w procesie kucia na zimno. Implementacja sieci LSTM została wykonana również na platformie ZYNQ, z użyciem płytki uruchomieniowej ZYBO Z7-20, z uwzględnieniem różnych podejść implementacyjnych. Porównano jednocześnie ręcznie projektowaną konfigurację z wygenerowaną przy pomocy Vitis HLS.

Rozdział 4 przedstawia moduł wspomagający proces uczenia sieci LSTM zrealizowany na FPGA, wykorzystujący algorytm propagacji wstecznej w czasie oraz porównuje wszystkie warianty implementacji funkcji aktywacji w tym zastosowaniu. Przedstawia również zalety i możliwości dotyczące wykorzystywania układów FPGA do zadań związanych z uczeniem maszynowym. W rozdziale przedstawiono również implementację procesu uczenia z użyciem płyty uruchomieniowej z układem ZYNQ oraz rozważono różne podejścia implementacyjne.

Przegląd literatury jest podzielony tematycznie na części. Literatura dotycząca funkcji aktywacji przedstawiona jest w punkcie 3.3.1, dotycząca wykorzystania układów FPGA do implementacji sieci LSTM w punkcie 3.5.1, zaś wykorzystania układów ZYNQ w tym samym zadaniu w punkcie 3.6.1. Analiza literatury dotyczącej wykorzystania układów FPGA i ZYNQ do implementacji sieci LSTM z możliwością uczenia została zaprezentowana w punkcie 4.1.

Porównanie wykorzystywanych w pracy układów FPGA oraz otrzymanych wyników z innymi układami dostępnymi na rynku zostało umieszczone jako część punktu 4, a zbiorcze podsumowanie wraz z wnioskami dotyczącymi całej treści pracy przedstawione są w punkcie 5.

Przeprowadzone badania miały charakter zarówno praktyczny, jak i symulacyjny. W celu realizacji części symulacyjnej wykorzystano środowiska Vivado oraz ISE Design Suite firmy Xilinx, w których wykonywano syntezę, implementację oraz symulację. Wyniki porównywano z obliczeniami wykonywanymi za pomocą PC, zarówno na CPU jak i na GPU oraz z obliczeniami wykonywanymi na Raspberry Pi 3. Modele wykorzystujące CPU napisane zostały przy użyciu Pythona i podstawowych bibliotek, takich jak NumPy. Przy wykorzystaniu GPU posłużono się biblioteką Keras, a program przetestowano na dwóch różnych GPU, zaś model pracujący na Raspberry Pi 3 wykorzystywał C++11. W celu realizacji części praktycznej wykorzystano płytkę ewaluacyjną ZYBO Z7-20 z układem XC7Z020, konfigurację sprzętową wykonano w języku Verilog, w środowisku Vivado oraz wykorzystując Vitis HLS (w jednym przypadku), część programową zrealizowano przy użyciu środowiska Vitis, a sam kod został napisany w C. Wyniki implementacji na platformie ZYNQ odniesiono do pozostałych rezultatów.

2. Narzędzia i technologie wykorzystane w pracy

W rozdziale przedstawiono wykorzystywane w pracy narzędzia, takie jak sprzęt komputerowy i oprogramowanie, a także przedstawiono tematykę układów programowalnych, metod i technik wykorzystywanych przy tworzeniu konfiguracji sprzętowych oraz przedstawiono najnowsze rodzaje układów programowalnych. Ze względu na dostępność oraz wykorzystywane narzędzia i płytki uruchomieniowe, w opisie skupiono się na układach firmy Xilinx.

2.1. Oprogramowanie oraz sprzęt komputerowy

W ramach pracy powstały programy referencyjne, wykorzystujące popularne obecnie narzędzia programistyczne, często spotykane przy tworzeniu i implementacji modeli uczenia maszynowego. Według publikacji [23], której autorzy gromadzili dane w oparciu o serwis GitHub, od 2015 roku TensorFlow [24] stanowi najpopularniejszą bibliotekę wykorzystywaną w zadaniach uczenia maszynowego. W zestawieniu jako druga, obecnie najpopularniejsza, występuje biblioteka Keras [25], która w istocie jest API dla TensorFlow, za pomocą której wykonywane są obliczenia. Poza tymi bibliotekami wyróżnić można scikit-learn i PyTorch, które wraz z dwiema pierwszymi zostały umieszczone w zestawieniu w wersji z interfejsem Python. Dopiero na piątym miejscu znajduje się biblioteka Caffe z interfejsem C++, a dalej MXNet i XGBoost również z interfejsem C++. W ramach pracy powstały trzy programy referencyjne, spośród których jeden korzystał właśnie z biblioteki Keras z interfejsem Pythona. Przy tworzeniu modelu w oparciu o bibliotekę Keras, interpreter Pythona został wykorzystany do konfiguracji modelu, a dalsze operacje wykonano wykorzystując TensorFlow, który korzystał z C++ oraz CUDA [25, 24]. Program referencyjny był uruchamiany na dwóch różnych komputerach z różnymi kartami graficznymi. Drugi program referencyjny w dużej mierze korzysta z interpretera Pythona, ale obliczenia wykonywane były na procesorze, za pomocą biblioteki NumPy [26], która według producentów wykorzystuje zalety języków C i Fortran. W żadnym z wykorzystywanych programów nie wykonywano obliczeń bezpośrednio w interpreterze języka Python. Drugi program referencyjny został przygotowany w celu zobrazowania czasu obliczeń nie wykorzystujących optymalizacji sprzętowych. Trzeci z programów referencyjnych wykorzystywał C++11 wraz z biblioteką standardową, a sam program był uruchamiany na Rapsberry Pi 3.

W tym przypadku Raspberry Pi 3 wyposażone było w system operacyjny Raspbian, który oparty jest na systemie Debian. Kod powstał i był kompilowany bezpośrednio na urządzeniu, na którym również był uruchamiany. Do kompilacji wykorzystywany był kompilator GCC (GNU Compiler Collection).

Parametry komputera PC, na którym były wykonywane obliczenia referencyjne, wyglądały następująco: procesor Intel Core i7-6700K CPU 4.00GHz, 16.0GB pamięci RAM oraz karta graficzna GeForce GTX 1060 6GB, komputer pracował z systemem operacyjnym Windows 10 (w dalszej części pracy oznaczony jako GPU1). Komputer, na których wykonywane były dodatkowe obliczenia z wykorzystaniem biblioteki Keras miał zainstalowany system Ubuntu 20.04 i był wyposażony w procesor Intel Core i9-12900H 5.00GHz, posiadał 64GB pamięci RAM oraz kartę graficzną NVIDIA GeForce RTX 3080 Ti (w dalszej części pracy oznaczony jako GPU2). Raspberry Pi 3 dysponował procesorem Broadcom BCM2837 quad-core, 64-bitowy ARM-8 Cortex-A53 1.2GHz oraz 1GB pamięci RAM.

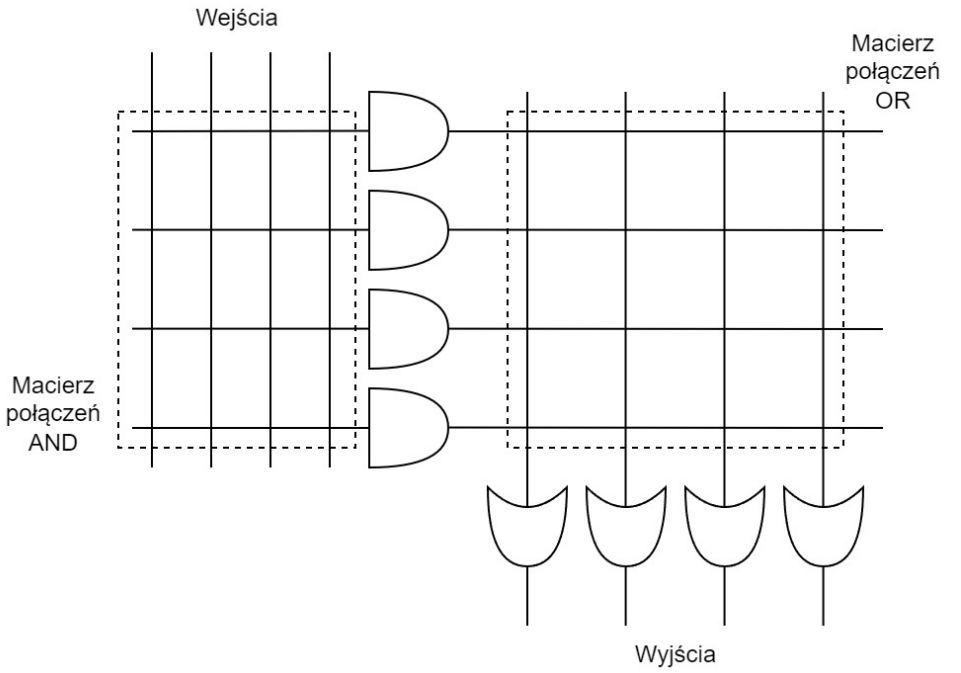
2.2. Układy programowalne FPGA

Układy FPGA są rodzajem zintegrowanych układów scalonych. W przeciwieństwie do zwykłych układów scalonych, układy FPGA pozwalają na konfigurację struktury logicznej po zakończeniu etapu produkcyjnego. Daje to możliwość implementacji struktury logicznej w zależności od potrzeb i dostępnych zasobów logicznych w układzie [27]. Współcześnie produkowane układy FPGA mogą być wyposażone w duże ilości zasobów logicznych, pozwalających na implementację bardzo złożonych algorytmów. Przykładowo płyta BittWare's XUP-VV4 wyposażona jest w 3.8 miliona elementów logicznych [28]. Możliwość konfiguracji układu po jego wyprodukowaniu znacząco obniża koszty projektowania w porównaniu z projektowaniem układów scalonych dla konkretnych zastosowań. Pozwala też na wprowadzanie ulepszeń na dalszych etapach pracy nad urządzeniem, w którym wykorzystany ma być układ scalony, a nawet wprowadzanie korekt w gotowym urządzeniu. Proces projektowania urządzenia z wykorzystaniem układu FPGA przypomina zatem proces projektowy z wykorzystaniem procesora [27]. Różnicą jest natomiast przygotowanie samego układu do pracy. W przypadku procesora przygotowuje się program, który można wgrać bezpośrednio lub po uprzednim wgraniu systemu operacyjnego. W przypadku FPGA przygotowuje się konfigurację logiczną układu. Program na procesorze działa w sposób sekwencyjny, zgodnie z kon-

kretną architekturą i listą rozkazów konkretnego procesora, wykonując kolejno polecenia zakodowane w programie. Uwzględnienie architektury procesora ma daleko idące konsekwencje w wygenerowanym na etapie kompilacji kodzie i nawet proste operacje, np. dodawanie dwóch liczb, muszą być obudowane dodatkowymi operacjami przygotowującymi do wykonania docelowej operacji np. załadowaniem wartości do rejestru operacyjnego.

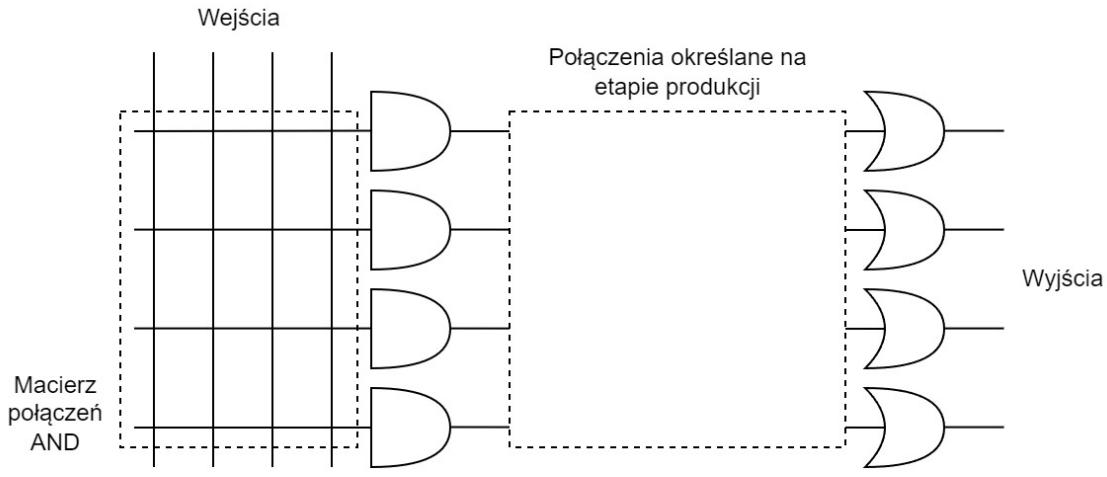
Układy FPGA nie są ograniczone do sekwencyjnego działania i mogą wykonywać wiele operacji równolegle, wykorzystując pry tym dobraną do projektu reprezentację liczbową oraz niestandardowe interfejsy komunikacyjne [29]. Dobór konfiguracji logicznej do realizacji konkretnych zadań pozwala na ograniczenie liczby operacji oraz na wykonywanie niektórych z nich równolegle, co z kolei pozwala znacząco zredukować liczbę cykli zegara potrzebnych do osiągnięcia tego samego rezultatu. Początkowo układy FPGA były głównie używane do prototypowania układów cyfrowych typu ASIC (Application Specific Integrated Circuit), obecnie układy FPGA można spotkać nawet w gotowym produkcie. Stanowią też silną konkurencję dla procesorów sygnałowych, gdzie przy bardziej złożonych zadaniach często przewyższają je w wydajności. Układy FPGA są bardzo popularne na styku warstwy fizycznej i łącza danych modelu OSI, który opisuje strukturę komunikacji w sieciach komputerowych [30]. Projektowanie konfiguracji układów FPGA jest procesem ważkim. Decyzje podjęte w trakcie definiowania architektury rzutują bezpośrednio na wiele parametrów, m.in. na prędkość obliczeń, przepustowość, wykorzystaną liczbę zasobów czy pobieraną moc. Przykładowo, długość ścieżki krytycznej bezpośrednio wpływa na maksymalną częstotliwość zegara jaką można taktować projektowany układ. Optymalizację struktury logicznej i dopasowanie do założeń projektowych (np. niska pobierana moc) można osiągnąć poprzez dobór odpowiednich w tym celu technik projektowania logiki w układach FPGA [31].

Poza układami FPGA istnieją również inne rodzaje układów programowalnych. Całą rodzinę układów programowalnych określa się mianem PLD (Programmable Logic Device), w której można wydzielić SPLD (Simple Programmable Logic Device), CPLD (Complex Programmable Logic Device) oraz opisane wcześniej FPGA. Historycznie pierwszą architekturą była rodzina układów SPLD, którą można podzielić na PLA (Programmable Logic Array), PAL (Programmable Array of Logic) oraz GAL (Generic Array of Logic) [32].



Rysunek 2.1: Schemat architektury PLA

Układy PLA składają się z szeregów bramek AND oraz OR, pomiędzy którymi oraz przed szeregiem bramek AND, znajdują się programowalne macierze połączeń (rys. 2.1), co pozwala na realizację szerokiej gamy funkcji boolowskich, gdzie głównym ograniczeniem jest rozmiar układu, narzucona restrykcja w rodzaju wykorzystywanych bramek logicznych oraz ograniczony sposób ich łączenia wewnątrz układu [32].



Rysunek 2.2: Schemat architektury PAL

Układ PAL jest podobny do układów PLA, natomiast wyposażony jest tylko w jedną macierz połączeń przed bramkami AND, co jest przedstawione na rys. 2.2. Macierz połączeń pomiędzy bramkami AND oraz OR nie jest konfigurowalna, a określana na etapie produkcji układu. Oba rodzaje tj. zarówno układy PLA jak i PAL pracują w oparciu o pamięć PROM (Programmable Read-Only Memory) i są konfigurowalne tylko raz [32].

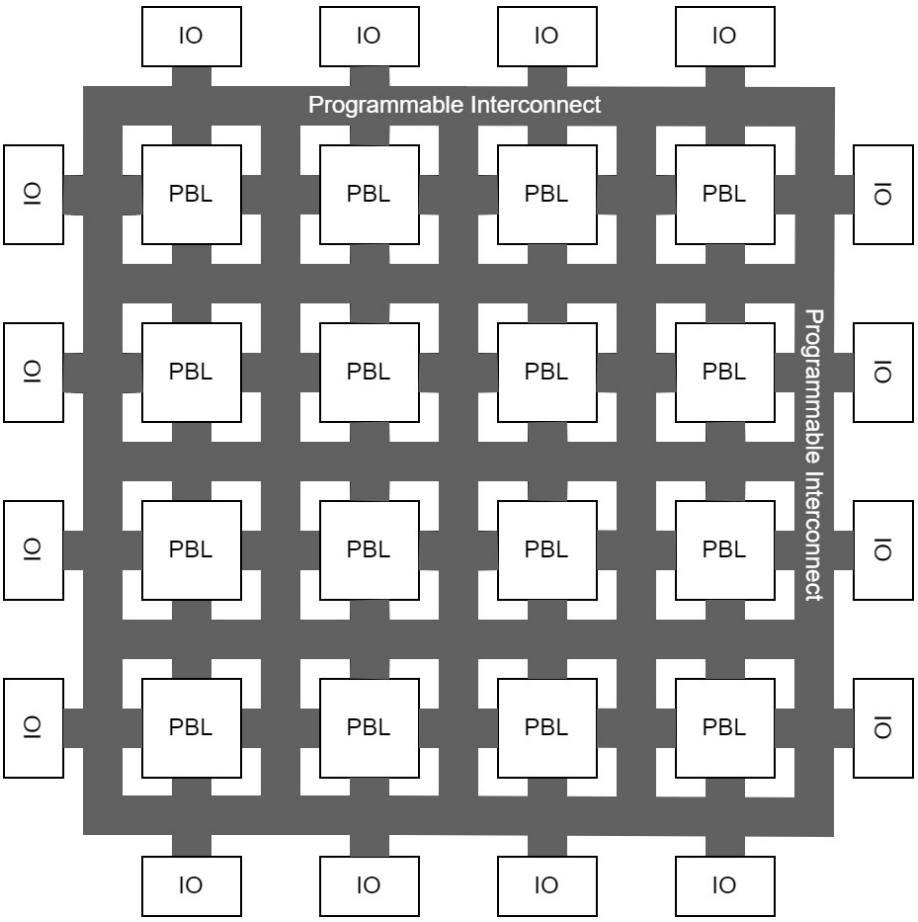
Układy typu GAL to układy podobne do PAL, tylko oparte o pamięć EEPROM (Electrically Erasable Programmable Read-Only Memory), która pozwala na rekonfigurację połączeń [32].

Kolejnym rodzajem układów programowalnych są układy CPLD, które są rozwinięciem układów SPLD i pozwalają na budowę dużo większych struktur logicznych. Układy CPLD składają się z bloków logicznych, w których znajduje się tzw. makrokomórka oraz PLA lub PAL. Bloki logiczne są ze sobą połączone programowalną macierzą połączeń. Makrokomórka jest odpowiedzialna za dopasowanie wyjścia tj. np. zapis wyniku do rejestru. Macierz połączeń może być realizowana w oparciu o model tablicowy lub multiplekserowy. Model tablicowy pozwala na połączenie dowolnego sygnału wejściowego z dowolnym blokiem logicznym poprzez dystrybucję połączeń zarówno wertykalnie jak i horyzontalnie. Model multiplekserowy wykorzystuje multipleksery do zarządzania połączeniami. Sygnały wejściowe są przyporządkowane konkretnym wejściom multiplekserów, a sygnały sterujące multiplekserami są programowalne [32].

2.2.1. Struktura i elementy FPGA

Podstawowa struktura układu FPGA przedstawiona jest na rysunku 2.3 i składa się z tzw. programowalnych bloków logicznych (PBL), programowalnych połączeń (na schemacie jako Programmable Interconnect) oraz wyprowadzeń oznaczanych jako IO [33]. PBL ułożone są w formie dwuwymiarowej tablicy i mogą składać się z podbloków. W przypadku układów firmy Xilinx podbloki określane są jako SLICE, składają się przede wszystkim z tablic przeglądowych oznaczanych jako LUT oraz przerzutników oznaczanych jako FF. Można wyróżnić dwa: SLICEL oraz SLICEM.

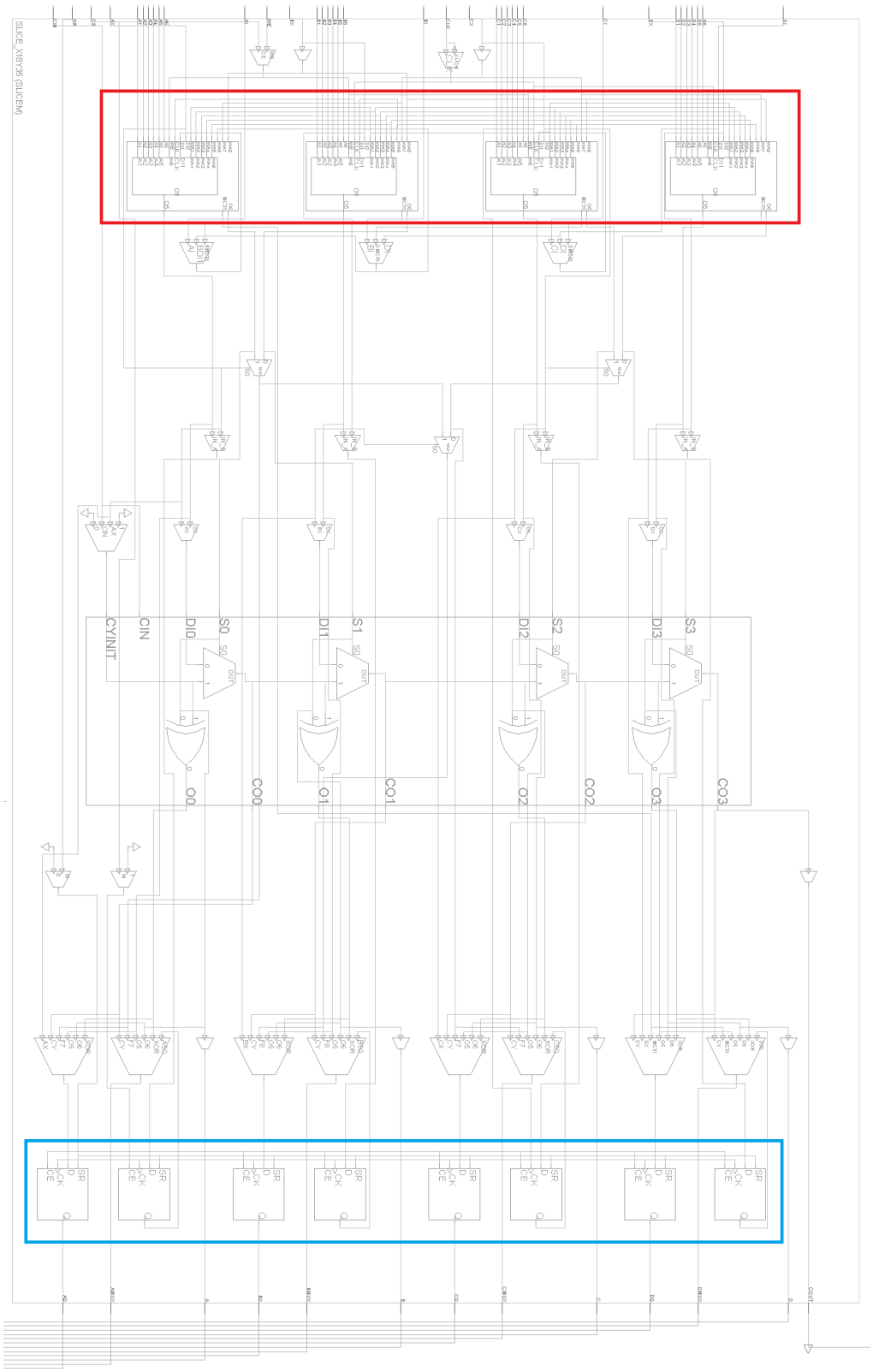
Podstawową różnicą pomiędzy nimi jest to, że SLICEL ma możliwość implementowania jedynie funkcji kombinacyjnych, podczas gdy SLICEM może być użyty do implementacji pamięci lub rejestrów przesuwnych [35, 36]. Na rys. 2.4, który prezentuje SLICEM w układzie Xilinx ZYNQ XC7Z020, można zauważyć cztery sześciowejściowe tablice LUT (kolor czerwony) oraz osiem przerzutników (kolor niebieski). Poza tymi



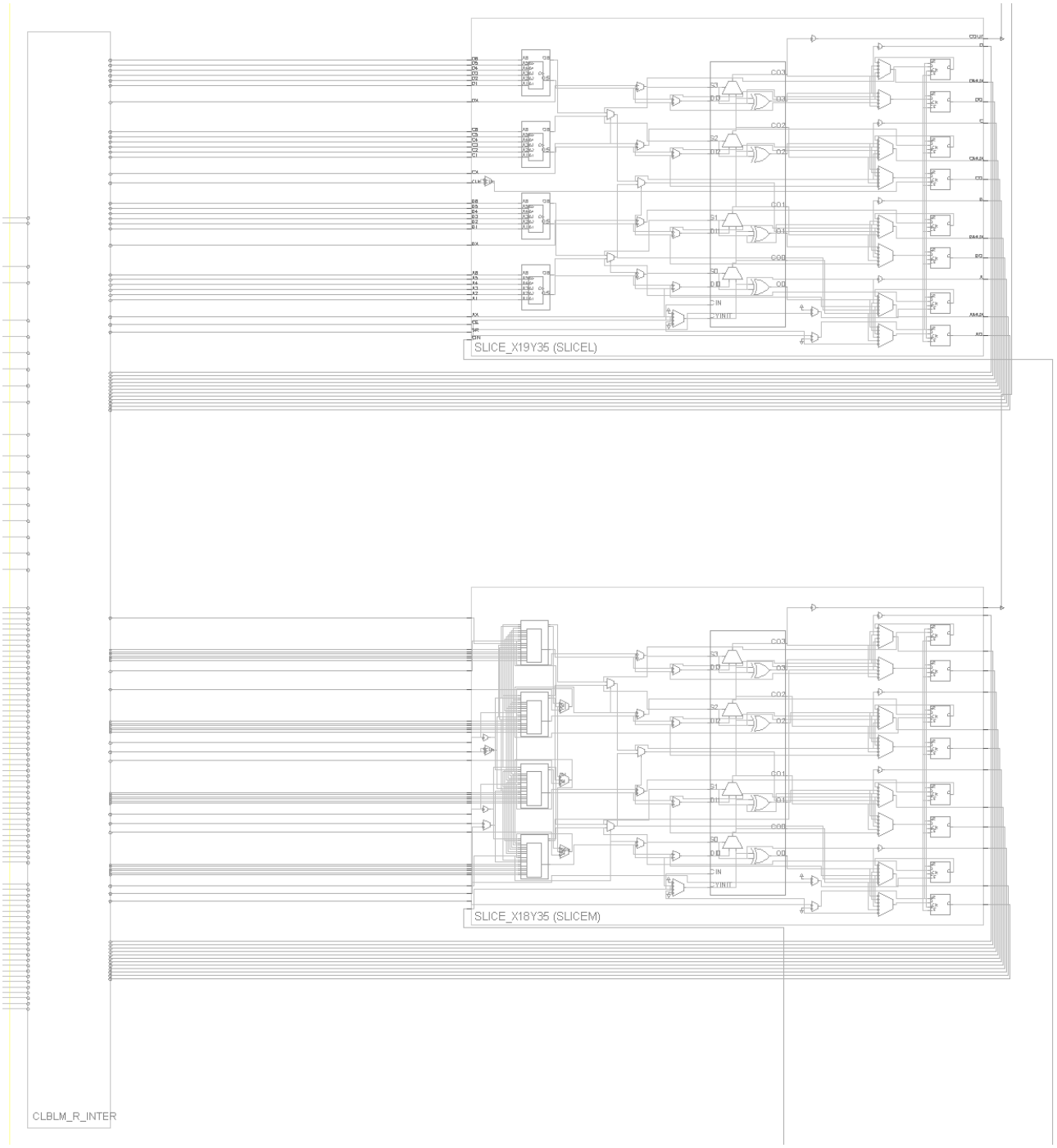
Rysunek 2.3: Schemat ogólny FPGA [34]

elementami można zauważyć dużą liczbę multiplekserów oraz tzw. łańcuch przenoszenia (blok po środku, ang. carry chain) wyposażony dodatkowo w bramki XOR. Łańcuch przenoszenia to szereg multiplekserów 2-do-1 wspomagający szybkie obliczenia arytmetyczne. W bloku PBL może być kilka elementów SLICE, przykładowo w układzie Xilinx ZYNQ XC7Z020 jest ich 2 i połączone są z pozostałymi elementami za pośrednictwem inter-konektorów (rys. 2.5). Współczesne architektury FPGA, poza podstawowymi elementami, zawierają w swojej strukturze dodatkowe elementy m.in. bloki obliczeniowe DSP, czy elementy do przechowywania danych.

Tablica przeglądowa LUT jest podstawowym elementem składowym FPGA, służącym do implementacji funkcji logicznych o dowolnej liczbie wejściowych zmiennych boolowskich, ograniczonej od góry w zależności od producenta. W przypadku firmy Xilinx ograniczenie liczby zmiennych jest równe 6 dla nowych układów, 4 w układach starszych [37]. Implementacja sprzętowa tablicy LUT może być rozumiana jako komórki pamięci połączone z multiplekserami. Taki układ może być użyty zarówno jako



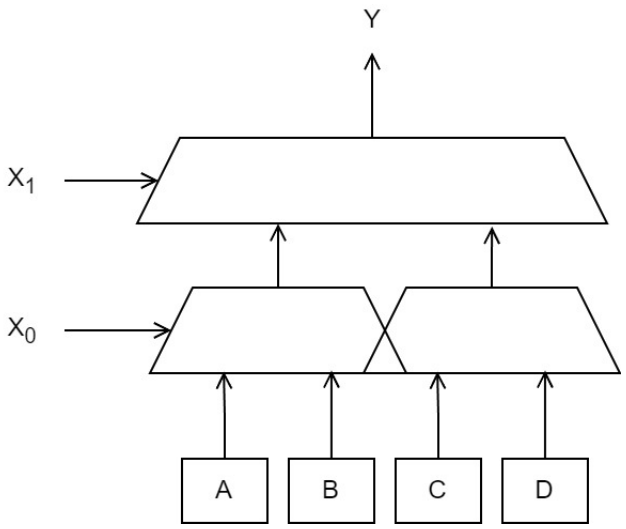
Rysunek 2.4: Zrzut z programu Vivado przedstawiający blok SLICEM układu Xilinx ZYNQ XC7Z020



Rysunek 2.5: Zrzut z programu Vivado przedstawiający blok PBL układu Xilinx ZYNQ XC7Z020 oraz blok inter-konektora

część systemu obliczeniowego, jak i element przechowujący dane. Z tablicy przeglądowej o sześciu wejściach można uzyskać 64 jednostek SRAM (Static Random Access Memory), które są najszybszym dostępnym na FPGA rodzajem pamięci [35]. Schemat tablicy LUT dla dwóch zmiennych wejściowych znajduje sie na rys. 2.6 [37]. Dla przykładu, w układzie XC7Z020 jest 53 200 tablic LUT, spośród których 17 400 tablic może

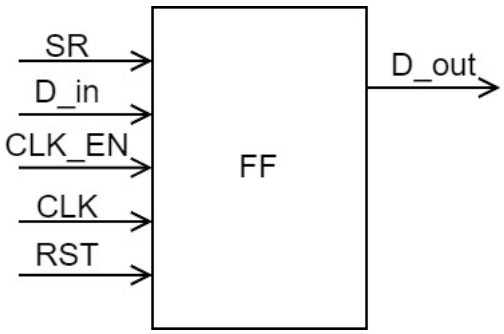
być użytych jako pamięć albo jako rejestr przesuwny. Dane do tablic LUT wgrywane są na FPGA wraz z konfiguracją.



Rysunek 2.6: Tablica LUT

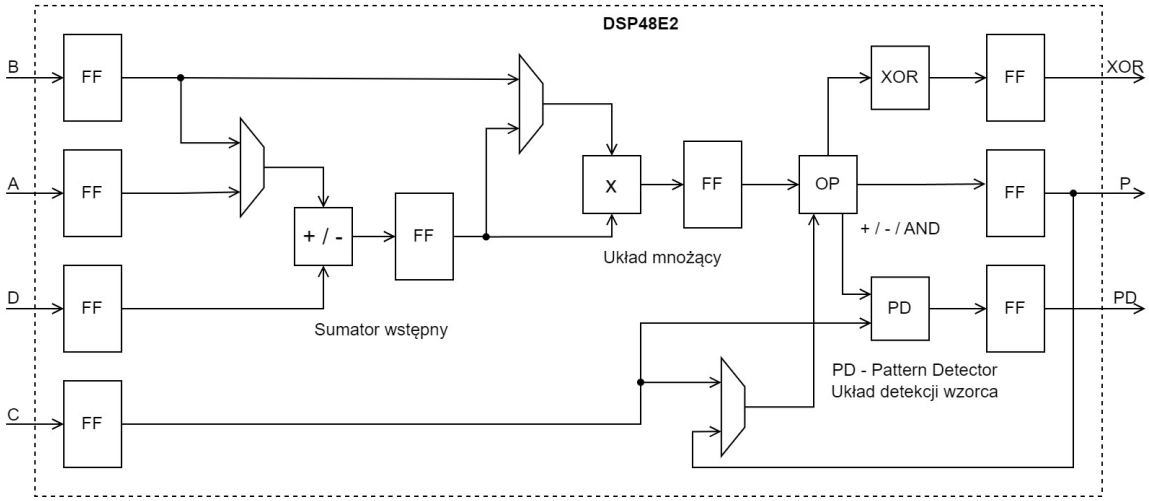
Przerzutnik to układ sekwencyjny Moore'a z jednobitowym stanem wewnętrznym, zgodnym ze stanem wyjściowym oraz stanem wejściowym równym wzbudzeniu. Jest to elementarny automat dwustanowy umożliwiający implementację układów sekwencyjnych [33]. Wykorzystywanym przerzutnikiem w strukturze FPGA jest przerzutnik typu D, przedstawiony na rys. 2.7. Przerzutnik ten jest rodzaju synchronicznego tj. zapamiętuje wartość z wejścia danych D_in i przekazuje na wyjście w każdym takcie zegara, ale tylko wówczas, gdy sygnał na wejściu CLK_EN jest w stanie aktywnym. Pozwala to na zapamiętanie wartości na wyjściu przez więcej niż jeden cykl zegara [37]. W przypadku przerzutników typu D zaimplementowanych w układzie Xilinx ZYNQ XC7Z020 (rys. 2.4) można zauważyć, że nie występuje sygnał RST, sygnał D_in oznaczony jest jako D, CLK_EN jako CE, CLK jako CK, a D_out jako Q.

Bloki DSP są złożonymi strukturami logicznymi dostępnymi w ograniczonej ilości, zależnej od układu, w strukturach FPGA, które można wykorzystać do realizacji różnorakich zadań związanych z obliczeniami lub przetwarzaniem danych. Wykorzystanie bloków DSP znacząco przyspiesza obliczenia i redukuje zużycie zasobów sprzętowych, takich jak tablice LUT lub przerzutniki. Uproszczony schemat wewnętrzny bloku DSP znajduje się na rys. 2.8. Można na nim wyróżnić przerzutniki wejściowe oraz wyjściowe, sumator wstępny, układ mnożący, blok operacyjny (dodawanie, odejmowanie,



Rysunek 2.7: Przerzutnik typu D

iloczyn logiczny) oraz układ detekcji wzorca. Blok DSP ma możliwość przekazania wyniku operacji ponownie na jedno z wejść bloku operacyjnego, co umożliwia wykonywanie takich operacji jak akumulacja [38]. Pojedynczy blok posiada m.in. układ mnożący 25x18 (szerokość wejść), pracujących w kodzie uzupełnień do dwóch, 27-bitowy układ sumatora wstępnego, 48-bitowy akumulator, który możne być kaskadowo rozbudowany, 48-bitowy blok logiczny oraz blok arytmetyczny wykonujący obliczenia w jednym cyklu zegara, działający w trybie dual 24-bitowym lub quad 12-bitowym. Bloki DSP wspierają zarówno sekwencyjne jak i kaskadowe operacje obliczeniowe [39].



Rysunek 2.8: Schemat DSP48E2 [39]

Budowa bloku DSP oraz jego konfiguracja jest kwestią kluczową, która rzutuje na jakość późniejszych obliczeń. Z tego względu w narzędziach takich jak Vivado czy ISE Design Suite firmy Xilinx, podczas prac nad projektem wykorzystującym DSP, można skorzystać z gotowych generatorów tzw. generatorów wirtualnych komponen-

tów oznaczanych jako IP-core (Intelectual Property), które pozwalają na utworzenie bloku funkcyjnego złożonego z tablic przeglądowych, przerzutników oraz właśnie bloków DSP, usprawniając proces projektowania. Utworzone bloki funkcyjne pozwalają realizować konkretną funkcjonalność np. mnożenie liczb zmiennoprzecinkowych. Wirtualne komponenty są zorganizowane w formie biblioteki dostępnej w środowisku projektowym, które znacząco ułatwiają proces tworzenia konfiguracji struktury sprzętowej. Są rozwijane i testowane przez twórców środowisk projektowych [40, 41]. W przypadku Vivado wykorzystanie wirtualnych komponentów jest jednym z sugerowanych podejść projektowych [42]. Dodanie takiego rdzenia do projektu następuje poprzez wybór odpowiedniej funkcjonalności z katalogu, a następnie uruchomienie generatora. Dodany rdzeń może zostać użyty np. w kodzie Verilog poprzez stworzenie instancji i przypisanie połączeń do sygnałów wchodzących do lub wychodzących z rdzenia.

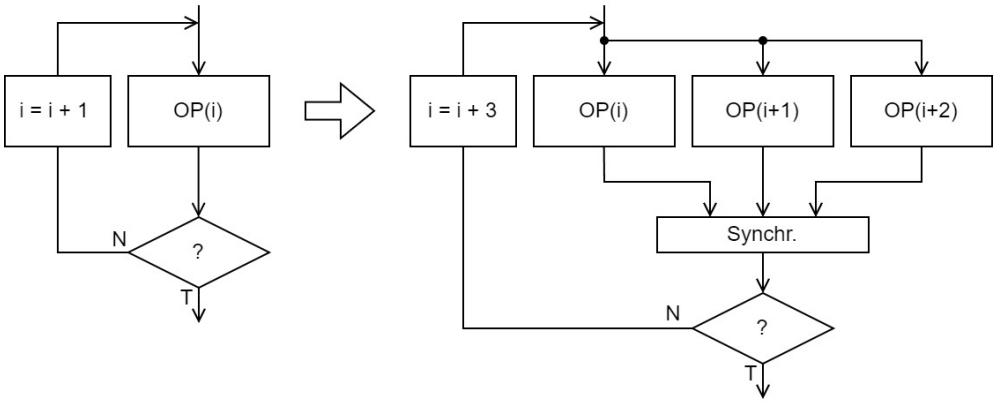
Poza wspomnianymi wcześniej tablicami LUT, które mogą pełnić rolę pamięci, w układach FPGA można spotkać bloki BRAM (Block Random Access Memory), które są blokami RAM wyposażonymi w dualny interfejs. Bloki BRAM mogą być wykorzystane do przechowywania dużego segmentu danych. W przypadku serii 7, bloki BRAM mają pojemność 36Kb, dysponują dualnym portem o szerokości portu do 72. Dodatkowo są wyposażone w programowalną kolejkę FIFO (First In First Out) oraz wbudowany, opcjonalny, obwód korekcji błędu. Każdy układ firmy Xilinx z serii 7 ma pomiędzy 5 a 1880 bloków BRAM. Operacje zapisu i odczytu są synchronizowane z zegarem. Każdy port może być skonfigurowany jako: 32Kb $\times$ 1, 16Kb $\times$ 2, 8Kb $\times$ 4, 4Kb $\times$ 9 (lub 8), 2Kb $\times$ 18 (lub 16), 1Kb $\times$ 36 (lub 32), lub 512 $\times$ 72 (lub 64). Każdy blok BRAM może zostać podzielony na dwa zupełnie niezależne 18Kb bloki BRAM, których porty mogą zostać analogicznie podzielone. Podczas odczytu każdy blok o szerokości 64b wykorzystuje kod Hamminga, wykonuje korekcję pojedynczego bitu oraz korekcję podwójnego bitu (ECC). Wbudowany kontroler dla kolejki FIFO ma możliwość pracy w sposób synchroniczny (z jednym zegarem) lub w sposób asynchroniczny (z podwójnym zegarem). Podczas pracy zwiększa wewnętrzny adres i wystawia cztery flagi używane do komunikacji. Flagi odpowiadają stanowi kolejki i reprezentują informację, czy kolejka jest pełna, pusta, prawie pełna lub prawie pusta [43, 44].

2.2.2. Techniki wykorzystywane podczas tworzenia konfiguracji logicznej

Można wyróżnić następujące techniki wykorzystywane przez projektantów systemów cyfrowych z układami FPGA podczas tworzenia opisu projektowanego systemu:

a) Planowanie - jest procesem, w trakcie którego identyfikuje się dane, zależności między nimi, sygnały sterujące oraz zależności pomiędzy operacjami zachodzącymi w strukturze logicznej. Na tym etapie jest istotne, by wyodrębnić bloki funkcjonalne, dobrać wykorzystywaną arytmetykę, wytypować operacje, które mogą zostać wykonane równolegle, oraz określić przepływ danych i sterowań w strukturze [29, 45].

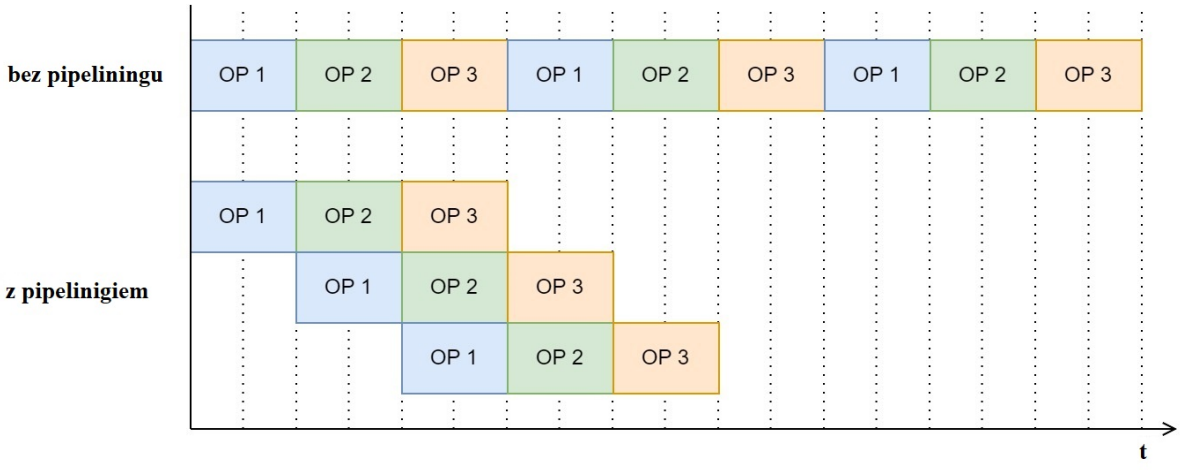
b) Rozwijanie pętli - polega na równoległym wykonaniu operacji z kilku iteracji. Na rysunku 2.9 zaprezentowany został przykład rozwinięcia pętli z jednej operacji na iterację do 3, z których każda została wykonana dla innej wartości iteratora. Ze względu na to, że czas obliczeń może być różny w zależności od wartości iteratora, w niektórych przypadkach, zależnych od warunku stopu, konieczna może być synchronizacja przed sprawdzeniem warunku stopu. Takie rozwiązanie pozwala na znaczące przyspieszenie wykonywania pełnego cyklu. Kosztem takiego rozwiązania jest zwiększenie zapotrzebowania na zasoby sprzętowe, co sprawia, że nie zawsze można z takiej techniki projektowania skorzystać. W innych przypadkach możliwe jest całkowite wyeliminowanie potrzeby stosowania pętli [45].



Rysunek 2.9: Rozwijanie pętli

c) Łączenie kodu typu pipelining - (w skrócie pipelining), to technika projektowania pozwalająca na zwiększenie poziomu równoległości w sprzętowej implementacji, w wyniku czego można np. znacząco przyspieszyć wykonywane obliczenia. Schemat zastosowania pipeliningu został przedstawiony na rys. 2.10. Na rysunku założono, że

wynik z poprzedniej operacji jest potrzebny w kolejnej w ramach jednego cyklu, tj. wynik z operacji OP 1 jest potrzebny w operacji OP 2 itd. Bez zastosowania tego typu łączenia, operacje wykonywane są sekwencyjnie, jedna operacja w danej jednostce czasu, co zaprezentowano na schemacie graficznym. Przy zastosowaniu pipeliningu, w jednej jednostce czasu może być wykonywanych wiele operacji, stąd operacja OP 1 wykonywana jest od razu po zakończeniu swojego pierwszego wywołania. Kolejne wykonania tej samej operacji powinny następować dla poprawnych danych wejściowych, które mogą być różne w zależności od wywołania. Zaletą takiego łączenia kodu jest przyrost zapotrzebowania na zasoby sprzętowe, który jest związany z bardziej rozbudowaną logiką i brakiem możliwości dzielenia niektórych zasobów (np. rejestrów do przechowywania danych). Tą zależność można wykorzystać również w drugą stronę tj. chcąc zmniejszyć wykorzystanie zasobów sprzętowych np. w przypadku małych układów, należy szeregować operacje i współdzielić zasoby sprzętowe. Kolejnym założeniem na rysunku 2.10 jest przyjęcie jednakowego czasu wykonania każdej z operacji. W rzeczywistości często zdarza się tak, że operacje mają różny czas wykonania - w takim wypadku należałoby uwzględnić etapy synchronizacji [29, 45].



Rysunek 2.10: Przykład pipeliningu

d) Redukcja ścieżki krytycznej - polega na skróceniu tzw. ścieżki krytycznej, która określa najdłuższą ścieżkę wykorzystywaną w projekcie, czyli połączenie dla którego występuje największe opóźnienie. Ma ono bezpośredni wpływ na maksymalną dopuszczalną częstotliwość zegara. W przypadku struktur, w których na dużą skalę stosowano technikę pipeliningu, może dojść do sytuacji pogorszenia maksymalnej czę-

stotliwości zegara jaką można taktować układ. W celu poprawienia tego parametru można dodać warstwę rejestrów (realizowaną przez grupę przerzutników), w której związana ze ścieżką krytyczną wartość będzie tymczasowo zapamiętywana. Taka strategia wydłuża jednorazowo o 1 liczbę cykli zegara, natomiast dzieli ścieżkę krytyczną na dwie mniejsze, a w rezultacie tego pozwala powiększyć maksymalną częstotliwość zegara, jaką można taktować układ. Inną techniką optymalizacyjną opartą o ścieżkę krytyczną jest rozpoznanie, czy równolegle do ścieżki krytycznej nie ma poprowadzonych ścieżek, które mogłyby zostać ze sobą zamienione. W rezultacie ścieżka krytyczna może zostać przemieszczona bliżej miejsca docelowego, tym samym ulegając skróceniu. Operacja może być wykonana, np. poprzez zmianę kolejności warunków w rozbudowanej instrukcji warunkowej [45].

2.3. Języki opisu sprzętu

Według ankiety przeprowadzonej w 2022 roku przez IEEE Spectrum [46] najpopularniejszym językiem HDL okazał się Verilog. Innym popularnym językiem HDL jest VHDL, który w tej ankiecie uplasował się zauważalnie za Verilogiem. Kolejnymi przykładami języków HDL są: AHDL (język HDL firmy Altera), czy ABEL-HDL, natomiast zdecydowanie ustępują popularności pierwszym dwóm i nie zostały nawet wspomniane w cytowanej ankiecie.

2.3.1. VHDL

Język VHDL powstał w latach 80., a pierwotnym celem jego stworzenia było dokumentowanie działania układów ASIC. Następnie zaczęto opracowywać w oparciu o niego symulatory logiczne, później z kolei zaczął być wykorzystywany do tworzenia i testowania układów ASIC oraz FPGA. Rozwój języka początkowo odbywał się w ramach programu rządowego Stanów Zjednoczonych, który miał na celu pracę nad bardzo szybkimi układami scalonymi. Od nazwy programu, tj. VHSIC (Very High Speed Integrated Circuit), zapożyczono nazwę języka: VHDL - VHSIC Hardware Description Language. W kolejnych latach, w roku 1987, w instytucie IEEE dopracowano język VHDL do formy IEEE Standard 1076. Kolejna wersja języka ukazała się w 1993 roku, w 2002, w 2008, a najnowsza wersja ukazała się w 2019 roku, niemniej jednak najpopularniejszą wersją języka jest wersja IEEE Standard 1076-1993 [47].

Język VHDL różni się od Veriloga przede wszystkim składnią, która bardziej

przypomina język Ada i jest zdecydowanie bardziej złożona. Poza składnią są pewne różnice funkcjonalne m.in. VHDL jest silnie typowanym językiem, pozwala użytkownikowi na tworzenie własnych typów danych, wspiera dynamiczną alokację pamięci, ale nie różnicuje zmiennych pod względem wielkości liter [48].

2.3.2. Verilog

Historia języka Verilog sięga początku lat 80. XX wieku, kiedy używany wyłącznie do symulacji układów cyfrowych w ramach symulatora Verilog XC. Pod koniec tej dekady zaczął być używany również w celach specyfikacji projektu dla narzędzi syntezy. W dalszych latach język opisu sprzętu Verilog został odseparowany od symulatora i stał się niezależnym produktem. W latach 90. został upubliczniony i doczekał się standaryzacji w roku 1995. Następnie język był nowelizowany jako standard w 2001 [49]. W tym roku również powstał System Verilog, który miał służyć do tworzenia narzędzi testowych wykorzystywanych przy weryfikacji sprzętu, lecz nie funkcjonował jeszcze jako standard. Kolejna nowelizacja nastąpiła w 2005 roku i dotyczyła zarówno języka Verilog, jak i System Verilog, który został oddzielnym standardem [50, 51]. W 2009 roku standardy zostały połączone pod nazwą System Verilog (IEEE Standard 1800-2009 [52]). Kolejna aktualizacja miała miejsce w 2012 [53], zaś obecnie język funkcjonuje jako IEEE Standard 1800-2017 [54].

Pomimo występowania w standardzie razem z SystemVerilog, narzędzia takie jak Vivado nie wspierają dodatkowych funkcji przy tworzeniu projektów RTL (Register Transfer Level). Nie znajdują one bowiem zastosowania w przypadku opisu sprzętu. Przy projektowaniu układów logicznych wykorzystuje się zatem język Verilog w standardzie z 2001 roku [49, 55]. Nowsze wersje wykorzystywane są głównie do symulacji [55].

Język Verilog pozwala na opis zachowania struktury sprzętowej jaka ma być wgrana na FPGA, na pewnym poziomie abstrakcji. Pozwala to na pominięcie żmudnych etapów m.in. tworzenia połączeń, wybierania elementów itp., które mają miejsce podczas syntezy i implementacji w środowisku projektowym [56]. Składnia języka przypomina język C. Pomimo tego podobieństwa w składni, sposób kodowania oraz przede wszystkim semantyka języka są znacząco różne [57]. Wykonanie działań w strukturze sprzętowej wygenerowanej na podstawie kodu Verilog nie jest sekwencyjne, sam język zawiera wiele niuansów, które należy rozważyć podczas projektowania, stąd proces ten może okazać się początkowo nieoczywisty [58]. Projekt wykorzystujący język Verilog

ma hierarchiczną strukturę, która pozwala na przejrzyste opisanie nawet bardzo złożonych systemów cyfrowych. Według autorów [56] Verilog jest bliższy sprzętowi niż VHDL i pozwala bardziej efektywnie tworzyć projekty. Autorzy również twierdzą, że stworzone w ten sposób implementacje są szybsze, pobierają mniej mocy i zajmują mniej miejsca.

2.3.3. Elementy języka Verilog

W niniejszym podpunkcie krótko przedstawione zostały podstawowe elementy języka Verilog, które są wykorzystywane na listingach w dalszej części pracy [50, 51]:

a) Reprezentacja liczb - w Verilogu wygląda następująco:

<rozmiar>'<podstawa><wartość>,

gdzie *rozmiar* określa liczbę bitów podawaną jako liczba dziesiętna; *podstawa* określa sposób podawania wartości *wartość* i może być: binarna (b lub B), dziesiętna (d lub D), szesnastkowa (h lub H) lub ósemkowa (o lub O). Przykładowo 1'b0 oznacza 1 bit o wartości 0.

b) Typy danych - występujące w Verilogu to przede wszystkim *wire* i *reg*. Typ *wire* reprezentuje fizyczne połączenia w opisywanym układzie i nie ma możliwości samodzielnego przechowywania własnej wartości. Typ *reg* wykorzystywany jest do modelowania zmiennych i ma możliwość przechowywania swojej wartości. Poza typami podstawowymi w języku Verilog występują też typy pochodne od typu wire, które można podzielić na: wykonujące operacje wykonywane "na przewodzie" *wand*, *wor*, *tri*, *triand* oraz *trior*; mające domyślną wartość sygnału: *tri1*, *tri0*; wykorzystywane do modelowania na poziomie przełączników: *trireg*; modelujące linie zasilania: *supply1*, *supply0*. Dodatkowo, w języku Verilog określone są typy *integer* oraz *real*, natomiast mają one ograniczone zastosowanie w procesie syntezy, natomiast mogą zostać użyte w trakcie symulacji. Wyróżnić można dodatkowo typy określające czas symulacji: *time* oraz *realtime*.

c) Selekcja bitu - w języku Verilog odbywa się poprzez potraktowanie zmiennych i sygnałów jako wektorów, z których wybiera się część wykorzystując nawias kwadratowy w sposób następujący:

identyfikator[wyrażenie_zakresu],

gdzie *wyrażenie_zakresu* może mieć formę pojedynczego bitu - wtedy precyzuje się, który bit ma zostać wyselekcjonowany, formę zakresu - wtedy podaje się wyraże-

nie w sposób:

wyrażenie_stałe_msb : *wyrażenie_stałe_lsb*

lub:

wyrażenie_podstawy + : *wyrażenie_stałe_szerokość*

lub:

wyrażenie_podstawy - : *wyrażenie_stałe_szerokość*.

d) Operatory - w języku Verilog przypominają operatory w języku C i zostały przedstawione w tabeli 2.1. Na szczególną uwagę zasługuje operator sklejania, który nie występuje w języku C. Operator ten pozwala na złączenie wielu wyrażeń w celu uzyskania jednego wektora o większej liczbie bitów. Poszczególne wyrażenia zawarte w klamrach są od siebie oddzielone przecinkami.

Tabela 2.1. Operatory języka Verilog

Rodzaj	Operator	Opis
Operatory arytmetyczne	+ - * /	suma, różnica, mnożenie, dzielenie
	**	potęgowanie
	%	reszta z dzielenia
Operatory relacji	> >= < <=	relacje
Operatory porównania	==	logiczna równość
	!=	logiczna nierówność
	===	logiczna (literalna) równość
	!==	logiczna (literalna) nierówność
Operatory logiczne	!	negacja logiczna
	&&	iloczyn logiczny
	||	suma logiczna
Operatory bitowe	~	negacja bitowa
	&	iloczyn bitowy
	|	suma bitowa
	^	XOR
	~^	XNOR
Operatory redukcji	&	redukcja AND
	|	redukcja OR
	~&	redukcja NAND
	~|	redukcja NOR
	^	redukcja XOR
	^~~^	redukcja XNOR

Operatory przesunięcia	<<	przesunięcie bitowe w lewo
	>>	przesunięcie bitowe w prawo
	<<<	przesunięcie bitowe ze znakiem w lewo
	>>>	przesunięcie bitowe ze znakiem w prawo
Operator warunkowy	?:	
Operator sklejania	{,}	sklejanie (konkatenacja)

Tabela 2.2. Priorytety operatorów w języku Verilog (od najwyższego)

Operator	Opis
[]	selekcja bitów
()	nawias okrągły
! ~	negacja logiczna i bitowa
& | ~& ~|^~^~	operatory redukcji
+ -	jednoargumentowe operatory zmiany znaku
{,}	operator sklejania
**	potęgowanie
* / %	operatory arytmetyczne
+ -	operatory arytmetyczne c.d.
<<>><<<>>>	przesunięcia bitowe
> >= <= <	operatory relacji
== != === !===	operatory porównania
&	AND bitowy
^~~^	XOR bitowy, XNOR bitowy
|	OR bitowy
&&	AND logiczny
||	OR logiczny
?:	operator warunkowy

Występowanie wielu operatorów obok siebie wymaga zdefiniowania kolejności wykonywania operacji, która to jest określona poprzez priorytety operatorów. Lista operatorów zaczynająca się od najwyższego priorytetu została przedstawiona jako tabela 2.2.

e) Instrukcja *always* **-** jest obok instrukcji *initial* instrukcją procesu tj. miejscem początku wykonywania kodu, natomiast jedynie instrukcja *always* jest w pełni syntezowalna.

f) Blok *begin end* - służy to grupowania wielu instrukcji prostych w jedną instrukcję złożoną. Zastosowanie tego bloku jest podobne do zastosowania klamer w języku C.

g) Lista wrażliwości procesu - jest wyrażeniem lub zbiorem wyrażeń, które określa moment wykonywania instrukcji procesu. W kodzie jest to wyrażenie zawarte w nawiasach okrągłych za znakiem '@'. Lista wrażliwości może być powiązana z sygnałem wyzwalającym wykonanie procesu i umożliwia modelowanie m.in. układu synchronizowanego sygnałem zegara. Przykładowo, modelując taki blok, za instrukcją *always* może znaleźć się: *@(***posedge** *axi_clk)*, które określa, że blok będzie wykonywany tylko wtedy, gdy pojawia się narastające zbocze sygnału *axi_clk*.

h) Instrukcje sterujące - typu *case* oraz *if*, w języku Verilog przypominają swoją budową i funkcjonalnością instrukcje sterujące z języka C. Formalna składnia instrukcji *if* jest następująca:

if(*wyrażenie*) *instrukcja_1*

else *instrukcja_2*.

Formalna składnia instrukcji *case* wygląda następująco:

case(*wyrażenie_1*) *wyrażenie_2* (, *wyrażenie_3*): *instrukcja* **endcase**

Po słowie kluczowym *case* występuje wyrażenie ujęte w nawias okrągły. Wyrażenie jest porównywane z każdym wyrażeniem z listy tj. z wyrażeniem 2 i 3. W przypadku, gdy wyrażenia są równe następuje wykonanie instrukcji. W języku Verilog nie występują ograniczenia co do typu wyrażeń i mogą to być również zmienne lub sygnały.

i) Przypisania - w języku Verilog można podzielić na trzy grupy: blokujące, nieblokujące i ciągłe. Przypisania blokujące i nieblokujące to przypisania proceduralne i dotyczą typów po lewej stronie operatora innych niż *wire*. Stosuje się je w celu przypisania wartości do zmiennej oraz do kontrolowania momentu, w którym przypisanie jest wykonane. Przypisanie blokujące '=' blokuje proces. Przykładowo:

#10 var1=var2;

odczekuje 10 jednostek czasu po czym wykonuje przypisanie i kolejne linie kodu w procesie. Funkcjonalność związaną z opóźnieniem można stosować tylko w przypadku symulatora. Innym przykładem może być:

@(**posedge** var1) var2=var3;

gdzie oczekuje się na pojawienie się narastającego zbocza zmiennej *var*1. Wykonanie

przypisań blokujących ma rezultat w następnej linii. Przykładowo, następujące po sobie linie kodu:

var1=data;

var2=var1;

sprawią, że zarówno zmienna *var*1 jak i *var*2 będą miały tą samą wartość po jednorazowym wykonaniu procesu. Sprawia to, że tego typu przypisania można skutecznie wykorzystać do modelowania układów kombinacyjnych. Przypisanie nieblokujące '<=' nie blokuje wykonywania procesu. Nawet jeśli wykonanie części kodu wiąże się z oczekiwaniem na jakąś akcję, to kolejne linie są wykonywane. Przy zastosowaniu przypisania nieblokującego analogiczny przykład następujących po sobie linii kodu:

var1<=data;

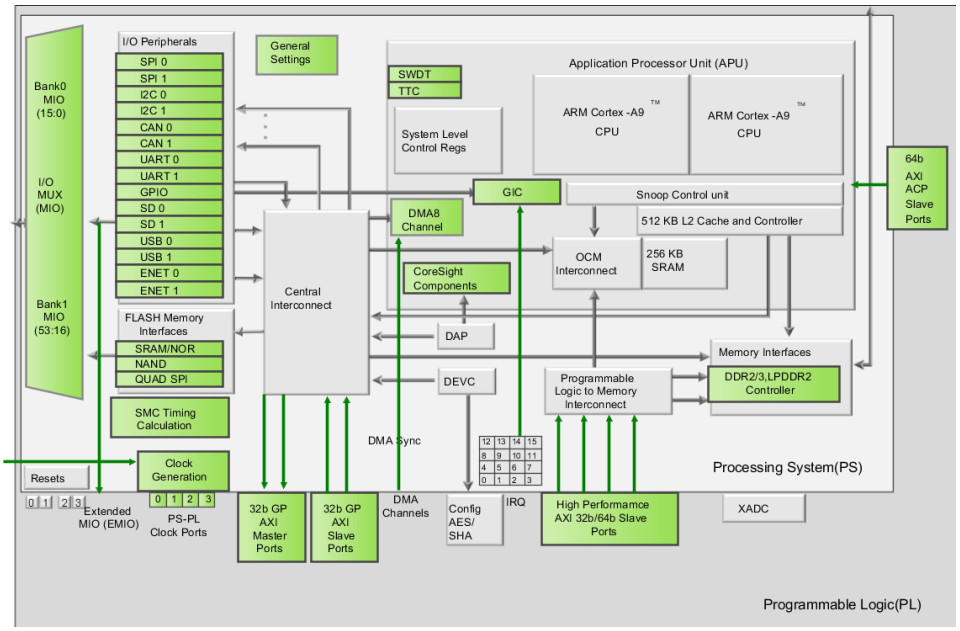
var2<=var1;

ma inny rezultat. Zmienna *var*2 ma poprzednią wartość zmiennej *var*1 i jeśli była różna od *data*, to *var*1 nie ma takiej samej wartości jak *var*2. Sprawia to, że przypisania nieblokujące są skutecznie wykorzystywane do modelowania układów sekwencyjnych. Przypisanie ciągłe może zostać użyte w przypadku jawnego skorzystania z instrukcji *assign*, po którym następuje przypisanie do instancji typu *wire*, lub w ramach specyfikacji przypisania w linii, w której definiowane jest połączenie *wire*. Przypisanie to dotyczy tylko połączeń (tj. typu *wire*) i umożliwia opis układu kombinacyjnego bez zajmowania się implementacją jego struktury. W języku Verilog występuje też proceduralna odmiana przypisania ciągłego, które jest wykonywane tylko wtedy, gdy blok w którym przypisanie ciągłe się znajduje, jest aktywny.

2.4. Układy ZYNQ

Rodzina układów ZYNQ, czyli układy SoC (System on Chip) z serii ZYNQ 7000, łączy w sobie możliwość programowania procesora ARM (jeden rdzeń Cortex-A9 w przypadku ZYNQ 7000S, dwa rdzenie Cortex-A9 w przypadku serii 7000) oraz możliwość konfiguracji sprzętowej układu FPGA. Umożliwia to rozszerzenie funkcjonalności jednego układu, minimalizując przy tym konieczność stosowania elementów peryferyjnych oraz upraszczając obwód elektroniczny [21]. Programowalna logika (PL) umożliwia dopasowanie funkcjonalności układu do rozwiązywanego problemu. Niezależnie od PL, układ jest wyposażony w szereg interfejsów m.in. w SPI, I2C, czy nawet CAN, do którego dostęp ma zarówno procesor (PS) jak i PL. Komunikacja w obrębie

układu, a więc i dostęp do poszczególnych interfejsów odbywa się poprzez blok o nazwie Central Interconnect, widoczny na rys. 2.11 i posiadający interfejsy oparte na specyfikacji AMBA4 [59].



Rysunek 2.11: Struktura wewnętrzna układów typu ZYNQ. Zrzut z edytora w programie Vivado.

Blok ten działa w sposób nieblokujący i umożliwia wiele równoległych transakcji typu master-slave, zgodnie z protokołem AXI4, który jest powszechnie używany w układach ARM, a jego pierwsza wersja została zaproponowana w 1996 roku jako część AMBA [59]. Obecnie, w wersji 4 (AMBA4) obok AXI4 zdefiniowane są jeszcze AXI4-Stream oraz AXI-Lite [60]. Komunikacja pomiędzy PS i PL może odbywać się na wiele różnych sposobów, jednym z nich jest wykorzystanie portów GP AXI (General Purpose; 2 porty typu slave oraz 2 porty typu master), które wykorzystują wcześniej wspomniany Central Interconnect. Kolejnym sposobem jest wykorzystanie portów HP AXI (High Performance; 4 porty typu slave), które umożliwia dostęp do pamięci DDR2/3, LPDDR2 oraz pamięci procesora OCM (On-Chip Memory). W ramach portów HP wbudowana jest kolejka FIFO o pojemności 1KB. Następnym sposobem jest wykorzystanie portów ACP AXI (Accelerator Coherency Port; 1 port typu slave), które mają dostęp bezpośrednio do pamięci podręcznej PS. Jeszcze innym ze sposobów jest wykorzystanie DMA (4 kanały przeznaczone dla PS i 4 dla PL) i bezpośrednie zapisywanie do pamięci, z której następnie dane mogą zostać odczytane przez drugi z komponentów.

Możliwość pominięcia procesora w zapisie do pamięci m.in. zmniejsza zużywaną energię i nie zajmuje czasu procesora [59]. Wymiana danych pomiędzy DMA a PL odbywa się za pośrednictwem protokołu AXI4-Stream, w którym dane wysyłane są w sposób strumieniowy [21, 60]. Protokół AXI4-Stream definiuje część sygnałów jako opcjonalne. Sygnały niezbędne do poprawnego działania to przede wszystkim TDATA, który transmituje przesyłane dane, ponadto TVALID oraz TREADY wykorzystywane do sterowania transmisją tj. czy slave jest gotowy do przyjmowania danych (TREADY) oraz czy dane na kanale TDATA są poprawnie umieszczone przez mastera (TVALID). Przydatnym, aczkolwiek opcjonalnym, jest sygnał TLAST, który sygnalizuje transmitowanie ostatniego elementu w strumieniu [61]. DMA może pracować w trybie pojedynczych transmisji lub w trybie scatter-gather, który automatyzuje proces zapisu lub odczytu z pamięci. Synchronizacja zadań pomiędzy PS i PL może wykorzystywać mechanizm przerwań, dzięki czemu część zadań może być wykonywana niezależnie od siebie, a komunikacja następuje tylko wtedy, gdy jest niezbędna. PS i PL są podłączone do oddzielnych obwodów zasilania, przez co możliwe jest ograniczenie zużycia energii w zależności od potrzeb i sposobu wykorzystania układu. Część programowa tj. PS uruchamia się jako pierwsza i ciąży na niej obowiązek konfiguracji całego układu [21]. Według producenta [62], układy typu ZYNQ są z powodzeniem wykorzystywane w obróbce obrazów, systemach wspomagania jazdy ADAS, w sprzęcie medycznym czy w telekomunikacji. W pracy [59] autor zwraca uwagę, że systemy oparte o układy ZYNQ często zajmują mniej miejsca i mniej ważą. Zauważa również, że zintegrowanie w jednej obudowie procesora i logiki programowalnej skutkuje małym opóźnieniem w stosunku do przypadku, w którym logika programowalna stanowiłaby oddzielny układ.

2.5. Układ Xilinx ZYNQ XC7Z020

W pracy jako fizyczny układ wykorzystano zestaw uruchomieniowy Digilent ZYBO Z7-20, wyposażony w układ Xilinx XC7Z020-1CLG400C, który poza dwurdzeniowym procesorem ARM Cortex-A9 zawiera:

- 53.2 tys. sześciowejściowych tablic LUT,
- 106.4 tys. przerzutników,
- 32KB L1 Cache dla instrukcji i 32KB dla danych,

- 630KB L2 Cache,
- 140 bloków BRAM o pojemności 36KB.
- 220 bloków DSP,
- 8 kanałów DMA (w tym 4 dla części PL),
- przetwornik ADC 2x12 bit,
- wspiera protokoły komunikacyjne: 2xUART, 2x CAN 2.0B, 2xI2C, 2xSPI, 4x 32b GPIO, 2x USB 2.0 (OTG), 2x Gigabit Ethernet, 2x SD/SDIO.

Zestaw doposaża układ wieloma elementami peryferyjnymi i multimedialnymi. Na płytce znajduje się złącze Pcam MIPI CSI-2, dwa złącza HDMI (wejściowe oraz wyjściowe), pamięć DDR3L, co umożliwia prowadzenie prac rozwojowych nad aplikacjami wizyjnymi, gdzie chętnie stosuje się układy FPGA. Płyta ewaluacyjna jest wyposażona w 6 portów typu Pmod w ustawieniu pinów 2x12, które są bezpośrednio sterowane przez główny układ i umożliwiają podpięcie do nich zewnętrznych układów np. sterownika silnika elektrycznego. Po podłączeniu przewodem USB do komputera PC możliwe jest bezpośrednie programowanie i debugowanie. W tym celu nie jest wymagany żaden zewnętrzny programator. Przewód zapewnia również niezbędne zasilanie. Ponadto na płytce znajdują się 4 przełączniki przesuwne, 6 przycisków (w tym 2 MIO), 7 diod LED (w tym 1 MIO oraz 2 RGB) [21, 63].

2.6. Układy Xilinx Versal

W ostatnim czasie firma Xilinx przedstawiła nową serię układów, będących hybrydową platformą obliczeniową, która integruje w sobie FPGA, procesor oraz programowalne, wektorowe silniki obliczeniowe. Jest więc to rozwinięcie koncepcji towarzyszącej projektowaniu układów typu ZYNQ o dodatkowe elementy. Prezentacji nowej serii układów towarzyszyło przedstawienie akronimu: ACAP, tj. Adaptive Computing Acceleration Platform. Układy Versal umożliwiają stosowanie innych modeli, korzystających z programowanych silników obliczeniowych oraz separując przesył danych od pozostałych elementów. Serię Versal można dodatkowo podzielić na pomniejsze serie, różniące się od siebie liczbą zasobów sprzętowych (od najmniejszej - względem liczby elementów logicznych): AI Edge, AI Core, Prime, HBM (seria z wysoko przepustową

pamięcią) oraz Premium. Serie Prime oraz HBM nie posiadają silników wspomagających obliczenia związanych ze sztuczną inteligencją. Każdy układ z serii Versal posiada dwurdzeniowy procesor skalarny Arm Cortex-A72 (pełniący rolę APU - Application Processing Unit) oraz dwurdzeniowy Arm Cortex-R5F (pełniący rolę RPU - Real-Time Processing Unit). Część PS jest dodatkowo wyposażona w kontrolery interfejsów m.in. USB 2.0, SPI, I2C, UART czy CAN-FD. Komunikacja pomiędzy elementami układu oparta jest, analogicznie jak w przypadku układów ZYNQ, o protokół AXI-4. Główną różnicą w porównaniu z układami ZYNQ są moduły programowalnych wektorowych silników obliczeniowych. Poza wspomnianymi seriami, każdy układ jest wyposażony w silnik obliczeniowy (AI Engine), procesor wektorowy VLIW (SIMD) oraz wewnętrzną pamięć [22]. VLIW (Very Long Instruction Word) jest rodzajem architektury o stałej długości instrukcji, zawierającej kilka poleceń jednocześnie, wykorzystującej równoległość na poziomie instrukcji. Z kolei SIMD (Single Instruction Multiple Data) to technika, w której wykorzystuje się wiele danych w trakcie pojedynczej instrukcji procesora, wykorzystując równoległość na poziomie danych [64, 65]. Silnik obliczeniowy zawiera jednostkę skalarną, wektorową, wczytującą oraz interfejs pamięci. Jednostka skalarna zawiera 32-bitowy procesor o architekturze RISC oraz 32x32 skalarny układ mnożący. Jednostka wektorowa oparta jest o arytmetykę pojedynczej precyzji i wspiera jednoczesną pracę na wielu liniach wektorowych [22].

Połączenie w jednej obudowie procesorów, FPGA oraz procesorów wektorowych pozwala na bardzo dużą swobodę. Samo projektowanie jest przede wszystkim oparte na językach programowania, co daje możliwość wykorzystania narzędzi takich jak język C, C++, czy biblioteka OpenCL. Umożliwia to prowadzenie projektów w metodykach używanych obecnie przy rozwoju oprogramowania wykorzystując narzędzia generujące strukturę logiczną dla części PL, z pominięciem projektowania opartego o RTL (który niemniej jednak wciąż jest wspierany) [66].

Według producentów seria Versal powstała w celu podążenia za trendami obserwowanymi w wielu technologiach takich jak 5G, czy motoryzacja (autonomiczne pojazdy), które napędzają zapotrzebowanie na większą wydajność obliczeniową. Z kolei podejście polegające na zagęszczeniu liczby komponentów poprzez zmniejszanie rozmiaru procesu produkcyjnego, a tym samym rozmiaru pojedynczego tranzystora, jest coraz trudniejsze do zrealizowania. W latach 60. Gordon Moore zauważył, że co 2 lata liczba komponentów w układach scalonych podwajała się, trend ten można było w przy-

bliżeniu obserwować aż do 2012 roku. Po osiągnięciu rozmiaru procesu litograficznego na poziomie 28 nm krzywa wzrostu zaczęła się nasycać, a korzyści w sensie zwiększania wydajności, redukcji kosztów i redukcji mocy przestały być tak intratne jak wcześniej. Układy FPGA ze względu na swoją konfigurowalną naturę oraz możliwość przeprowadzania obliczeń równolegle, mają szansę na pokonanie obecnie napotykanych ograniczeń, szczególnie w połączeniu z procesorami wektorowymi, które w związku z ostatnio obserwowalną eksplozją danych i popularnością sztucznej inteligencji, były w ostatnich latach silnie rozwijane [67, 68].

2.7. Narzędzia projektowe dla FPGA

W punkcie 3.3, w ramach rozważań na temat funkcji aktywacji, wykorzystano ISE Design Suite firmy Xilinx [69], który umożliwia przeprowadzenie syntezy, implementacji oraz symulacji zaprojektowanego układu. Niestety, ze względu na to, że nie jest już rozwijany, okazał się niewystarczający przy większych projektach, przez co w dalszych punktach pracy wykorzystano program Vivado [70, 71], będący nowszą wersją narzędzi do projektowania logiki układów FPGA firmy Xilinx. W przypadku prac z układem ZYNQ, do projektowania logiki wykorzystano również oprogramowanie Vivado. Do napisania oprogramowania na procesor ARM będący częścią układu ZYNQ, do kompilacji, wgrywania konfiguracji i programu oraz do uruchamiania, wykorzystano program Vitis firmy Xilinx [72]. Program ten, obok Vivado, jest rekomendowanym narzędziem do pracy z układami z serii ZYNQ. W jednym z eksperymentów wykorzystano również program Vitis HLS do wygenerowania struktury logicznej dla układu ZYNQ z kodu C. We wszystkich przypadkach, do projektowania konfiguracji logicznej FPGA wykorzystano język Verilog, a program działający na procesorze w układzie ZYNQ został napisany w języku C.

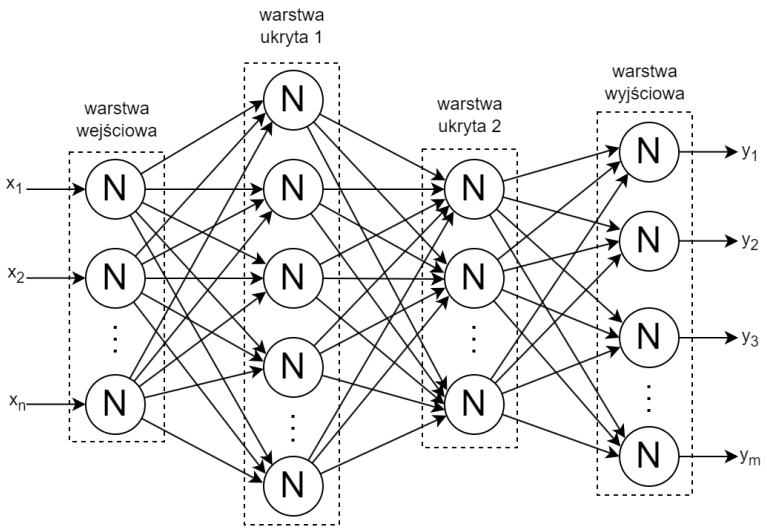
3. Sieć LSTM

Główną tematyką niniejszej pracy są sieci LSTM, które są rodzajem rekurencyjnych sieci neuronowych. Określenie sieci LSTM odnosi się do sztucznych sieci neuronowych, w których główna warstwa składa się z połączonych ze sobą rekurencyjnie komórek LSTM. Połączenie rekurencyjne oznacza, że sygnał z wyjść komórek jest zawracany z powrotem na ich wejście w kolejnym kroku obliczeniowym. Poza warstwą LSTM w sieci mogą występować również inne, wspomagające pracę sieci jednostki np. warstwa wyjściowa. Sama sieć LSTM może składać się z więcej niż jednej warstwy LSTM. Dodatkowa warstwa może działać w przeciwnym kierunku tj. przetwarzać dane od końca analizowanego szeregu. Mowa wtedy o sieci biLSTM [73]. W pracy skupiono się na sieci LSTM przetwarzającej dane w jednym kierunku oraz jej implementacji w sprzęcie. W punktach niniejszego rozdziału przedstawiono najpierw budowę komórki LSTM, będącej podstawowym elementem sieci LSTM, przedstawiono rozważania dotyczące implementacji funkcji aktywacji w układach programowalnych oraz wyszczególniono warianty implementacyjne wykorzystane w dalszych eksperymentach. Przedstawiono również implementacje komórki LSTM na układach programowalnych z uwzględnieniem każdego z wyszczególnionych wariantów oraz przedstawiono implementację klasyfikatora pracującego w oparciu o sieć LSTM do rozwiązania rzeczywistego problemu. Implementację wykonano zarówno na rzeczywistym układzie, jak i symulacyjnie w środowisku Vivado.

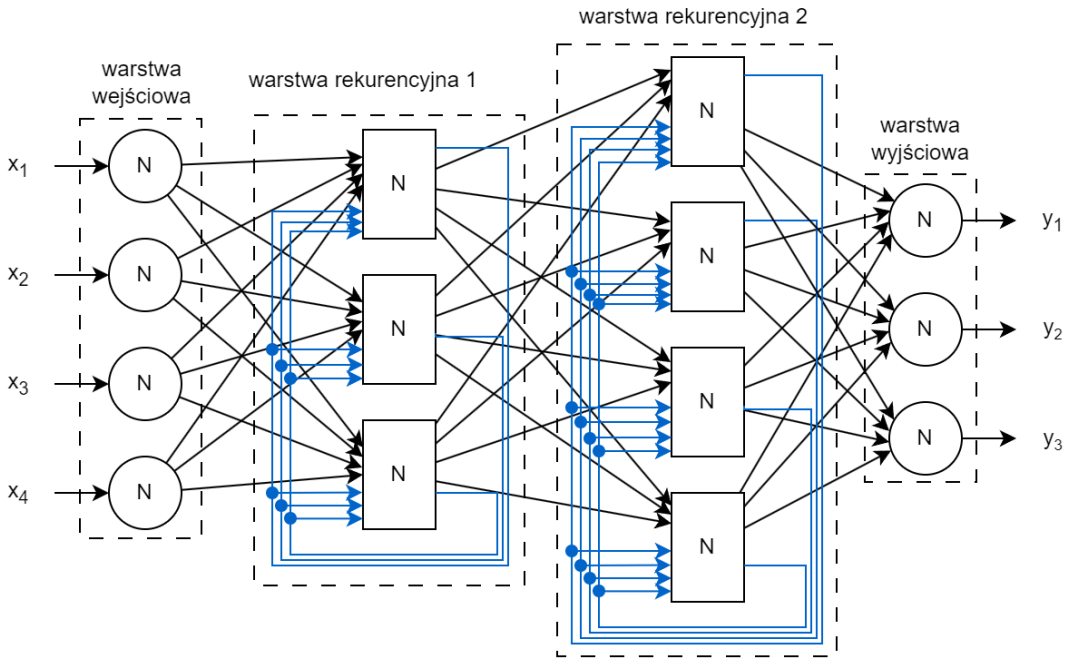
3.1. Wprowadzenie do sieci LSTM

Sieć neuronowa składa się z warstw, w których znajdują się pojedyncze elementy zwane neuronami. W przypadku struktur bardziej złożonych, jaką jest między innymi sieć LSTM, elementy te nazywa się komórkami. Przykładowy schemat sieci neuronowych znajduje się na rysunku 3.12, gdzie widać wyraźny podział na warstwy, akcentowany poprzez obrysy z przerywanymi liniami, oraz występujące w nich komponenty. Na przykładowym schemacie, który przedstawia jednokierunkową sieć neuronową, zaprezentowano cztery warstwy: wejściową, dwie warstwy ukryte oraz warstwę wyjściową. Liczba warstw może się znacząco różnić, np. sieć VGG16 posiada w swojej strukturze 21 warstw [1], natomiast ogólnie ich liczba może być znacznie większa. Elementy w obrębie sieci mogą być ze sobą różnorako połączone, przedstawiana na

rysunku 3.12 sieć jest jednokierunkowa, co oznacza, że połączenia występują pomiędzy kolejnymi warstwami, ze zwrotem skierowanym na następną warstwę. W przypadku łączenia elementów kolejnych warstw często stosuje się tzw. łączenie gęste, które oznacza połączenie każdego elementu z jednej warstwy z każdym elementem następującej po niej warstwy - co ukazano na rysunku 3.12.



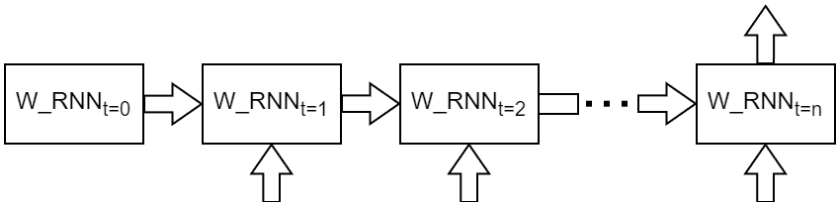
Rysunek 3.12: Schemat jednokierunkowej sieci neuronowej



Rysunek 3.13: Schemat rekurencyjnej sieci neuronowej

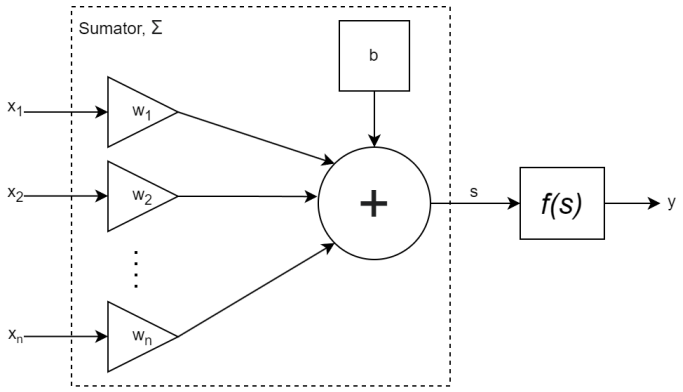
Poza jednokierunkowymi sieciami neuronowymi można wyróżnić rekurencyjne sieci neuronowe, których przykład przedstawiono na rysunku 3.13. Elementem wyróżniającym ten rodzaj sieci jest wystąpienie połączeń rekurencyjnych, które zaznaczone zostały na rysunku 3.13, kolorem niebieskim. Przedstawiona sieć składa się z dwóch warstw rekurencyjnych, wejściowej oraz wyjściowej. Liczba warstw i neuronów może się różnić, tak jak i rodzaje wykorzystanych warstw, bowiem sieć rekurencyjna nie musi składać się z samych ukrytych warstw. Połączenia rekurencyjne to połączenia doprowadzające sygnał z wyjścia elementu sieci z powrotem na wejście, co pozwala na wykorzystanie informacji z poprzedniego cyklu obliczeniowego w dalszych obliczeniach. Sygnały rekurencyjne są rozprowadzane w obrębie całej warstwy, co sprawia, że dostęp do informacji o stanie warstwy w poprzednim kroku obliczeniowym jest dostępny dla każdego elementu sieci w kolejnym kroku obliczeniowym. Takie podejście umożliwia wychwycenie zależności z danych występujących w obrębie analizowanych sekwencji. Przepływ sygnałów podczas analizy danych sekwencyjnych przedstawiono na rysunku 3.14.

W pierwszym kroku obliczeniowym (t=1) wykorzystywane są wartości początkowe, które zostały oznaczone jako stan warstwy dla t=0. Dopiero dla t=1 podawane są informacje z poprzedniej warstwy, a po wykonaniu obliczeń sygnały rekurencyjne są wystawiane na wyjściach elementów warstwy LSTM i wykorzystywane w następnym kroku obliczeniowym dla nowych wartości sygnałów z poprzedniej warstwy. Obliczenia w obrębie trwają tak długo, jak długa jest analizowana sekwencja. Na rysunku 3.14 strzałki poziome odpowiadają przepływowi sygnałów rekurencyjnych, a pionowe oznaczają przepływ sygnałów pomiędzy warstwami.



Rysunek 3.14: Schemat rekurencyjnej sieci neuronowej

Połączenia pomiędzy elementami sieci mają określone wagi, przez które przemnażane są sygnały wchodzące do poszczególnych neuronów/komórek, następnie są one sumowane, co pozwala uzyskać wartość liczbową, która jest dalej przetwarzana

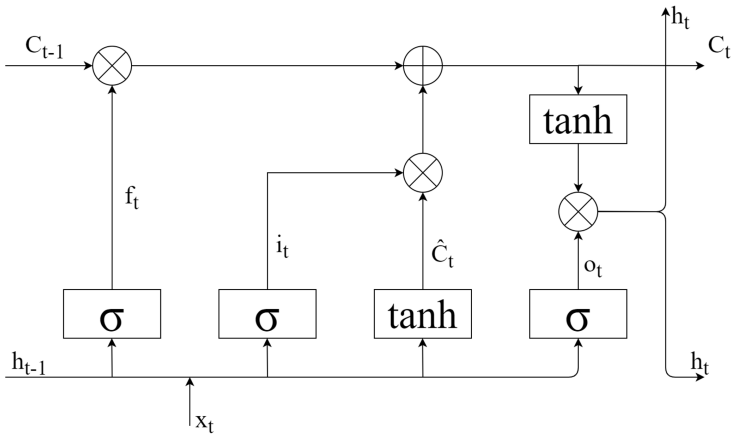


Rysunek 3.15: Schemat ogólny neuronu

(sygnał s na rysunku 3.15). Dodatkowo, poszczególne elementy mogą zawierać w sobie tzw. przesunięcie (ang. bias, b na rysunku 3.15), które jest wartością stałą dodawaną do wartości sumy. Dalsze przetwarzanie wartości z wejścia elementu odbywa się poprzez obliczenie wartości funkcji aktywacji, której argumentem jest sygnał s, czyli zsumowana wartość iloczynów z wejścia oraz przesunięcia. Można zatem, w najprostszym przypadku, funkcjonalnie wyróżnić sumator, który łączyłby w sobie operacje przemnażania przez odpowiednie wagi oraz sumowanie sygnałów i przesunięcia, oraz funkcję aktywacji, co zostało pokazane na rysunku 3.15. Funkcje aktywacji mogą być różne i podobnie jak w przypadku doboru rodzaju i liczby wykorzystywanych warstw lub elementów w tychże, są wyborem projektowym i mogą się znacząco różnić w zależności od zastosowania, a nawet w obrębie jednej sieci. Taki podział struktury wewnętrznej elementu sieci tj. wydzielenie sumatora oraz elementu przetwarzającego wartość z wyjścia sumatora, będzie stosowany w dalszej części pracy.

Sieć typu LSTM została zaprezentowana w latach 90-tych przez S. Hochreitera oraz J. Schmidhubera [14], jednak zyskała na popularności dopiero niedawno, po wygranym przez Alexa Gravesa konkursie rozpoznawania pisma ręcznego ICDAR w 2009 roku. Architektura LSTM należy do grupy sieci rekurencyjnych, co oznacza, że w ramach warstwy rekurencyjnej informacja z wyjścia jest zawracana z powrotem na wejście w następnym cyklu obliczeniowym. W obrębie sieci mogą występować również warstwy nierekurencyjne, które mogą być np. odpowiedzialne za analizę danych wychodzących z warstwy LSTM lub przygotowujące dane przed podaniem na warstwę LSTM [1, 74]. W jednej sieci może występować po sobie kaskadowo więcej niż jedna warstwa LSTM, a w niektórych przypadkach dodatkowa warstwa może działać rów-

nolegle z inną i analizować szereg wartości na wejściu warstwy od końca - biLSTM [73].



Rysunek 3.16: Schemat komórki LSTM

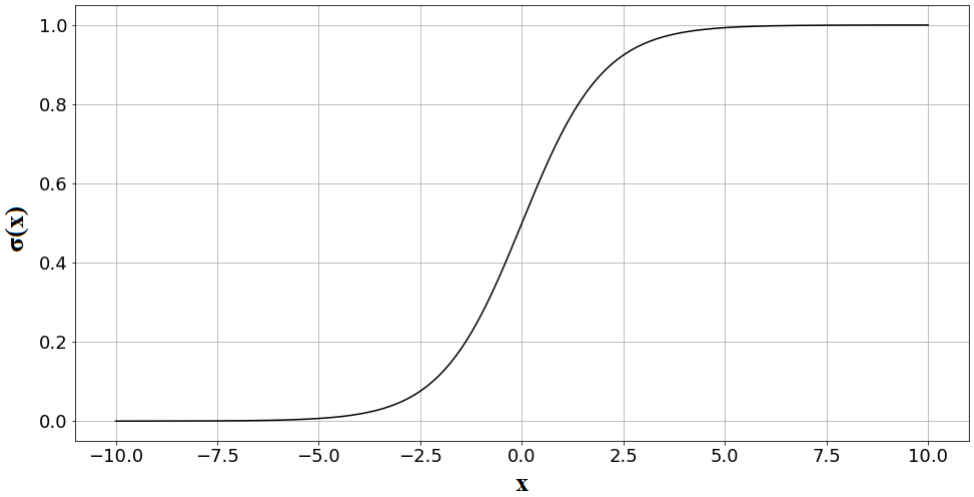
Podstawowym elementem składowym warstwy LSTM, jest komórka LSTM w której można wyróżnić następujące sygnały (rys. 3.16):

- $\mathbf{x_t}$ - wektor sygnałów z poprzedniej warstwy,
- $\mathbf{h_{t-1}}$ - wektor sygnałów wyjściowych z poprzedniego cyklu,
- $\mathbf{h_t}$ - wektor obecnych sygnałów wyjściowych,
- $\mathbf{C_{t-1}}$ - wektor stanów z poprzedniego cyklu,
- $\mathbf{\hat{C}_t}$ - wektor potencjalnie nowych stanów,
- $\mathbf{f_t}$ - wektor sygnałów ograniczających poprzedni stan,
- $\mathbf{i_t}$ - wektor sygnałów z bramki wejściowej,
- $\mathbf{o_t}$ - wektor sygnałów z bramki aktywującej wyjście.

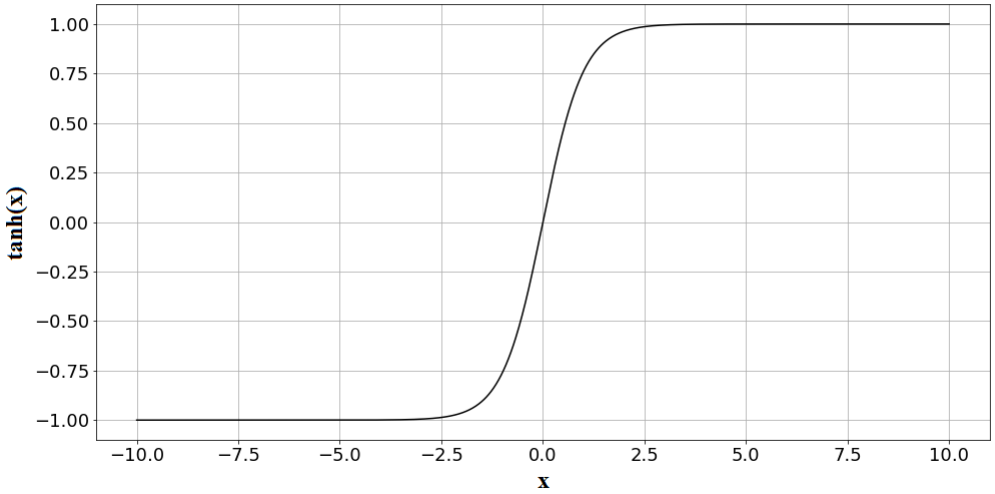
Przedstawione sygnały są wynikiem działania elementów składowych komórki - tak zwanych bramek:

- **bramka zapominająca** zdefiniowana jako (3.1),

$$f_t = \sigma(\sum_{n=0}^{n_i} v_{n,f} x_{n,t} + \sum_{m=0}^{m_j} w_{m,f} h_{m,t-1} + b_f) \tag{3.1}$$



Rysunek 3.17: Wykres funkcji sigmoidalnej



Rysunek 3.18: Wykres funkcji tangensa hiperbolicznego

- **bramka wejściowa** zdefiniowana jako (3.2),

$$i_t = \sigma(\sum_{n=0}^{n_i} v_{n,i} x_{n,t} + \sum_{m=0}^{m_j} w_{m,i} h_{m,t-1} + b_i) \tag{3.2}$$

- **bramka aktywacji wyjścia** zdefiniowana jako (3.3),

$$o_t = \sigma(\sum_{n=0}^{n_i} v_{n,o} x_{n,t} + \sum_{m=0}^{m_j} w_{m,o} h_{m,t-1} + b_o) \tag{3.3}$$

- **bramka aktywacji wejścia** zdefiniowana jako (3.4),

$$\hat{C}_t = \tanh(\sum_{n=0}^{n_i} v_{n,C} x_{n,t} + \sum_{m=0}^{m_j} w_{m,C} h_{m,t-1} + b_C) \tag{3.4}$$

- **bramka wyjściowa** korzysta z wyniku z (3.5) i jest zdefiniowana jako (3.6).

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C} \tag{3.5}$$

$$h_t = o_t \odot \tanh(C_t + b_C) \tag{3.6}$$

gdzie w przedstawionych równaniach: $\odot$ - iloczyn Hadamanda; $v_{i,j}$ - waga i-tego połączenia wejściowego do bramki j; $w_{i,j}$ - waga i-tego połączenia rekurencyjnego do bramki j; b_k - wartość stała dodawana do wejścia k-tej bramki (w pracy będzie określana mianem przesunięcia); σ - funkcja logistyczna (sigmoidalna), opisana wzorem (3.7), której wykres znajduje się na rysunku 3.17; tanh - funkcja tangensa hiperbolicznego, opisana wzorem (3.8), której wykres znajduje się na rysunku 3.18 [75].

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3.7}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.8}$$

W równaniach (3.1) - (3.4) występuje działanie: $\sum_{n=0}^{n_i} v_n x_n + \sum_{m=0}^{m_j} w_m h_m + b$, które funkcjonalnie odpowiada sumatorowi przedstawionemu na rysunku 3.15. Sumator jest częścią opisywanych bramek komórki LSTM i nie został wyszczególniony na rysunku 3.16, ponieważ w dalszych rozważaniach przedstawiona zostanie wersja komórki bez sumatora oraz z sumatorem, co zostanie zaznaczone w tekście.

3.2. Założenia implementacyjne

W ramach implementacji przedstawionych w niniejszej pracy poczyniono kilka wstępnych założeń:

- dokładność aproksymacji funkcji aktywacji dla tangensa hiperbolicznego przyjmuje się na poziomie 10^{-7} dla zakresu $< -6,6 >$ - założenie to zostanie zweryfikowane w punkcie 3.5.6 i 4.3.4;

- na podstawie dokładności aproksymacji funkcji tangensa hiperbolicznego porównane zostaną poszczególne podejścia implementacyjne funkcji aktywacji;

- ze względu na chęć podkreślenia wpływu zmiany platformy, na której wykonywane są obliczenia związane z siecią LSTM, przyjęto, że zarówno na FPGA jak i w programach przygotowanych w C/C++ wykorzystana zostanie arytmetyka zmiennoprzecinkowa pojedynczej precyzji zgodna z IEEE754.

3.3. Implementacja funkcji aktywacji

Zgodnie z założeniami w punkcie 3.2, w przedstawionych w pracy implementacjach funkcji aktywacji, docelowa wartość maksymalnego błędu bezwzględnego powinna wynosić 10^{-7} dla funkcji tangensa hiperbolicznego. Ustalenie docelowej wartości błędu ma na celu umożliwienie porównania różnych podejść implementacyjnych. Implementacje zawarte w tym rozdziale wykonane zostały dla układu Artix-7 XC7A100T-3CSG384 przy wykorzystaniu oprogramowania ISE Design Suite.

3.3.1. Powiązane prace dotyczące implementacji funkcji aktywacji na FPGA

W tym rozdziale zaprezentowany został stan nauki głównie w odniesieniu do funkcji aktywacji. W ostatnich latach wielu badaczy podjęło problematykę wykorzystania układów FPGA do wykonywania zadań związanych ze sztucznymi sieciami neuronowymi. Podstawowym elementem każdej sieci są funkcje aktywacji. Według [76] funkcje aktywacji są najważniejszą, najbardziej kosztowną i najtrudniejszą do implementacji częścią sztucznych sieci neuronowych. Autor [77] zwraca uwagę, że wysoka dokładność funkcji aktywacji jest szczególnie istotna w przypadkach implementacji uczenia sztucznych sieci neuronowych.

W [9] autorzy skupili się na wdrożeniu funkcji aktywacji dla sieci konwolucyjnych (CNN), którą w tym przypadku była TanhExp. Funkcja została podzielona na odcinki, które zostały aproksymowane. Wykorzystana tutaj została zarówno aproksymacja liniowa, jak i kwadratowa. W obu przypadkach rozważana dziedzina została podzielona na 13 przedziałów. Błąd jaki udało się uzyskać wyniósł 0.03105 oraz 0.00740 odpowiednio dla liniowej i kwadratowej aproksymacji, przy użyciu reprezentacji zgodnej z IEEE754. Pomimo, że implementacja osiąga stosunkowo duży błąd, to opóźnienie wprowadzane przez implementację wyniosło tylko 4 cykle.

Artykuł [78] proponuje implementację funkcji aktywacji w oparciu o metodę zwaną DCTIF (Discrete Cosine Transform Interpolation Filter), która przyjmuje jako parametr założoną dokładność i w zależności od niej reguluje liczbę próbek i punktów

pomiędzy nimi. Implementacja osiąga maksymalny błąd 0.004 i zużywa 1.45 Kb BRAM oraz 21 tablic LUT przy 8-bitowej długości słowa.

W artykule [79] przedstawiona została implementacja funkcji aktywacji dla klasyfikatorów bazujących na koncepcji ekstremalnego uczenia maszynowego (EML), w tym celu zaprezentowano trzy różne funkcje aktywacji tj. wartość stałą, tangens hiperboliczny oraz kosinus. Funkcja aktywacji w oparciu o wartości stałe polegała na tym, że dla argumentów powyżej 0, funkcja przyjmowała wartość 1, zaś dla tych poniżej wartość 0. Funkcje tangens hiperboliczny oraz kosinus były aproksymowane szeregiem Taylora w przedziale $< -1, 1)$, zaś poza tym zakresem przypisywana była wartość stała. Implementacja z użyciem szeregu Taylora pozwala na zoptymalizowanie architektury poprzez użycie rejestrów przesuwnych. Zużycie zasobów wyniosło 9 bloków DSP i 86 tablic LUT dla funkcji tangensa hiperbolicznego oraz 5 bloków DSP i 109 tablic LUT dla funkcji kosinus. Analiza błędu skupiała się na przedziale $< -1, 1)$ i została jedynie przedstawiona w formie wykresów bez podawania dokładnych wartości. Wartości odczytane z wykresów to 0.06 dla funkcji tangensa hiperbolicznego oraz 0.001 dla funkcji kosinus.

Inną publikacją, w której zastosowano podejście do aproksymacji funkcji aktywacji szeregiem Taylora jest [80]. Aproksymowaną funkcją była funkcja sigmoidalna określona w dziedzinie $< -8, 8 >$, którą podzielono na przedziały w zależności od dopuszczalnego błędu oraz maksymalnego stopnia wielomianu w szeregu Taylora. Rozważane były szeregi pierwszego i drugiego stopnia. Dla dopuszczalnego maksymalnego błędu 10^{-4} i dla wielomianów pierwszego stopnia, liczba interwałów wyniosła 102, zaś w przypadku wykorzystywania wielomianów drugiego stopnia 28.

Artykuł [81] prezentuje cztery podejścia do implementacji tangensa hiperbolicznego w strukturze FPGA: szereg Taylora, Elliott-2, Elliott-93 i wykorzystujące obliczenia trygonometryczne z wykorzystaniem algorytmu CORDIC oraz tablic LUT. Prezentowane podejścia korzystały z arytmetyki zmiennoprzecinkowej, zgodnej z IEEE754 oraz dotyczą zakresu $< -6, 6 >$. W przypadku wykorzystania szeregu Taylora aproksymowano funkcję eksponenty, która miała zostać wykorzystana do obliczeń funkcji aktywacji. Szereg Taylora ograniczono do wielomianów 5. stopnia. W przypadku wykorzystania algorytmu CORDIC obliczano najpierw wartości kosinusa hiperbolicznego oraz sinusa hiperbolicznego, którymi posługiwano się do obliczenia funkcji eksponenty. Analogicznie w przypadku algorytmów Elliott-2 i Elliott-93, również najpierw obliczano

funkcję eksponenty. W pracy posługiwano się m.in. średnim błędem bezwzględnym, który dla szeregu Taylora, Elliott-2, Elliott-93 i opartym o CORDIC wyniósł odpowiednio: 0.496668, 0.099109, 0.201963, $1.81 \cdot 10^{-6}$.

W [82] przedstawiono implementację opierającą się na aproksymacji funkcji eksponencjalnej, a następnie wykorzystaniu wzorów matematycznych do obliczenia właściwej funkcji aktywacji. Osiągnięto błąd 0.0177 przy zużyciu 103 przerzutników oraz 145 tablic LUT. Wynik zwracany był po 4 cyklach zegara.

W [83], jak i później w kontynuacji artykułu jako [84], przedstawiona jest metoda liniowo-odcinkowej aproksymacji funkcji sigmoidalnej oraz tangensa hiperbolicznego. Dziedzina została podzielona na 8 przedziałów równej długości, w których dokonywano aproksymacji. W przypadku funkcji sigmoidalnej dziedzina zawierała się w przedziale $< -8, 8 >$, natomiast w przypadku tangensa hiperbolicznego w przedziale $< -4, 4 >$. Maksymalny błąd aproksymacji (odczytany z wykresu) miał wartość $7.47 \cdot 10^{-3}$ dla funkcji sigmoidalnej oraz 0.017 dla tangensa hiperbolicznego. Autorzy dokonali również sprawdzenia wpływu reprezentacji liczbowej na błąd aproksymacji funkcji sigmoidalnej. Przy zachowaniu arytmetyki zmiennoprzecinkowej zmieniano długość słowa na: 8, 10, 12, 14, 16 bitów. Maksymalny błąd w przypadku reprezentacji 8-bitowej był około 1.5 razy większy niż w pozostałych przypadkach. Dla wartości 10, 12, 14 i 16 błąd nie zmieniał się znacząco.

Innym podejściem zaprezentowanym w [85] jest wykorzystanie tablicy LUT z 8192 elementami, zakodowanymi jako 16-bitowe liczby całkowite, do realizacji tangensa hiperbolicznego. Podobne podejście zastosowano w [86], gdzie rozszerzono rozważania o przypadek z liniową interpolacją pomiędzy punktami z tablicy LUT, która pozwoliła zredukować rozmiar tablicy LUT. Zastosowano tu arytmetykę stałopozycyjną o różnej liczbie bitów. Dla przypadku z 16 bitami odpowiadającymi części ułamkowej i tablicy LUT o wielkości $2 \cdot 10^{15}$ osiągano błąd maksymalny $1.6 \cdot 10^{-7}$ dla przypadku z zastosowaniem interpolacji liniowej oraz $1.2 \cdot 10^{-4}$ bez zastosowania interpolacji.

Alternatywne podejście zaprezentowano w pracy [87], które opiera się na analogowej realizacji funkcji aktywacji, zaimplementowanej w technologii CMOS, przeznaczonej do współpracy z systemami opartymi o memrystory. W przedstawianym rozwiązaniu dodatnią oraz ujemną część funkcji modelowano oddzielnie, a sygnałem było napięcie na rezystorze umieszczonym pomiędzy tranzystorem polowym a potencjałem masy. Wartość, z której liczona była funkcja, podawana była w postaci odpowiednio

przeskalowanego prądu doprowadzonego do wyprowadzenia tranzystora - w zależności od modelowanej połówki: dla ujemnej połówki do drenu, dla dodatniej do źródła. Autorzy postulują, że takie podejście do obliczeń może okazać się bardzo korzystne przy rozwinięciu techniki opartej o memrystory.

W [88] przedstawiono dwa podejścia do realizacji funkcji aktywacji. Oba podejścia opierały się na wyliczeniu najpierw wartości funkcji eksponenty, a w oparciu o nią wartości funkcji tangensa hiperbolicznego lub funkcji sigmoidalnej. W pierwszym podejściu wykorzystano interpolację McLaurina, a w drugim aproksymację Padego. W pracy przedstawiono wiele wariantów implementacyjnych osiągając przy tym błąd maksymalny na poziomie 10^{-7} przy zużyciu zasobów od 1885 do 2624 tablic LUT, od 792 do 1059 przerzutników FF, od 4 do 8 DSP, od 51 do 139 cykli potrzebnych do wykonania obliczeń oraz częstotliwości zegara od 88.5 do 99 MHz. W pracy wykonano też rzadko spotykane w literaturze porównanie narzędzi do generowania opisu sprzętu przygotowanego własnoręcznie. Do porównania wybrano narzędzia: Matlab HDL Coder oraz Vivado HLS. Kod wygenerowany przez oba narzędzia wypadł znacznie gorzej niż w przypadku samodzielnego kodowania. W przypadku Vivado HLS zużycie tablic LUT było 1.66 razy większe, zużycie przerzutników było 2.77 razy większe, zużycie bloków DSP było 8 razy większe, minimalny okres zegara 1.09 razy większy, a liczba cykli zegara 2.11 razy większa. W przypadku Matlaba przy włączonej opcji zwiększenia zużycia bloków DSP tablic LUT wykorzystano 3.29 razy więcej, przerzutników 1.01 razy mniej, bloków DSP 40 razy więcej, minimalny cykl zegara był 29.5 razy większy, ale za to liczba cykli potrzebnych na wykonanie obliczeń zmalała 9.67 razy, co i tak skutkowało trzykrotnie dłuższymi obliczeniami.

Aproksymacja funkcji eksponencjalnej została przedstawiona w [89], zastosowano tam reprezentację zmiennoprzecinkową pojedynczej precyzji, zgodną z IEEE754. Czas obliczeń wyniósł 11 cykli zegara, zaś maksymalny błąd $5.9623 \cdot 10^{-7}$. Implementacja wykorzystywała 2559 tablic LUT oraz 141 przerzutników. Praca nie przedstawia bezpośrednio implementacji funkcji aktywacji, ale implementacja taka może zostać w tym celu wykorzystana przy użyciu wzorów matematycznych.

W [90] przedstawiono implementację wykorzystującą 9 segmentów modelu SCPL oraz algorytm szarych wilków (GWO) do optymalizacji aproksymacji funkcji LogSig, TanSig oraz radialną funkcją bazową (RBF). Maksymalny błąd wyniósł $5.2 \cdot 10^{-3}$, $15.4 \cdot 10^{-3}$ oraz $7 \cdot 10^{-3}$ odpowiednio dla LogSig, TangSig oraz RBF. Autorzy podają,

że implementacja zużywa 581 tablic LUT, 9 bloków DSP i osiąga opóźnienie 35.346 ns przy 16-bitowej reprezentacji.

Artykuł [17] prezentuje podejście z aproksymacją liniową, w której głównym celem, według autorów, było osiągnięcie dokładności pozwalającej na zastosowanie na FPGA sieci pracujących na PC. W tym celu zastosowano logikę 32-bitową zgodną ze standardem IEEE754, zaś samą funkcję aproksymowano liniowo w 256 przedziałach. Maksymalny błąd wyniósł $2.18 \cdot 10^{-5}$.

W pracy [10] funkcja aktywacji została implementowana za pomocą interpolacji funkcjami sklejanymi osiągając błąd $7 \cdot 10^{-5}$. W publikacji nie podano szczegółów implementacyjnych.

W [91] przedstawiono implementacje funkcji tangensa hiperbolicznego oraz sigmoidalną, w oparciu o reprezentacje zmiennoprzecinkową pojedynczej i połówkowej precyzji. Analiza błędu nie została przeprowadzona, jednakowoż podano informację, iż dla pojedynczej precyzji osiągnięto błąd około $1.695 \cdot 10^{-7}$ podczas aproksymacji funkcji tangensa hiperbolicznego. Autorzy za to zauważają, że bezpośrednia aproksymacja funkcji aktywacji może znacząco zredukować ilość wykorzystywanych zasobów oraz przyspieszyć czas obliczeń, w przeciwieństwie do implementacji obliczających najpierw funkcję eksponenty.

Odmiennym ujęciem cechuje się artykuł [20], w którym przedstawiono implementację funkcji tangensa hiperbolicznego przy użyciu trzech metod: interpolacji Lagrange'a, metody najmniejszych kwadratów oraz interpolacji z użyciem wielomianów Czebyszewa (stopień najwyższego wielomianu został ograniczony do 3). Dla interpolacji Lagrange'a dziedzinę podzielono na 27 przedziałów, zaś dla pozostałych dwóch metod na 25 przedziałów. Osiągnięto odpowiednio błędy $1.935 \cdot 10^{-7}$ dla interpolacji Lagrange'a, $1.955 \cdot 10^{-7}$ dla metody najmniejszych kwadratów i $1.929 \cdot 10^{-7}$ dla interpolacji wielomianami Czebyszewa. Natomiast w pracy nie przedstawiono prób implementacji.

Proponowane w literaturze rozwiązania generują dobre rezultaty, pozostawiają również pole do dalszych rozważań. Kwestią otwartą są zagadnienia dotychczas niezbadane lub słabo zbadane, m.in. sprawdzenie wpływu uzyskanej wartości błędu na działanie systemu czy sprawdzenie różnych wariantów implementacyjnych. W wielu publikacjach informacja dotycząca szczegółów implementacyjnych nie została zawarta, co wiąże się z koniecznością przeprowadzania eksperymentów we własnym zakresie,

np. w trakcie projektowania własnej sieci.

Brak informacji dotyczącej sprzętowej implementacji funkcji aktywacji, w części podanej literatury przedmiotu, m.in. w artykule [92], utrudnia analizę wyników. W związku z tym istnieje potrzeba sprawdzenia i przetestowania wielu rozwiązań dotychczas albo niedostatecznie zbadanych albo niedostatecznie opisanych.

3.3.2. Implementacja wzorowana na algorytmie CORDIC

W niniejszym punkcie przedstawiono nowy sposób realizacji funkcji aktywacji. Podejście, podobnie jak w [82, 91, 93, 89], polega na obliczeniu wartości eksponenty, a następnie skorzystaniu ze wzoru (3.7) lub (3.8) w celu obliczenia wartości konkretnej funkcji. W przedstawionym jako listing 3.1 kodzie napisanym w języku Verilog obliczana jest wartości funkcji sigmoidalnej. W kod wbudowany jest algorytm obliczania eksponenty wzorowany na algorytmie CORDIC, który jest powszechnie używany przy wykonywaniu obliczeń trygonometrycznych np. generowaniu sygnału sinusoidalnego [94]. Aproksymację funkcji przeprowadzono w przedziale $< -6,6 >$, ponieważ poza tym zakresem funkcja nie zmienia się znacząco, a ze względu na budowę algorytmu większy zakres wiązałby się z koniecznością wykonania porównań najpierw dla $|x| > 6$. Listing 3.1 przedstawia część kodu wykorzystywaną do obliczeń, celowo z pominięciem głównego interfejsu modułu, bloków arytmetycznych oraz bloków z parametrami wykorzystywanymi podczas obliczeń, dla jasności zapisu i podkreślenia sposobu wykonywania obliczeń. Występujące w kodzie zmienne GEA, GEB oraz GEY są połączone z wyprowadzeniami bloku komparatora, GEA i GEB z jego wejściami, a GEY z wyjściem. Zmienne $ADDA$, $ADDB$, $ADDOP$ oraz $ADDY$ są połączone z interfejsem bloku dodająco-odejmującego, gdzie $ADDA$ i $ADDB$ są połączone z wejściami danych, $ADDOP$ jest połączone z wejściem określającym jaka operacja ma być wykonywana przez blok tj. czy dodawanie zmiennych, czy ich odejmowanie, a $ADDY$ z wyjściem z którego odczytywany jest wynik operacji. Zmienne $MULA$, $MULB$ oraz $MULY$ są połączone z wyprowadzeniami bloku mnożącego, gdzie $MULA$ i $MULB$ są połączone z wejściami danych, a $MULY$ wyjściem danych. Ostatnim blokiem arytmetycznym jest blok wykonujący operację dzielenia i z nim połączone są zmienne $DIVA$, $DIVB$ (wejścia danych) oraz $DIVY$ (wyjście danych). Poza blokami arytmetycznymi występują dwa bloki przechowujące dane i ustawiające zmienne k_value oraz k_exp, połączone z wyjściami bloków, w zależności od zmiennej i, która jest podawana na wejścia bloków.

Na początku obliczeń, po otrzymaniu sygnału potwierdzającego obecność poprawnych danych na wejściu tj. *i_data_valid*, zapisywana jest wartość bezwzględna z wejścia do zmiennej *Xtmp*, na której wykonywane są dalsze obliczenia. Inicjowana jest też tymczasowa zmienna wyjściowa *Ytmp*, której wartość na początku wynosi 1. Na komparator, który realizuje operację $GEY = GEA \geq GEB$, podawana jest wartość bezwzględna wejścia oraz wartość górnej granicy aproksymacji (w tym przypadku liczba 6). W stanie 1 sprawdzany jest wynik porównania. Jeśli wartość zmiennej wejściowej jest poza granicą aproksymacji, to zwracana jest stała wartość tj. 0 lub 1 w zależności od znaku zmiennej wejściowej. W przeciwnym razie obliczenia wykonywane są dalej. W każdej iteracji wartość zmiennej *Xtemp* jest porównywana z wartością logarytmu naturalnego z wartości liczonej w zależności od wartości iteratora. Gdy iterator jest mniejszy od 8, to logarytm naturalny liczony jest z wartości $\frac{256}{2^i}$, w przeciwnym wypadku wartość logarytmu liczona jest z wartości $1 + 2^{7-i}$. W celu zminimalizowania liczby wykonywanych działań i uproszczenia kodu, wartości logarytmu zostały obliczone wcześniej i umieszczone w kodzie, w postaci modułu ustawiającego *k_value*.

Listing 3.1. Obliczanie wartości funkcji sigmoidalnej

```
1. always @(posedge axi_clk)
2. begin
3.     if(axi_reset_n) begin ST<=0; end
4. end else
5. begin
6.     case(ST)
7.     0: begin
8.         i<=0;
9.         if(i_data_valid) begin
10.            GEA<={1'b0,i_data[30:0]}; GEB<=32'h40c00000; ST<=1;
11.            Ytmp<=32'h3f800000; Xtmp<={1'b0,i_data[30:0]}; ADDOP<=1;
12.            MULA<=32'h3f800000; MULB<=32'h3f800000;
13.            o_data_ready <= 0;
14.        end
15.    end
16.    1: begin
17.        if(GEY) begin
18.            Y <= i_data[31]?32'h00000000:32'h3f800000; ST<=0;
19.        end
```

```
20.          else begin
21.              ST<=2;
22.          end
23.      end
24.      2: begin
25.          GEA<=Xtmp; GEB<=k_value; ST<=3;
26.      end
27.      3: begin
28.          if(GEY) begin
29.              ADDA<=Xtmp; ADDB<=k_value;
30.              MULA<=Ytmp; MULB<=k_exp;
31.              ST<=4;
32.          end
33.          else begin
34.              if(i<30) begin i<=i+1; ST<=2; end
35.              else begin
36.                  ADDOP<=0; ADDA<=32'h3f800000; ADDB<=MULY; ST<=5;
37.              end
38.          end
39.      end
40.      4: begin
41.          Xtmp<=ADDY;
42.          Ytmp<=MULY;
43.          if(i<30) begin i<=i+1; ST<=2; end
44.          else begin
45.              ADDOP<=0; ADDA<=32'h3f800000; ADDB<=MULY; ST<=5;
46.          end
47.      end
48.      5: begin
49.          if(i_data[31]) begin DIVA<=32'h3f800000; end else begin DIVA<=Ytmp; 50. end
51.          DIVB<=ADDY; ST<=6;
52.      end
53.      6: begin o_data<=DIVY; o_data_ready<=1; ST<=0; end
54.      endcase
55. end
```

Jeśli zmienna $Xtmp$ jest większa od wartości logarytmu, to pomniejsza się jej wartość o wartość logarytmu i jednocześnie przemnaża się zmienną Y o wartość z ja-

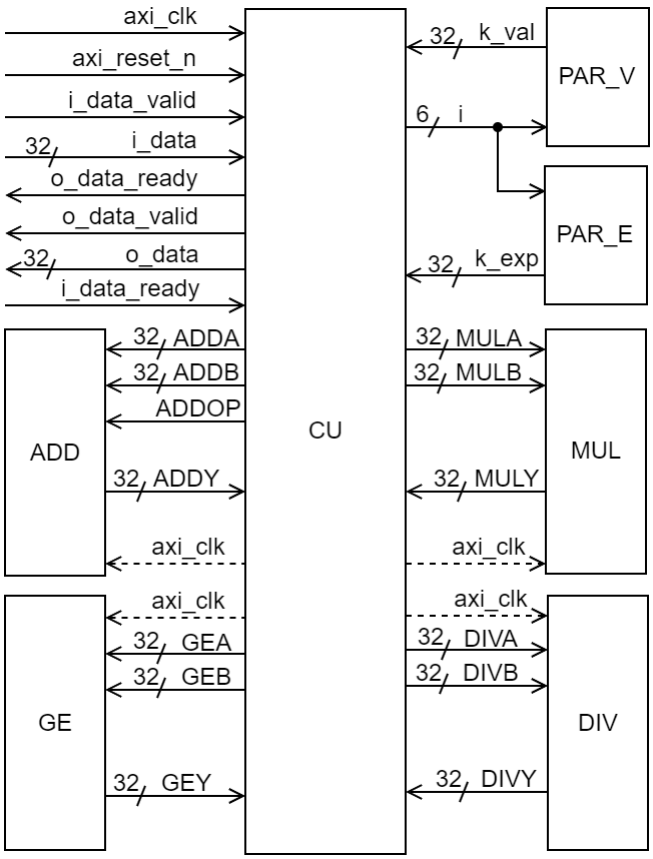
kiej liczony był logarytm, która również jest stabelaryzowana w module, który ustawia zmienną *k_exp*, z tych samych przyczyn, co wartości logarytmu. W przypadku tangensa hiperbolicznego obliczana jest zawsze dodatnia wartość funkcji według wzoru (3.8) i wykorzystywana jest własność symetrii funkcji względem początku układu współrzędnych. Gdy znak zmiennej wejściowej ma wartość 1, co oznacza wartość poniżej zera, zwracana jest wartość przeciwna do otrzymanego wyniku. W przypadku funkcji sigmoidalnej wykorzystuje się przekształcenie wzoru (3.7) w zależności od znaku zmiennej wejściowej, co widać w (3.9) oraz w stanie 5 na listingu 3.1 napisanym przy użyciu Veriloga (elementy języka Verilog zostały przedstawione w punkcie 2.3.3). Różnica jest w liczniku równania i dla wartości większych bądź równych zeru wykorzystuje się dwukrotnie wcześniej obliczoną wartość eksponenty, zaś dla wartości mniejszych od zera licznik ma wartość 1.

$$\sigma(x) = \begin{cases} \frac{e^x}{1+e^x},\ x \geq 0 \\ \frac{1}{1+e^x},\ x < 0 \end{cases} \tag{3.9}$$

Oznaczenia wykorzystywane na listingu 3.1 odpowiadają schematowi logicznemu implementacji, zaprezentowanemu na rysunku 3.19. Na schemacie znajdują się więc bloki ADD, GE, MUL i DIV, za którymi stoją operacje odpowiednio: dodawania-odejmowania, porównania dwóch liczb podawanych, mnożenia i dzielenia.

Poza wykorzystywanymi w kodzie wyprowadzeniami bloków, do każdego bloku doprowadzony był sygnał zegara, wykorzystywany w zależności od nastaw bloku. W przypadku obliczeń, przy nastawie na 0 cykli (konfiguracja w pełni kombinacyjna), sygnał taktujący nie był wykorzystywany - stąd został przedstawiony przerywaną linią. Podczas implementacji logiki, wartości wykorzystywane w trakcie obliczeń, tj. *k_val* oraz *k_exp*, zebrane zostały w blok PAR_V oraz PAR_E, które zwracały konkretną wartość, bazując na zmiennej i. Implementacja przedstawiona na rysunku 3.19 wyposażona jest dodatkowo w porty, zgodne z AXI4-Stream, wykorzystywane do komunikacji przez główny blok CU.

Liczba iteracji została ograniczona do 31 (od 0 do 30), co wynika z zakładanej w punkcie 3.2. dokładności. Maksymalny błąd dla różnej maksymalnej liczby iteracji został przedstawiony na rys. 3.20 dla tangensa hiperbolicznego oraz na rys. 3.21 dla funkcji sigmoidalnej. Każdy z punktów wyznaczony został jako wartość mak-

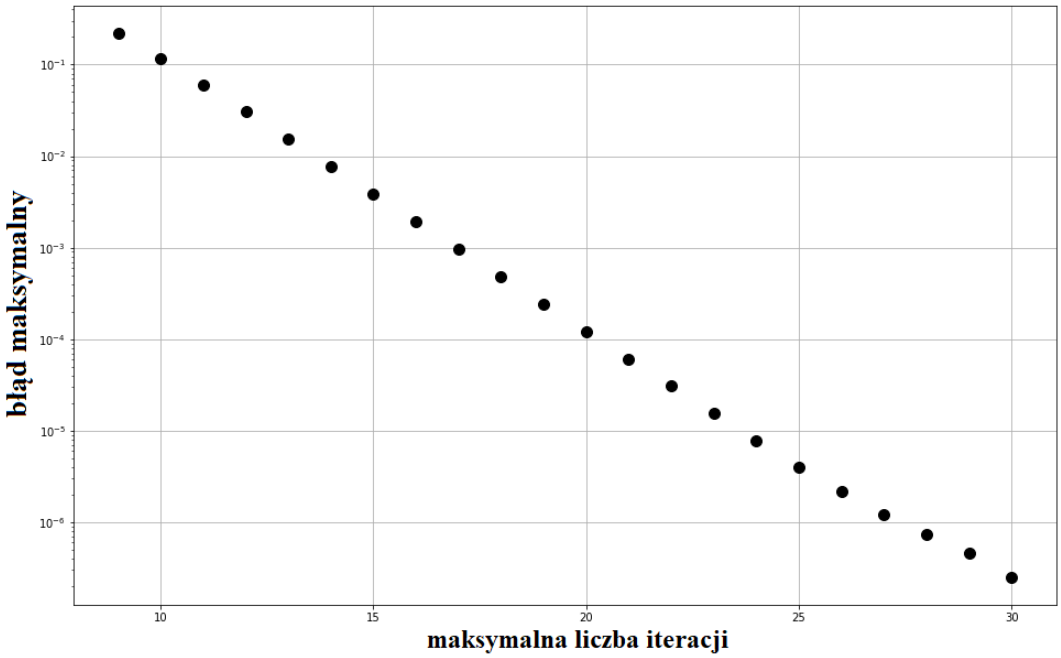


Rysunek 3.19: Schemat blokowy implementacji wzorowanej na algorytmie CORDIC

symalna błędu dla 1 mln punktów z przedziału od -6 do 6 i odniesiony do wartości obliczonych na komputerze klasy PC w programie napisanym w C++ i korzystającym z bibliotek standardowych. Dla funkcji sigmoidalnej wykres błędu stabilizuje się i nie schodzi poniżej $7.854 \cdot 10^{-7}$, zaś w przypadku tangensa hiperbolicznego spada niemal liniowo i dla i dochodzącego do 30, osiąga wartość $2.473 \cdot 10^{-7}$.

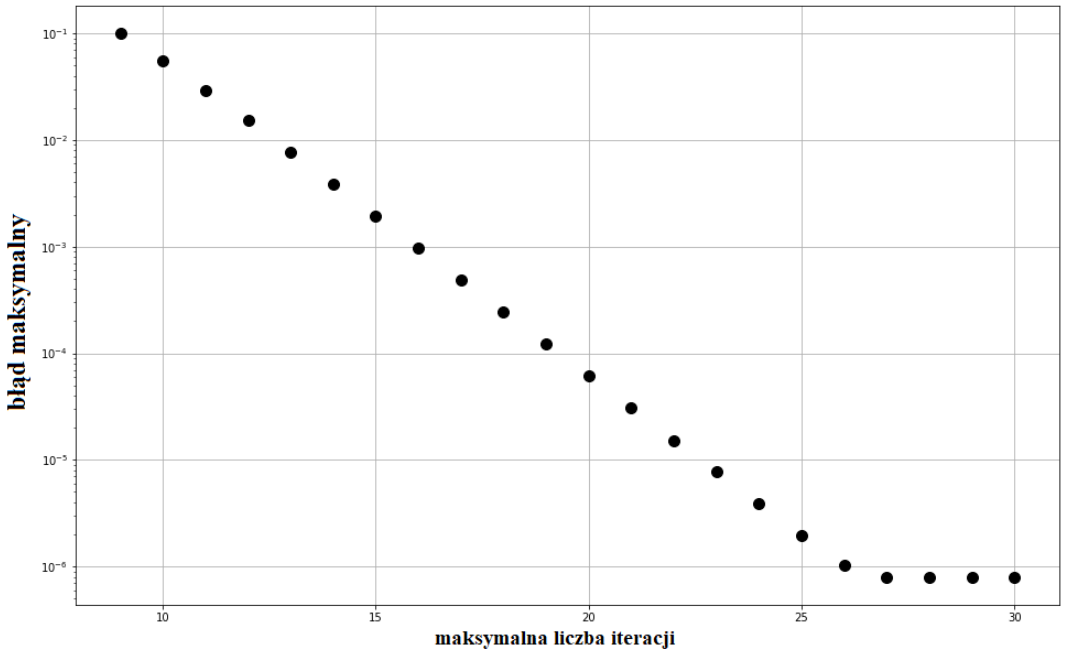
Wykresy wartości błędów dla ostatnich przypadków z rys. 3.20 oraz 3.21 (i=30), w zależności od wartości argumentu, znajdują się na rys. 3.22. Zostały one wyznaczone dla 1 tys. punktów z przedziału $< -6, 6 >$. Mniejsza liczba punktów testowych była motywowana czytelnością wykresów. Można zauważyć, że dla funkcji sigmoidalnej dla większości dziedziny błąd maksymalny jest mniejszy niż $8 \cdot 10^{-8}$, lecz wzrasta w przedziale $|x| \in < 4.15, 4.85 >$. Dla tangensa hiperbolicznego wartości błędu są zbliżone w obrębie całej dziedziny, zaś maksimum lokuje się w okolicach zera.

Pierwotnie implementacja dotyczyła jednego modułu realizującego obie funkcje aktywacji. Funkcję wybierało się poprzez odpowiednie ustawienie jednego z wejść modułu. Implementacja potrzebowała 189 cyklów zegara do obliczenia wyniku, zaj-



Rysunek 3.20: Dokładność funkcji tangensa hiperbolicznego w zależności od liczby iteracji

mowała 5107 tablic LUT, 288 przerzutników. Wykorzystując 5 bloków DSP, zużycie tablic LUT spadło do 4373. Obliczona za pomocą programu ISE Design Suite częstotliwość zegara wyniosła 12.8 MHz. Taka implementacja została przedstawiona przez autora w pracy [95]. Implementacja ta została później poprawiona, co pozwoliło na zmniejszenie wykorzystywanych zasobów oraz zwiększenie maksymalnej częstotliwości taktowania. Realizowane funkcje aktywacji, zostały rozdzielone na oddzielne moduły (tj. oddzielny moduł realizujący obliczenia dla funkcji sigmoidalnej i oddzielny dla funkcji tangensa hiperbolicznego). Poprawiona implementacja funkcji tangensa hiperbolicznego wykorzystała 1305 tablic LUT, 262 przerzutników i 4 DSP, zaś obliczona częstotliwość zegara wyniosła 33.0 MHz. Poprawiona implementacja funkcji sigmoidalnej zajęła natomiast 1298 tablic LUT, 253 przerzutników, 4 DSP, obliczona częstotliwość zegara wynosiła 34.6 MHz. Zmniejszono również liczbę cykli potrzebnych na obliczenie wartości funkcji oraz zrezygnowano z ujednolicenia liczby tejże w zależności od wartości wejściowej. Ze względu na to, że część działań przedstawiona na listingu 3.1 jest wykonywana warunkowo, to w przypadku pierwotnej implementacji, w trakcie

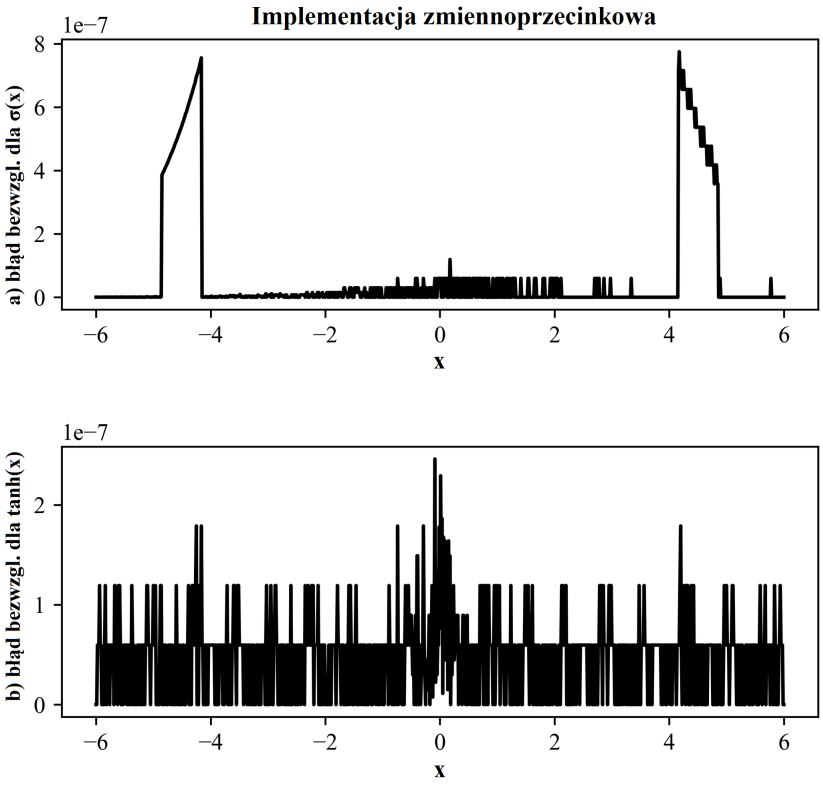


Rysunek 3.21: Dokładność funkcji sigmoidalnej w zależności od liczby iteracji

obliczeń wymagających mniej cykli zegara, czekano odpowiednią liczbę cykli. W poprawionej implementacji w zależności od wartości wejściowej liczba cykli jest różna, co prezentuje histogram na rys. 3.23. Największą częstość obserwuje się w okolicach 74 cykli. Maksymalna wartość to 80 cykli, zaś minimalna 67. Omawiana implementacja wykorzystywała bloki arytmetyczne wykonujące układy kombinacyjne, co pozwala na wykorzystanie wyników już w następnym takcie zegara. Inne nastawy bloków arytmetycznych zostały również rozważone, wyniki zawarto w tabeli 3.1.

Tabela 3.1. Wyniki implementacji dla różnych nastaw bloków arytmetycznych (Artix-7 XC7A100T-3CSG384)

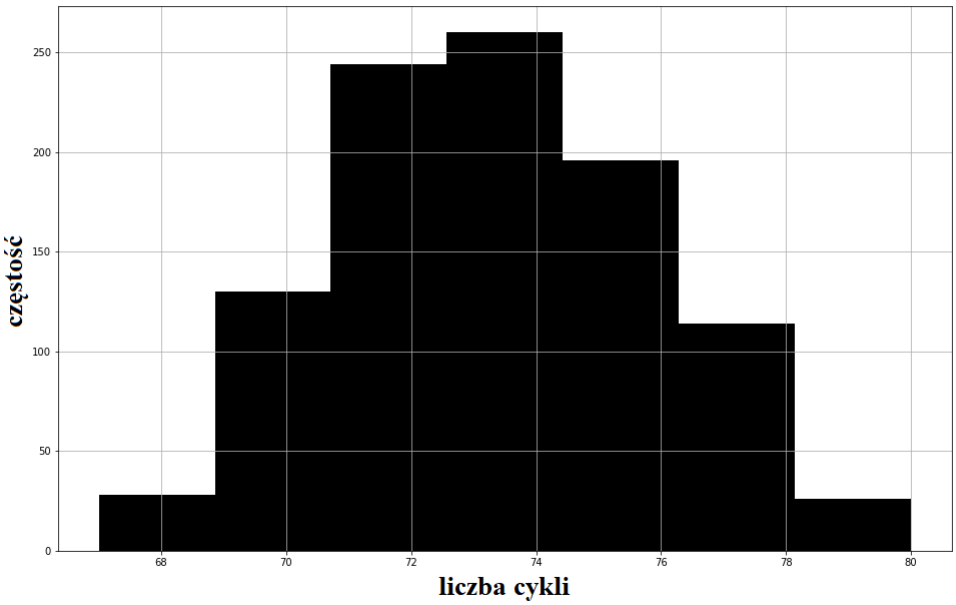
Liczba cykli na operację	LUT	FF	DSP	Liczba taktów zegara	Maksymalna częstotliwość zegara [MHz]
0	1305	262	4	$67-80\,(Rys.\,2.7)$	33.0
1	1302	288	4	$108-133$	34.2
2	1339	421	4	$149-186$	35.5



Rysunek 3.22: Błąd aproksymacji funkcji dla reprezentacji zmiennoprzecinkowej, metodą wzorowaną na CORDIC

Ustawienie bloków arytmetycznych na opóźnienie o 1 cykl lub 2 nie wykazało polepszenia wyników, czy to w zapotrzebowaniu na zasoby sprzętowe, czy w maksymalnej częstotliwości zegara. Większych wartości nie rozważano ze względu na maksymalną liczba cykli w module komparatora wynoszącą 2. Wyniki wstępnej implementacji oraz po poprawie przedstawiono w tabeli 3.2, gdzie poprawioną wersję oznaczono jako ‘nowa‘.

Stosowanie implementacji zmiennoprzecinkowych sprawia, że obliczenia są bardziej odporne np. na przypadkowe przepełnienie, mogą również poprawić dokładność (co dzieje się w omawianym podejściu). Wykorzystanie arytmetyki zmiennoprzecinkowej często wiąże się dłuższym czasem obliczeń, co sprawia, że reprezentacje stałoprzecinkowe są równie chętnie stosowane. W związku z tym dalej przedstawiona została alternatywna implementacja z użyciem reprezentacji stałoprzecinkowej, która koresponduje z pierwotną wersją implementacji zmiennoprzecinkowej (przed poprawą). Rozmiar słowa wyniósł 32 bity, gdzie najstarszy bit jest bitem znaku. Metoda obli-

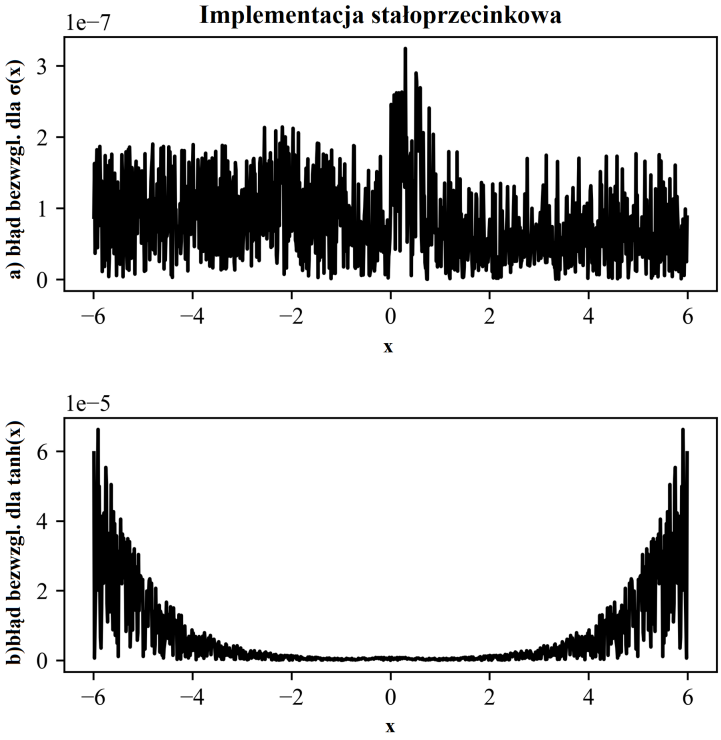


Rysunek 3.23: Histogram liczby cykli potrzebnych do obliczenia wartości funkcji tangensa hiperbolicznego

czeniowa jest analogiczna do tej z listingu 3.1, gdzie obliczenia opierają się na wyliczeniu wartości eksponenty dla argumentów z przedziału $< -6,6 >$. W implementacji założono, że w każdym punkcie wykorzystana będzie jednakowa liczba bitów, reprezentacja liczbowa musi zatem pomieścić liczby do e^6. Współczynnik konwersji wyniósł więc 5323079 (3.10), co pozwoliło na osiągnięcie rozdzielczości $1.8786 \cdot 10^{-7}$. Operacje arytmetyczne w tym przypadku są w pełni kombinacyjne.

$$coef = int(\frac{2^{31}}{e^6}) = 5323079 \tag{3.10}$$

Do sporządzenia wykresów z rys. 3.24 wygenerowano 1000 punktów testowych w zakresie $< -6,6 >$, dla których obliczono wartości funkcji sigmoidalnej oraz tangensa hiperbolicznego. Błąd liczony był względem programu referencyjnego na PC napisanego w C++, korzystającego z bibliotek standardowych i jedynie przeskalowującego wartości. Przedstawione rozwiązanie potrzebowało 65 cykli zegara na obliczenie wartości obu funkcji aktywacji oraz wykorzystało 10465 tablic LUT oraz 178 przerzutników. Dokładność dla funkcji sigmoidalnej wyniosła $3.112 \cdot 10^{-7}$ zaś dla tangensa hiperbolicznego $6.505 \cdot 10^{-5}$ i w obu przypadkach wartości te były obliczone dla 10^6



Rysunek 3.24: Błąd aproksymacji funkcji dla reprezentacji stałoprzecinkowej, metodą wzorowaną na CORDIC

punktów. Dla reprezentacji stałoprzecinkowej poprawionej implementacji nie wykonano. Zbiorcze wyniki implementacji przedstawiono w tabeli 3.2.

Tabela 3.2. Wyniki implementacji (Artix-7 XC7A100T-3CSG384)

Funkcja	Wersja	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Maksymalna częstotliwość zegara [MHz]
$\sigma/\tanh$	$float$	5107	288	0	$7.854 \cdot 10^{-7}/2.473 \cdot 10^{-7}$	189	12.8
$\sigma/\tanh$	$float$	4373	288	5	$7.854 \cdot 10^{-7}/2.473 \cdot 10^{-7}$	189	13.1
$\sigma/\tanh$	int	10465	178	0	$3.112 \cdot 10^{-7}/6.505 \cdot 10^{-5}$	65	36.2
σ	$nowa; float$	1298	253	4	$7.854 \cdot 10^{-7}$	$65-78$	34.6
$\tanh$	$nowa; float$	1305	262	4	$2.473 \cdot 10^{-7}$	$67-80\,(Rys.2.7)$	33.0

3.3.3. Implementacja z użyciem wielomianów Czebyszewa

W podrozdziale zaprezentowano implementację funkcji aktywacji w oparciu głównie o funkcję tangensa hiperbolicznego z użyciem wielomianów Czebyszewa pierwszego rodzaju. Funkcja ta jest symetryczna względem początku układu współrzędnych tj. $\tanh(-x) = -\tanh(x)$, więc wystarczy zaimplementować funkcję dla dodatniej części dziedziny, a dla argumentów z ujemnej części zmieniać znak wartości obliczonej dla wartości bezwzględnej. Funkcję sigmoidalną zbudowaną w oparciu o przedstawioną metodę i wykorzystywaną w dalszych rozważaniach zaimplementowano w analogiczny sposób z tą różnicą, że do osiągnięcia symetrii przesunięto funkcję o wartość stałą -0.5 przez co, poza obliczeniem wartości aproksymacji, należało dodać wartość 0.5 do wyniku.

Wielomiany Czebyszewa pierwszego rodzaju są powszechnie stosowanymi wielomianami ortogonalnymi, opisanymi rekurencyjnie (3.11-3.13), gdzie pojedynczy wielomian jest opisany jako T_n [20].

$$T_0(x) = 1 \tag{3.11}$$

$$T_1(x) = x \tag{3.12}$$

$$T_n(x) = 2 \cdot x \cdot T_{n-1}(x) - T_{n-2}(x) \tag{3.13}$$

Szereg Czebyszewa tworzy się według (3.14), gdzie każdy z wielomianów jest mnożony przez swój współczynnik, a następnie sumuje się otrzymane wartości w celu określenia wartości interpolowanej funkcji. Liczba wielomianów (tym samym stopień najwyższego) dopasowuje się w zależności od założonych kryteriów.

$$f(x) = \sum_{n=0}^{k} c_n \cdot T_n(x) \tag{3.14}$$

W celu uproszczenia obliczeń szereg może zostać przekształcony do układu równań za pomocą algorytmu Clenshawa, skutkuje to zmniejszeniem czasu obliczeń oraz ilości zużytych zasobów, gdyż wielokrotne wykonywanie operacji potęgowania dla każdego z wielomianów, a następnie sumowanie wartości każdego z wielomianów, jest zastąpione kilkoma operacjami mnożenia, dodawania i odejmowania. Dla przykładu przedstawiono układ równań (3.15-3.20) dla szeregu ograniczonego do 5. stopnia wielomianu Czebyszewa.

$$ct_2 = 2 \cdot x \tag{3.15}$$

$$d_4 = ct_2 \cdot c_5 + c_4 \tag{3.16}$$

$$d_3 = ct_2 \cdot d_4 - c_5 + c_3 \tag{3.17}$$

$$d_2 = ct_2 \cdot d_3 - d_4 + c_2 \tag{3.18}$$

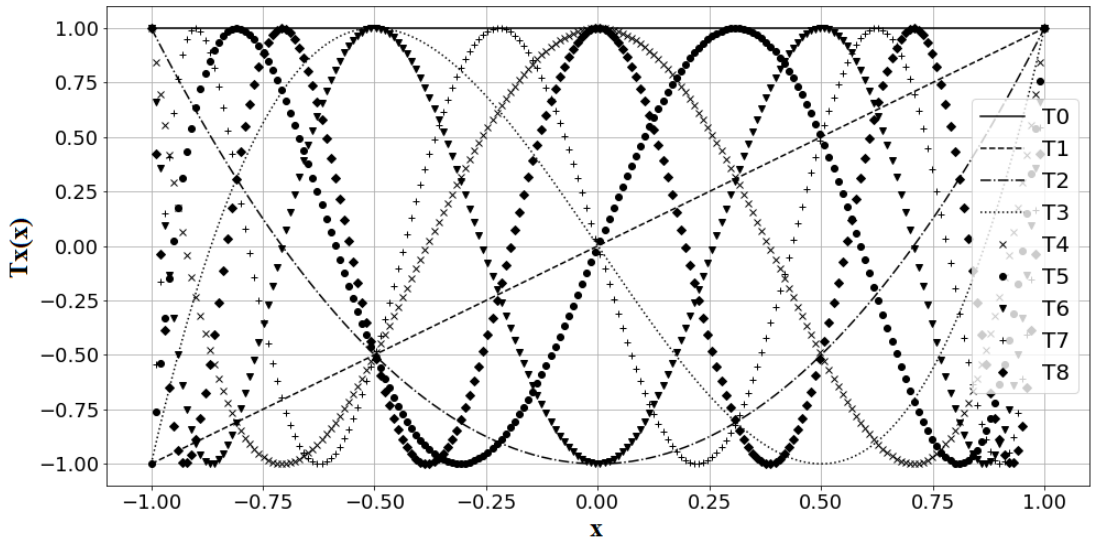
$$d_1 = ct_2 \cdot d_2 - d_3 + c_1 \tag{3.19}$$

$$y = x \cdot d_1 - d_2 + c_0 \tag{3.20}$$

Wielomiany Czebyszewa określone są w przedziale $< -1,1 >$ (rys. 3.25), dlatego niezbędne jest skalowanie wartości argumentu aproksymowanej funkcji, zgodnie z (3.21).

$$x = 2 \cdot \frac{x_{in} - a}{b - a} - 1 = \frac{2 \cdot x_{in} - a - b}{b - a}, \tag{3.21}$$

gdzie: a, b - granice przedziału; $a < b$.



Rysunek 3.25: Wykres pierwszych dziewięciu wielomianów Czebyszewa

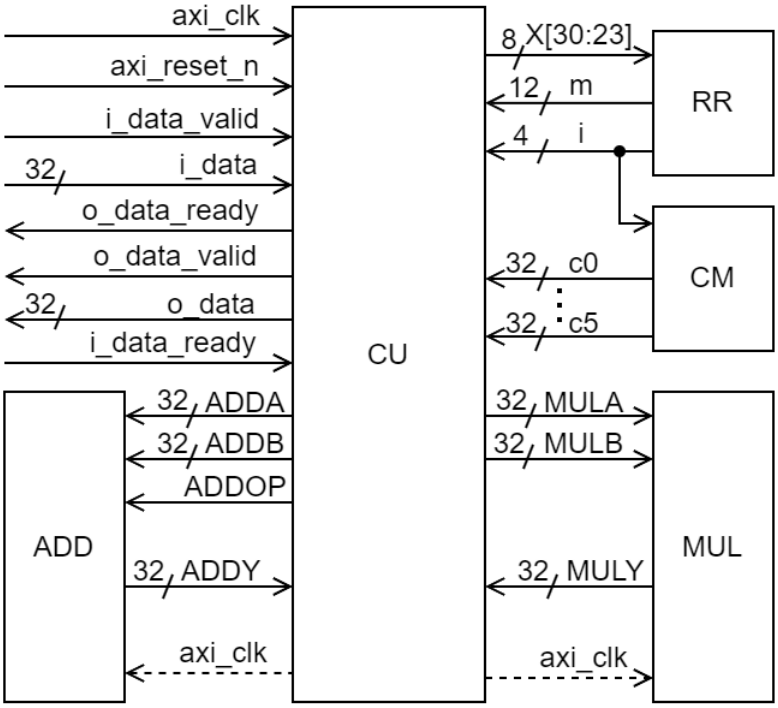
W celu uzyskania dużej dokładności, złożone funkcje m.in. tangens hiperboliczny, wymagają podziału na odcinki, które będą oddzielnie interpolowane. Im więcej przedziałów, tym wyższa dokładność, ale zarazem większe zużycie zasobów FPGA.

Wybór liczby przedziałów oraz ich długość powinny zatem być dokładnie rozważone. Wykorzystanie przedziałów będących potęgą liczby 2 (dodatnią lub ujemną) pozwala na zastąpienie operacji dzielenia przesunięciem bitowym. W wyniku przeprowadzenia kilku wstępnych eksperymentów obliczeniowych, ustalono jako punkt wyjścia 15 następujących przedziałów: $[0, 0.25)$, $[0.25, 0.5)$, $[0.5, 0.75)$, $[0.75, 1.0)$, $[1.0, 1.25)$, $[1.25, 1.5)$, $[1.5, 2.0)$, $[2.0, 2.5)$, $[2.5, 3.0)$, $[3.0, 3.5)$, $[3.5, 4.0)$, $[4.0, 5.0)$, $[5.0, 6.0)$, $[6.0, 8.0)$, $[8.0, 10.0)$. W wyniku przedstawionego podziału dziedziny powstało 15 szeregów dla których należało zapisać współczynniki. Dla wartości argumentu powyżej 10 zwracana była stała wartość funkcji, zaś dla argumentów poniżej 0 wykorzystana została własność symetrii funkcji względem początku układu współrzędnych (funkcja sigmoidalna aproksymowana będzie jako funkcja przesunięta o -0.5 względem OY). W prezentowanej implementacji, funkcjonalność detekcji przedziału została zamknięta w oddzielnym bloku (blok *RR* na rysunku 3.26), który odpowiada zarówno za wystawienie 4-bitowego sygnału i, którego wartość odpowiada numerowi przedziału, oraz sygnału określającego wartość 12-bitowej zmiennej $m = -a - b$, która była wykorzystywana do dopasowania funkcji do przedziału według wzoru (3.21).

Tabela 3.3. Blok detekcji przedziału

Zakres zmiennej X	X[30:23]	X[22]	X[21]	i	m
$0 \leq \|X\| < 0.25$	$\leq 0x7C$	–	–	0	$0xBE8$
$0.25 \leq \|X\| < 0.5$	$\leq 0x7D$	–	–	1	$0xBF4$
$0.5 \leq \|X\| < 0.75$	$\leq 0x7E$	0	–	2	$0xBFA$
$0.75 \leq \|X\| < 1.0$	$\leq 0x7E$	1	–	3	$0xBFE$
$1.0 \leq \|X\| < 1.25$	$\leq 0x7F$	0	0	4	$0xC01$
$1.25 \leq \|X\| < 1.5$	$\leq 0x7F$	0	1	5	$0xC03$
$1.5 \leq \|X\| < 2.0$	$\leq 0x7F$	1	–	6	$0xC06$
$2.0 \leq \|X\| < 2.5$	$\leq 0x80$	0	0	7	$0xC09$
$2.5 \leq \|X\| < 3.0$	$\leq 0x80$	0	1	8	$0xC0B$
$3.0 \leq \|X\| < 3.5$	$\leq 0x80$	1	0	9	$0xC0D$
$3.5 \leq \|X\| < 4.0$	$\leq 0x80$	1	1	10	$0xC0F$
$4.0 \leq \|X\| < 5.0$	$\leq 0x81$	0	0	11	$0xC11$
$5.0 \leq \|X\| < 6.0$	$\leq 0x81$	0	1	12	$0xC13$
$6.0 \leq \|X\| < 8.0$	$\leq 0x81$	1	–	13	$0xC16$
$8.0 \leq \|X\| < 10.0$	$\leq 0x82$	0	0	14	$0xC19$
$\|X\| \geq 10.0$	pozostałe przypadki			15	$0x0$

Detekcja przedziału odbywała się na podstawie 10 bitów sygnału wejściowego, a wartości zmiennej m są wcześniej obliczone i przedstawione w tabeli 3.3. Dwa bity sygnału wejściowego czasem nie są brane pod uwagę - "−" w tabeli 3.3, ponieważ przedział jest określony przez bity starsze i przypisanie do przedziału jest niezależne od tych wartości.



Rysunek 3.26: Schemat blokowy implementacji przy zastosowaniu zarówno zwykłych wielomianów, jak i wielomianów Czebyszewa

Poza blokiem detekcji przedziału komunikującym się bezpośrednio z blokiem CU, który odpowiada za zarządzanie obliczeniami oraz obsługę interfejsów (w tym głównego interfejsu zgodnego z AXI4-Stream), na rysunku 3.26 można wyróżnić jeszcze bloki arytmetyczne ADD i MUL oraz blok CM. Sekwencyjny charakter obliczeń, będących implementacją równań (3.16)-(3.20), przedstawiony na listingu 3.3, który został napisany przy użyciu Veriloga (elementy języka Verilog zostały przedstawione w punkcie 2.3.3), pozwolił na ograniczenie zużycia modułów obliczeniowych do jednego układu mnożącego i jednego dodająco-odejmującego. Blok dodająco-odejmujący komunikuje się z blokiem głównym za pośrednictwem sygnałów $ADDA$, $ADDB$, $ADDOP$ oraz $ADDY$. Sygnały $ADDA$ i $ADDB$ są wejściami danych bloku, $ADDOP$ określa wykonywaną operację (0 - dodawanie, 1 - odejmowanie), zaś $ADDY$ jest wyjściem

danych z bloku. Analogicznie wygląda komunikacja bloku mnożącego, gdzie sygnały $MULA$ i $MULB$ stanowią wejścia danych do bloku, zaś sygnał $MULY$ wyjście danych. Do obu bloków dochodzi również sygnał zegarowy, który jest wykorzystywany w zależności od nastawy bloków na wykonywanie operacji w konkretną liczbę cykli zegara. W przypadku nastawy na 0, co odpowiada układom w pełni kombinacyjnym, sygnał zegarowy nie jest wykorzystywany - dlatego na schemacie zaznaczony jest linią przerywaną. Ostatnim z występujących bloków jest blok CM, w którym zapisane są współczynniki dla każdego z przedziałów. Na rysunku 3.26 przedstawiona jest sytuacja, gdy rozpatrywana jest zmiennoprzecinkowa implementacja dla wielomianów Czebyszewa piątego stopnia, stąd na schemacie występują sygnały $c0 - c5$. Tego samego przypadku dotyczy implementacja przedstawiona na listingu 3.3. Sekwencja czynności umożliwiająca wyliczenie współczynników wielomianu Czebyszewa jest następująca:

1. Oblicz węzły zgodnie z (3.22).
2. Przeskaluj węzły do interpolowanego odcinka funkcji zgodnie z (3.23).
3. Oblicz wartości funkcji dla węzłów i zapisz jako y.
4. Oblicz współczynniki zgodnie z (3.24).

Liczba węzłów musi być maksymalnie równa liczbie wielomianów, czyli liczbie o 1 większej od stopnia najwyższego wielomianu.

$$r_k = -cos(\frac{2k-1}{2t} \cdot \pi),\ \ dla\ k = 1, ..., t;\ t-1 \geq n \tag{3.22}$$

$$z = (r+1) \cdot \frac{b-a}{2} + a,\ \ gdzie:\ a, b - granice\ przedziału;\ \ a\ < b \tag{3.23}$$

$$c_i = \frac{\sum_{k=1}^{m} y_k \cdot T_i(r_k)}{\sum_{k=1}^{m} T_i(r_k)^2} \tag{3.24}$$

Listing 3.2. Deklaracja zmiennych pomocniczych

```
1. wire [7:0] e2=i_data[30:23]+1;
2. wire [7:0] ee2=ADDY[30:23]+2;
3. wire [7:0] ee1=ADDY[30:23]+1;
4. wire [7:0] ee3=ADDY[30:23]+3;
5. wire [7:0] eem1=ADDY[30:23]+8'b1111_1111;
```

Listing 3.3. Obliczanie wartości funkcji tangensa hiperbolicznego przy użyciu wielomianów Czebyszewa

```
1. always @(posedge axi_clk)
2. if(axi_reset_n) ST<=0;
3. else
4. case(ST)
5. 0: begin
6.       RDY<=0;
7.       if(i_data_valid) begin XI<=i_data; ST<=3; end
8. end
9. 1: begin ST<=2; end
10. 2: begin ST<=STR; end
11. 3: begin
12.       ADDB<={m,20'h00000}; ADDA<={1'b0,e2,XI[22:0]};
13.       ADDOP<=0; riv<=i;
14.       if(iv==15) begin
15.           ST<=15;
16.       end else begin
17.           ST<=1; STR<=4;
18.       end
19. end
20. 4: begin
21.       if(riv<=5) begin
22.           nx<={ADDY[31],ee2,ADDY[22:0]}; ct2<={ADDY[31],ee3,ADDY[22:0]};
23.           MULB<={ADDY[31],ee3,ADDY[22:0]};
24.       end else if(riv>5&&riv<=10) begin
25.           nx<={ADDY[31],ee1,ADDY[22:0]}; ct2<={ADDY[31],ee2,ADDY[22:0]};
26.           MULB<={ADDY[31],ee2,ADDY[22:0]};
27.       end else if(riv==13||riv==14) begin
28.           nx<={ADDY[31],eem1,ADDY[22:0]}; ct2<={ADDY[31],ADDY[30:0]};
29.           MULB<={ADDY[31],ADDY[30:0]};
30.       end else begin
31.           nx<=ADDY; ct2<={ADDY[31],ee1,ADDY[22:0]}; MULB<={ADDY[31],ee1,ADDY[22:0]};
32.       end
33.       MULA<=c5;
34.       ST<=1; STR<=5;
35. end
36. 5: begin
37.       ADDA<=c4; ADDB<=MULY; ADDOP<=0; STR<=6; ST<=1;
38. end
```

```
6: begin
    d4<=ADDY; MULA<=ADDY; MULB<=ct2; ADDA<=c3; ADDB<=c5; ADDOP<=1;
    ST<=1; STR<=7;
end
7: begin
    ADDA<=MULY; ADDB<=ADDY; ADDOP<=0; ST<=1; STR<=8;
end
8: begin
    d3<=ADDY;
    MULA<=ADDY; MULB<=ct2; ADDA<=c2; ADDB<=d4; ADDOP<=1;
    ST<=1; STR<=9;
end
9: begin
    ADDA<=MULY; ADDB<=ADDY; ADDOP<=0; ST<=1; STR<=10;
end
10: begin
    d2<=ADDY;
    MULA<=ADDY; MULB<=ct2; ADDA<=c1; ADDB<=d3; ADDOP<=1;
    ST<=1; STR<=11;
end
11: begin
    ADDA<=MULY; ADDB<=ADDY; ADDOP<=0; ST<=1; STR<=12;
end
12: begin
    MULA<=nx; MULB<=ADDY; ADDA<=c0; ADDB<=d2; ADDOP<=1;
    ST<=1; STR<=13;
end
13: begin
    ADDA<=MULY; ADDB<=ADDY; ADDOP<=0; ST<=1; STR<=14;
end
14: begin
    o_data<=XI[31]? ADDY[31],ADDY[30:0]:ADDY; o_data_ready<=1; ST<=0;
end
15: begin
    o_data<=i_data[31]?32'hbf800000:32'h3f800000; o_data_ready<=1; ST<=0;
end
endcase
```

Wykorzystanie wspomnianego parametru m do dopasowania do przedziału ma miejsce w linii 12 na listingu 3.3. Ze względu na rozmiar zmiennej m, wynoszący 12 bitów, przed podaniem na wejście układu dodająco-odejmującego wartość ta jest uzupełniana w operatorze sklejania. Na drugie wejście podawana jest wartość z wejścia i_data przemnożona razy 2. W celu szybkiego przemnożenia wartość wejściowa rozbijana jest na oddzielne elementy: znak, wykładnik oraz mantysę, i wykorzystuje się zmienną $e2$ jako nową wartość wykładnika. Zmienna $e2$ odpowiada wartości wykładnika sygnału wejściowego zwiększonego o 1 (równoważne mnożeniu razy 2), co przedstawiono na listingu 3.2 w linii 1.

Mianownik z równania (3.21) jest szerokością przedziału i jest uwzględniony w stanie 4 na listingu 3.3 (linie 21-34) poprzez rozkład wyniku dodawania składników licznika oraz odpowiednią zmianę samego wykładnika liczby, a następnie zapisanie wyniku jako znormalizowaną wartość wejściową nx oraz podwojoną wartość wejścia $ct2$. Przedstawione i wykorzystywane w liniach 22-31 zmienne $ee3$, $ee2$, $ee1$ i $eem1$ są określone na listingu 3.2, a ich wykorzystanie zależy od rozmiaru rozpatrywanego przedziału. Informacja na temat rozmiaru jest powiązana z numerem przedziału zwracanym z bloku detekcji przedziału RR. Dla przedziałów o długości 0.25, $i < 6$ wykładnik jest powiększany o $j = 2$, co odpowiada mnożeniu razy 4. Dla przedziałów o długości 0.5, $6 \leq i < 10$ wykładnik powiększany jest o $j = 1$, co odpowiada mnożeniu razy 2, zaś dla przedziałów o długości 2, $i = \{13, 14\}$ - $j = -1$, co odpowiada dzieleniu przez 2. Dla przedziałów o długości 1 nie wykonuje się operacji na wykładniku i wtedy $j = 0$.

Tabela 3.4. Wyniki implementacji z zastosowaniem wielomianów Czebyszewa dla arytmetyki stałoprzecinkowej U2 (Artix-7 XC7A100T-3CSG384)

Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba cykli	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
8	771	331	8	$5.595 \cdot 10^{-11}$	20	202.6	98.7
7	677	295	8	$2.141 \cdot 10^{-10}$	18	171.6	104.9
6	633	247	4	$2.973 \cdot 10^{-9}$	16	122.9	130.2
5	619	253	4	$1.951 \cdot 10^{-8}$	14	106.2	131.8
4	548	255	4	$2.482 \cdot 10^{-7}$	12	90.7	132.3
3	497	252	4	$1.177 \cdot 10^{-5}$	10	75.2	132.9
2	462	216	4	$2.699 \cdot 10^{-4}$	8	61.0	131.2

W tabeli 3.4 przedstawiono wyniki implementacji stałoprzecinkowej dla różnych stopni wielomianów oraz kodowania U2, zaś w tabeli 3.5 dla kodowania zmiennoprzecinkowego. Przy zastosowaniu arytmetyki stałoprzecinkowej operacje na wykładniku zamienione zostały na przesunięcia bitowe. W obu przypadkach kodowania, dla stopni 2-6, część ułamkowa w słowie kodowym miała długość 31 bitów zarówno dla wyjścia, jak i dla współczynników. W przypadku stopnia 7. część ułamkowa wynosiła 35 bitów dla wyjścia i dla współczynników, zaś w przypadku 8. stopnia wyjście Y miało 35 bitów, a współczynniki 38. Zwiększenie rozmiaru części ułamkowej poskutkowało zwiększeniem zapotrzebowania na DSP z 4 na 8. Wykorzystanie zwykłego kodowania spowodowało wystąpienie problemu z przepełnianiem się sumatora dla stopni 4, 7 i 8. Obsłużenie tego problemu spowodowało zwiększenie zużycia zasobów sprzętowych, lecz pozwoliło na utrzymanie dobrych wyników dokładności obliczeń. Dla pozostałych stopni problem nie wystąpił i nie było potrzeby wykorzystywania dodatkowej logiki. Operacje arytmetyczne w trakcie eksperymentów wykonywane były kombinacyjnie i wynik dostępny był już w następnym cyklu zegara. Na tym etapie pomiar dokładności wykonany został dla 1 mln próbek z przedziału $< -10, 10 >$ ze względu na prezentowanie wyników częściowo opublikowanych przez autora w [96].

Wyniki dla przedziału zgodnego z założeniami niniejszej pracy tj. $< -6, 6 >$ zostały przedstawione w dalszej części podrozdziału dla arytmetyki zmiennoprzecinkowej.

Tabela 3.5. Wyniki implementacji z zastosowaniem wielomianów Czebyszewa dla arytmetyki stałoprzecinkowej (Artix-7 XC7A100T-3CSG384)

Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
8	1055	334	8	$8.750 \cdot 10^{-11}$	20	199.3	100.3
7	844	267	8	$2.158 \cdot 10^{-10}$	18	194.2	92.7
6	684	269	4	$3.890 \cdot 10^{-9}$	16	137.6	116.3
5	665	285	4	$1.925 \cdot 10^{-8}$	14	123.3	113.5
4	830	244	4	$2.540 \cdot 10^{-7}$	12	106.6	112.5
3	571	162	4	$1.200 \cdot 10^{-5}$	10	77.0	129.8
2	550	173	4	$2.700 \cdot 10^{-4}$	8	61.0	131.2

Wraz ze wzrostem stopnia wielomianu zauważalnie wzrasta dokładność, kosztem wykorzystywania większej liczby zasobów oraz dłuższym czasem obliczeń. Dla 8. stopnia wielomianu potrzebne jest 20 cykli zegara na otrzymanie wyniku dla obu wariantów kodowania. Kodowanie U2 potrzebuje nieco więcej czasu ze względu na niższą maksymalną częstotliwość zegara, ale osiąga nieco lepszą dokładność i ma mniejsze zapotrzebowanie na zasoby sprzętowe. Zmniejszanie stopnia maksymalnego wielomianu zauważalnie wpływa na dokładność i liczbę potrzebnych cykli zegara. Zmniejszenie o jeden stopień powoduje zmniejszenie czasu obliczeń o dwa cykle zegara, co jest związane z redukcją układu równań, a tym samym redukcją rozmiaru automatu sekwencyjnego o dwa stany. Idąc od 8. stopnia można zaobserwować, że dokładność maleje niemal o rząd wielkości na stopień maksymalnego wielomianu. W przypadku przejścia z 4 na 3 są to 2 rzędy wielkości.

Tabela 3.6. Wyniki implementacji z zastosowaniem wielomianów Czebyszewa dla arytmetyki zmiennoprzecinkowej (Artix-7 XC7A100T-3CSG384)

Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
7	899	355	4	$1.192 \cdot 10^{-7}$	46	408.66	112.5
6	829	350	4	$1.192 \cdot 10^{-7}$	40	329.72	121.3
5	757	353	4	$1.192 \cdot 10^{-7}$	34	290.77	116.9
4	744	347	4	$2.980 \cdot 10^{-7}$	28	247.80	112.9
3	692	323	4	$1.180 \cdot 10^{-5}$	22	193.03	113.9
2	566	288	4	$2.700 \cdot 10^{-4}$	16	135.01	118.5

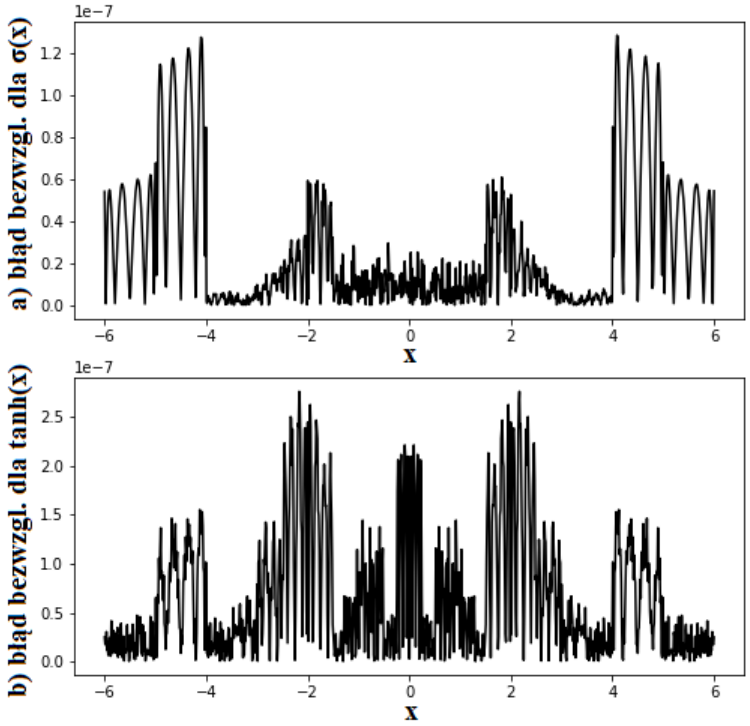
W tabeli 3.6. przedstawiono wyniki implementacji dla arytmetyki zmiennoprzecinkowej zgodnej z IEEE754. Podobnie jak poprzednio pomiary dokładności wykonano dla 1 mln próbek z przedziału $< -10, 10 >$, a pomiary porównawcze, dla przedziału $< -6, 6 >$, zostały przedstawione w dalszej części podrozdziału oraz w punkcie 3.3.5. Operacje arytmetyczne w tym przypadku wykonywane były w 2 cyklach zegara, co pozwoliło na osiągnięcie wyższych częstotliwości maksymalnych zegara oraz zredukowanie zapotrzebowania na zasoby sprzętowe. Pozostałe konfiguracje zostały przedstawione w tabeli 3.7 dla 5. stopnia maksymalnego wielomianu. Zauważyć można, że obniżenie liczby cykli zegara potrzebnych na wykonanie obliczeń zmiennoprzecinko-

wych, znacząco obniża maksymalną częstotliwość zegara, co wpływa na częstotliwość taktowania konfiguracji, w której funkcja aktywacji może zostać osadzona - stanowi to podstawę do odrzucenia mniejszych liczb cykli na operację niż 2. Zwiększenie liczby operacji powyżej 2 nie wpływa na polepszenie parametrów implementacji.

Tabela 3.7. Wpływ różnych nastaw bloków arytmetycznych (Artix-7 XC7A100T-3CSG384)

Liczba cykli na operację	LUT	FF	DSP	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
0	1332	238	2	13	230.9	56.3
0	756	236	4	13	216.5	60.1
1	1299	315	2	23	377.7	60.8
1	734	269	4	23	356.8	64.4
2	1342	447	2	34	321.7	105.6
2	757	353	4	34	290.7	116.9
3	1361	508	2	45	292.4	153.8
3	764	322	4	45	288.6	155.8
4	941	416	2	56	272.6	205.4
4	754	359	4	56	302.5	185.1

Zauważyć można również, że zmiana strategi generowania bloków arytmetycznych, z zastosowania 4 bloków DSP zamiast 2, zauważalnie redukuje liczbę pozostałych wykorzystywanych zasobów sprzętowych oraz podwyższa maksymalną częstotliwość zegara. Największa osiągnięta dokładność w tabeli 3.6 wynosi $1.192 \cdot 10^{-7}$ i może być osiągnięta przy 34 cyklach. Również dokładność dla 4. stopnia odpowiada założeniom osiągając dokładność $2.980 \cdot 10^{-7}$ przy 28 cyklach i nieco mniejszym zapotrzebowaniu na sprzęt, dlatego też właśnie implementacja dla 4. stopnia była wykorzystywana w dalszych eksperymentach. Ze względu na brak przyrostu dokładności wraz ze wzrostem stopnia wielomianu powyżej 5. stopnia, zaniechano eksperymenty z 8. stopniem. Zmniejszenie stopnia wielomianu poskutkowało jednocześnie zmniejszeniem dokładności, ale również krótszym czasem obliczeniowym oraz mniejszym zużyciem zasobów sprzętowych. Największy błąd osiągnięto dla 2. stopnia wielomianu i wyniósł $2.700 \cdot 10^{-4}$, natomiast obliczenia zajęły 16 cykli, czyli ponad 2 razy mniej niż dla 5. stopnia.



Rysunek 3.27: Błąd aproksymacji funkcji dla reprezentacji zmiennoprzecinkowej, metodą opartą o wielomiany Czebyszewa

Na rys. 3.27 przedstawiono wykres błędu bezwzględnego aproksymacji dla funkcji sigmoidalnej oraz tangensa hiperbolicznego dla implementacji z wielomianami 5. stopnia. Wykres przygotowano dla 1000 punktów testowych w przedziale od $< -6, 6 >$. W przypadku funkcji tangensa hiperbolicznego stosunkowo dużo argumentów osiąga wartości powyżej połowy największej wartości błędu. W przypadku funkcji sigmoidalnej przeważa mała wartość błędu. Widoczne jest zwiększenie wartości błędu w przedziale $|x| \in < 4, 5 >$, w którym znajduje się również wartość maksymalna. Implementacja funkcji sigmoidalnej, jak wspomniano na początku podrozdziału, jest niemal taka sama jak funkcji hiperbolicznej. Różnica leży we współczynnikach oraz konieczności dodania wartości 0.5 do wyniku. Dodatkowa operacja dodawania wydłuża obliczenia poprzez rozszerzenie automatu sekwencyjnego o 3 cykle zegara i zmienia zużycie zasobów do 730 LUT, 323 FF oraz osiąga dokładność $6.008 \cdot 10^{-7}$ dla 1 mln próbek z przedziału $< -10, 10 >$ oraz $5.811 \cdot 10^{-7}$ dla 1 mln próbek z przedziału $< -6, 6 >$, co zamieszczono w tabeli 3.12. w punkcie 3.3.5.

3.3.4. Implementacja z użyciem aproksymacji klasycznymi wielomianami

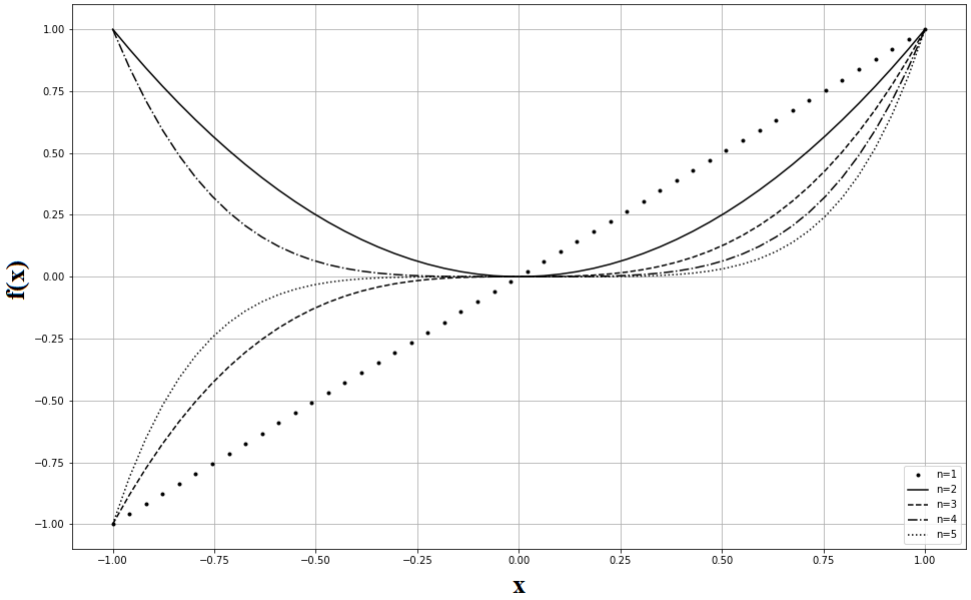
Obok implementacji funkcji aktywacji z użyciem wielomianów Czebyszewa oraz algorytmu wzorowanego na CORDIC, zaprezentowanych w poprzednich punktach, w niniejszym zaprezentowano implementację z użyciem zwykłych wielomianów, w której analogicznie skorzystano z symetrii względem początku układu współrzędnych i aproksymowano część funkcji dla $x \geq 0$, zaś dla $x < 0$ zmianie ulegał znak wyniku. Komplementarna funkcja sigmoidalna dla zwykłych wielomianów została zaimplementowana podobnie jak w przypadku wielomianów Czebyszewa tj. poprzez przesunięcie aproksymowanej funkcji o -0.5, a następnie dodawanie wartości 0.5 do wyniku aproksymacji. W omawianym podejściu implementacja polega na wstępnym przeskalowaniu wartości argumentu zgodnie z (3.21) do przedziału $< -1, 1 >$, a następnie obliczenia wartości wielomianu zgodnie z algorytmem Hornera przedstawionym jako (3.25).

$$f(x) = ((...(a_n \cdot x + a_{n-1}) \cdot x + ... + a_2) \cdot x + a_1) \cdot x + a_0 \tag{3.25}$$

gdzie $a_0 ... a_n$ - współczynniki wielomianu.

Obliczanie wartości według tego algorytmu pozwala na zminimalizowanie liczby wykonywanych operacji arytmetycznych. Na rys. 3.28 przedstawiono wielomiany od stopnia 1 do 5 dla rozważanego zakresu.

Skalowanie zmiennej wejściowej oraz sekwencyjny charakter obliczeń przypomina podejście przedstawione w punkcie 3.3.3. Analogicznie, w tym przypadku potrzebny był jeden układ mnożący i jeden układ dodający, a całość obliczeń dla przypadku 4. stopnia wielomianu została ujęta w automacie sekwencyjnym i została przedstawiona jako listing 3.4, napisany przy użyciu Veriloga (elementy języka Verilog zostały przedstawione w punkcie 2.3.3). Opisywanej implementacji odpowiada taki sam schemat logiczny, jak w punkcie 3.3.3 tj. z rysunku 3.26. Jedyną różnicą jest zawartość bloków CM i CU. Wstępna liczba i długość przedziałów jest taka sama jak w przypadku implementacji z zastosowaniem wielomianów Czebyszewa. Tutaj również blok RR na podstawie 10 najstarszych bitów sygnału wejściowego dokonuje detekcji przedziału i zwraca jednocześnie parę parametrów: i oraz m, w oparciu o tabelę 3.2, według których przeskalowywana jest wartość wejściowa i_data oraz wybierana jest wartość współczynników wielomianu. Do obliczenia wartości wielomianu wykorzystywane jest równanie (3.25), gdzie zmienna wejściowa po przeskalowaniu jest początkowo prze-



Rysunek 3.28: Wykres pierwszych pięciu wielomianów w zakresie $< -1, 1 >$

mnażana przez współczynnik dla najwyższej potęgi, a następnie do wyniku dodawany jest kolejny współczynnik dla drugiej najwyższej potęgi. Wynik znów przemnażany jest przez przeskalowaną wartość zmiennej wejściowej. Schemat się powtarza aż do dodania ostatniego współczynnika.

Listing 3.4. Obliczanie wartości funkcji tangensa hiperbolicznego przy użyciu klasycznych wielomianów

```
1. always @(posedge axi_clk)
2. if(axi_reset_n) ST<=0;
3. else
4. case(ST)
5. 0: begin
6.       RDY<=0;
7.       if(ND) begin XI<=i_data; ST<=3; end
8. end
9. 1: begin ST<=2; end
10. 2: begin ST<=STR; end
11. 3: begin
12.       ADDB<={m,20'h00000}; ADDA<={1'b0,e2,XI[22:0]}; ADDOP<=0; riv<=i;
13.       if(iv==15) begin
```

```
14.          ST<=15;
15.      end else begin
16.          ST<=1; STR<=4;
17.      end
18.      end
19. 4: begin
20.      if(riv<=5) begin nx<={ADDY[31],ee2,ADDY[22:0]}; ct2<={ADDY[31],ee3,ADDY[22:0]};
21.          MULB<={ADDY[31],ee2,ADDY[22:0]}; end
22.      else
23.      if(riv>5&&riv<=10) begin nx<={ADDY[31],ee1,ADDY[22:0]};
24.          ct2<={ADDY[31],ee2,ADDY[22:0]}; MULB<={ADDY[31],ee1,ADDY[22:0]}; end
25.      else
26.      if(riv==13||riv==14) begin nx<={ADDY[31],eem1,ADDY[22:0]};
27.          ct2<={ADDY[31],ADDY[30:0]}; MULB<={ADDY[31],eem1,ADDY[22:0]}; end
28.      else begin nx<=ADDY; ct2<={ADDY[31],ee1,ADDY[22:0]}; MULB<=ADDY; end
29.      MULA<=c5;
30.      ST<=1; STR<=5;
31. end
32. 5: begin
33.      ADDA<=c4; ADDB<=MULY; ADDOP<=0; STR<=6; ST<=1;
34. end
35. 6: begin
36.      MULA<=ADDY; MULB<=nx; ST<=1; STR<=7;
37. end
38. 7: begin
39.      ADDA<=MULY; ADDB<=c3; ADDOP<=0; ST<=1; STR<=8;
40. end
41. 8: begin
42.      MULA<=ADDY; MULB<=nx; ST<=1; STR<=9;
43. end
44. 9: begin 45.      ADDA<=MULY; ADDB<=c2; ADDOP<=0; ST<=1; STR<=10;
46. end
47. 10: begin
48.      MULA<=ADDY; MULB<=nx; ST<=1; STR<=11;
49. end
50. 11: begin
51.      ADDA<=MULY; ADDB<=c1; ADDOP<=0; ST<=1; STR<=12;
52. end
```

```
53. 12: begin
54.     MULA<=ADDY; MULB<=nx; ST<=1; STR<=13;
55. end
56. 13:begin
57.     ADDA<=MULY; ADDB<=c0; ADDOP<=0; ST<=1; STR<=14;
58. end
59. 14: begin
60.     o_data<=XI[31]?{ ADDY[31],ADDY[30:0]}:ADDY; o_data_ready<=1; ST<=0;
61. end
62. 15: begin
63.     o_data<=i_data[31]?32'hbf800000:32'h3f800000; o_data_ready<=1; ST<=0;
64. end
65. endcase
```

W tabeli 3.8 przedstawiono wyniki implementacji dla arytmetyki stałoprzecinkowej. Analogicznie jak w przypadku implementacji z użyciem wielomianów Czebyszewa, dla stopni 2-6, część ułamkowa w słowie kodowym miała długość 31 bitów zarówno dla wyjścia, jak i dla współczynników. W przypadku stopnia 7. część ułamkowa wynosiła 35 bitów dla wyjścia i dla współczynników, zaś w przypadku 8. stopnia wyjście Y miało 35 bitów, a współczynniki 38 bitów.

Tabela 3.8. Wyniki implementacji z zastosowaniem wielomianów dla arytmetyki stałoprzecinkowej (Artix-7 XC7A100T-3CSG384)

Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
8	690	161	8	$6.733 \cdot 10^{-11}$	18	175.9	102.3
7	656	159	8	$4.020 \cdot 10^{-10}$	16	157.7	101.4
6	558	152	4	$4.600 \cdot 10^{-9}$	14	105.0	133.3
5	540	152	4	$3.922 \cdot 10^{-8}$	12	91.2	131.5
4	526	152	4	$5.052 \cdot 10^{-7}$	10	74.7	133.9
3	478	152	4	$2.123 \cdot 10^{-5}$	8	59.7	134.0
2	532	188	4	$4.159 \cdot 10^{-4}$	6	45.4	132.2

Zwiększenie rozmiaru części ułamkowej poskutkowało zwiększeniem zapotrzebowania na DSP z 4 na 8, ale też poskutkowało osiągnięciem dokładności $6.733 \cdot 10^{-11}$ dla 1 mln próbek z przedziału $< -10, 10 >$, dla 8. stopnia w 18 cyklach zegara. Podobnie jak w przypadku wielomianów Czebyszewa, większa dziedzina jest związana z opublikowaniem przez autora częściowych wyników w publikacji [96]. Zmniejszanie stopnia wielomianu skutkuje redukcją niezbędnej liczby cykli do wykonania obliczeń o 2 cykle na stopień. W przypadku implementacji stałoprzecinkowej wynik z każdej operacji arytmetycznej dostępny jest w następnym cyklu zegara. Zmniejszenie wykorzystywanego wielomianu o 1 stopień zauważalnie redukuje obliczenia o jedną operację dodawania i jedną mnożenia.

Tabela 3.9. Wyniki implementacji z zastosowaniem wielomianów dla arytmetyki stałoprzecinkowej i różnej liczby przedziałów (Artix-7 XC7A100T-3CSG384)

Liczba przedziałów	Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
17	3	527	152	4	$9.347 \cdot 10^{-6}$	8	58.8	102.3
17	2	559	188	4	$2.439 \cdot 10^{-4}$	6	46.0	101.4
19	2	595	189	4	$8.631 \cdot 10^{-5}$	6	44.7	133.3
59	6	676	167	8	$7.500 \cdot 10^{-11}$	14	131.1	131.5
59	5	610	151	8	$3.515 \cdot 10^{-10}$	12	115.3	133.9
59	4	531	145	4	$4.122 \cdot 10^{-9}$	10	74.3	134.0
59	3	514	145	4	$3.723 \cdot 10^{-8}$	8	59.8	132.2
59	2	568	181	4	$1.080 \cdot 10^{-5}$	6	44.8	132.2
186	6	1055	168	8	$7.034 \cdot 10^{-11}$	14	137.4	132.2
186	5	958	151	8	$3.032 \cdot 10^{-10}$	12	118.9	132.2
186	4	940	151	8	$1.351 \cdot 10^{-9}$	10	97.5	132.2
186	3	811	145	4	$6.579 \cdot 10^{-9}$	8	60.4	132.2
186	2	837	181	4	$6.937 \cdot 10^{-8}$	6	44.1	132.2
244	6	1137	167	8	$1.376 \cdot 10^{-10}$	14	135.6	132.2
244	5	1099	165	8	$1.376 \cdot 10^{-10}$	12	119.2	132.2
244	4	992	151	8	$6.178 \cdot 10^{-10}$	10	97.3	132.2
244	3	869	145	4	$3.237 \cdot 10^{-9}$	8	60.2	132.2
244	2	905	181	4	$2.998 \cdot 10^{-8}$	6	45.5	132.2

Podobnie dla stopni 7. i 8. zauważalnie pomniejsza się maksymalna częstotliwość zegara. W tabeli 3.9 zestawiono wyniki implementacji dla różnej liczby przedziałów i stopni wielomianu oraz dla arytmetyki stałoprzecinkowej. Na podstawie danych można zaobserwować, że zwiększanie liczby przedziałów znacząco polepsza dokładność przy stosunkowo niewielkim dodatkowym zużyciu zasobów sprzętowych i bez wpływu na liczbę cykli potrzebnych na wykonanie obliczeń. Nie pogarsza też maksymalnej częstotliwości zegara. Z punktu widzenia czasu obliczeń korzystniej jest poprawić dokładność poprzez zwiększanie liczby przedziałów, i tak dla 2. stopnia wielomianu, ale przy 186 przedziałach, osiąga się podobną dokładność jak dla 5. stopnia przy 15 przedziałach, a przy tym liczba potrzebnych cykli dla 2. stopnia jest dwukrotnie mniejsza. W obu przypadkach podobna jest również maksymalna częstotliwość zegara.

Tabela 3.10. Wyniki implementacji z zastosowaniem wielomianów dla arytmetyki zmiennoprzecinkowej (Artix-7 XC7A100T-3CSG384)

Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
7	821	341	4	$5.895 \cdot 10^{-8}$	46	352.9	130.3
6	781	341	4	$5.895 \cdot 10^{-8}$	40	329.9	121.2
5	726	339	4	$6.656 \cdot 10^{-8}$	34	270.9	125.5
4	672	339	4	$5.216 \cdot 10^{-7}$	28	230.2	121.6
3	650	339	4	$2.121 \cdot 10^{-5}$	22	170.4	129.0
2	564	338	4	$3.283 \cdot 10^{-3}$	16	125.4	127.5

Analogicznie jak w przypadku wielomianów Czebyszewa, także tutaj zmniejszenie liczby cykli potrzebnych na wykonanie operacji arytmetycznej poniżej 2 skutkowało znaczącym spadkiem maksymalnej częstotliwości taktowania. Może negatywnie wpłynąć na konfigurację, w której funkcja tangensa hiperbolicznego byłaby zaimplementowana według opisywanego podejścia. Zwiększenie liczby cykli na operację nie skutkowało znaczącym polepszeniem parametrów, zaś znacząco zwiększyło ogólną liczbę cykli potrzebnych na wykonanie obliczeń. Motywuje to wykorzystanie ustawienia bloków arytmetycznych na 2 cykle na operację arytmetyczną, jako podejście wykorzystywane w dalszych eksperymentach opisywanych w tekście.

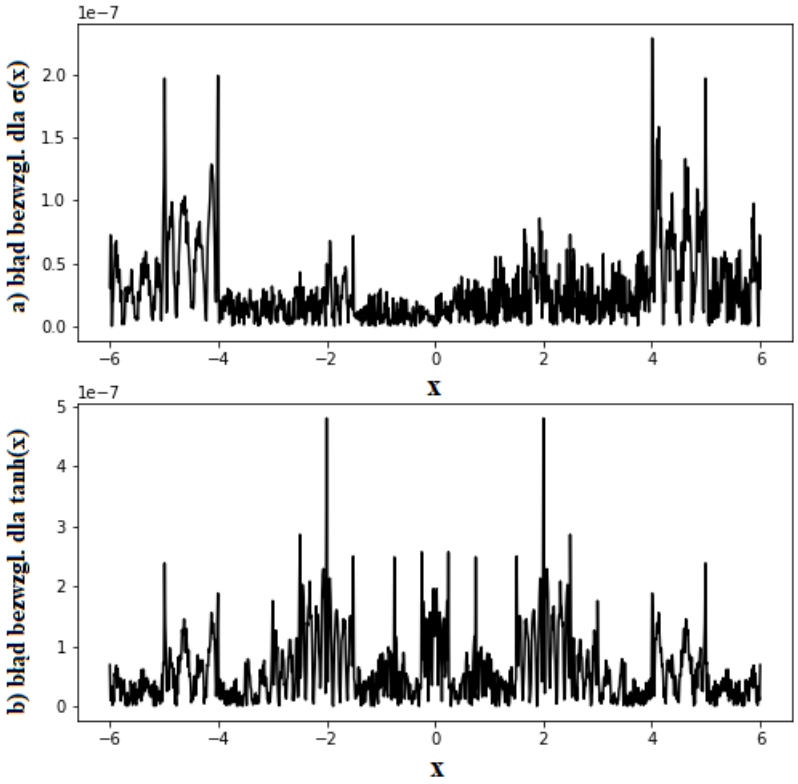
Tabela 3.11. Wyniki implementacji z zastosowaniem wielomianów dla arytmetyki zmiennoprzecinkowej i różnych nastaw bloków arytmetycznych (Artix-7 XC7A100T-3CSG384)

Liczba cykli na operację	LUT	FF	DSP	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
0	874	203	2	13	227.1	57.2
0	689	203	4	13	247.2	52.6
1	857	290	2	23	360.5	63.8
1	677	270	4	23	364.3	63.1
2	912	327	2	34	275.3	123.5
2	726	339	4	34	270.9	125.5
3	902	425	2	45	266.1	169.1
3	742	363	4	45	291.1	154.6
4	951	481	2	56	273.2	204.9
4	953	429	4	56	268.8	208.3

Tabela 3.12. Wyniki implementacji z zastosowaniem wielomianów dla arytmetyki zmiennoprzecinkowej i różnej liczby przedziałów (Artix-7 XC7A100T-3CSG384)

Liczba przedziałów	Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Czas obliczeń [ns]	Maksymalna częstotliwość zegara [MHz]
17	3	616	340	4	$9.353 \cdot 10^{-6}$	22	129.8	169.4
17	2	553	339	4	$2.439 \cdot 10^{-4}$	22	124.6	128.4
19	2	572	339	4	$8.630 \cdot 10^{-5}$	16	121.9	131.2
59	3	744	341	4	$7.630 \cdot 10^{-8}$	22	129.4	169.9
186	2	923	324	4	$1.213 \cdot 10^{-7}$	16	124.7	128.3
214	2	951	341	4	$9.968 \cdot 10^{-8}$	16	129.8	123.2
244	2	953	342	4	$8.498 \cdot 10^{-8}$	16	126.8	126.2

W tabeli 3.12 zestawiono wyniki implementacji dla arytmetyki zmiennoprzecinkowej, z uwzględnieniem różnej liczby przedziałów i różnych stopni wielomianu. Stosując 2. stopień wielomianu i 244 przedziałów osiąga się podobną dokładność jak przy zastosowaniu 5. stopnia i 15 przedziałów, przy wzroście zużycia zasobów, lecz przy zauważalnej redukcji liczby cykli potrzebnych na wykonanie obliczeń. Niezależnie



Rysunek 3.29: Wykres błędów dla aproksymacji funkcji sigmoidalnej oraz tangensa hiperbolicznego

od liczby przedziałów maksymalna częstotliwość zegara jest podobna w zakresie tego samego stopnia, zarówno dla 2. stopnia jak i dla 3. stopnia wielomianu.

Wykres błędu dla obu funkcji aktywacji został przedstawiony na rys. 3.29. Wykres został przygotowany dla 1000 punktów z przedziału $< -6, 6 >$. Można zaobserwować, że największe wartości błędów są osiągane dla małej liczby argumentów, co zostało zobrazowane w postaci wysokich pionowych linii. Dla większości argumentów wartość błędu jest zdecydowanie mniejsza i dla obu funkcji mniejsza od $1.5 \cdot 10^{-7}$. Bazując na otrzymanych wynikach oraz zgodnie z założeniem porównania dotyczącym osiąganej dokładności, do dalszych eksperymentów wykorzystana została implementacja funkcji tangens hiperboliczny z użyciem 4. stopnia wielomianu i 15 przedziałami, która osiąga dokładność $5.216 \cdot 10^{-7}$ zarówno dla zakresu $< -10, 10 >$ jak i $< -6, 6 >$, potrzebuje 28 cykli na obliczenia i zużywa 672 tablic LUT, 339 FF oraz 4 DSP oraz implementacja z użyciem 2. stopnia wielomianu i 186 przedziałami, która osiąga dokład-

ność $1.213 \cdot 10^{-7}$ dla przedziału $< -10, 10 >$ oraz $1.192 \cdot 10^{-7}$ dla przedziału $< -6, 6 >$, potrzebuje 16 cykli na obliczenia oraz zużywa 923 tablice LUT, 324 FF i 4 DSP. Analogiczna implementacja funkcji sigmoidalnej dla 4. stopnia i 15 przedziałów osiąga błąd $2.936 \cdot 10^{-7}$ dla przedziału $< -6, 6 >$ i zużywa 691 tablic LUT, 246 FF i 4 DSP. Dla 2. stopnia i 186 przedziałów osiąga błąd $2.896 \cdot 10^{-7}$ i zużywa 961 LUT, 320 FF oraz 4 DSP.

3.3.5. Analiza wyników

W tabeli 3.13 zestawiono wszystkie implementacje, które zostały wybrane do wykorzystania w dalszych eksperymentach. Przedstawiony w tabeli maksymalny błąd dla każdej implementacji został obliczony dla zakresu $< -6, 6 >$. Implementacje zostały podzielone na cztery warianty. Jako pierwszy wariant oznaczone zostało podejścia z zastosowaniem algorytmu wzorowanego na CORDIC w wersji poprawionej. Drugi wariant to podejście z zastosowaniem wielomianów Czebyszewa 4. stopnia i 15. przedziałów. Trzeci wariant dotyczy podejścia z zastosowaniem zwykłych wielomianów 4. stopnia i algorytmu Hornera, przy wykorzystaniu 15 przedziałów. Ostatni, czwarty wariant, to zastosowanie zwykłych wielomianów 2. stopnia i 186 przedziałów. Warianty 2, 3 i 4 miały bloki arytmetyczne ustawione na 2 cykle zegara, wariant pierwszy natomiast na 0.

Spośród wszystkich wariantów najbardziej oszczędnym w sensie ilości zasobów był wariant 3, który zużywał tylko 672 tablice LUT, 339 FF oraz 4 DSP w przypadku funkcji tangensa hiperbolicznego oraz 691 tablic LUT, 246 FF oraz 4 DSP w przypadku funkcji sigmoidalnej. Najszybszym wariantem okazał się wariant 4, w którym obliczenia wykonywane były w odpowiednio 16 i 19 cykli, przy wysokiej częstotliwości zegara. Najwięcej zasobów zużył wariant 1, który okazał się również najwolniejszy, natomiast prosty do wykonania. Tablicę danych do wyznaczania eksponenty można użyć w obrębie układu do innego rodzaju obliczeń, co sprawia, że wariant ten wciąż jest warty uwagi. Cechuje go też ważna właściwość, istotna z punktu widzenia analizy wpływu dokładności wykorzystanej aproksymacji funkcji aktywacji na pracę sieci, tj. łatwość w zmianie dokładności poprzez manipulację liczbą iteracji w algorytmie.

Tabela 3.13. Implementacje do dalszych rozważań (Artix-7 XC7A100T-3CSG384)

Liczba przedziałów	Funkcja / podejście	Stopień wielomianu	LUT	FF	DSP	Błąd maksymalny	Liczba taktów zegara	Maksymalna częstotliwość zegara[MHz]
Wariant 1								
–	$\tanh/COR$	–	1305	262	4	$2.473 \cdot 10^{-7}$	$67-80$	33.0
–	σ/COR	–	1298	253	4	$7.854 \cdot 10^{-7}$	$65-78$	34.6
Wariant 2								
15	$\tanh/Cz$	4	744	347	4	$2.980 \cdot 10^{-7}$	28	112.9
15	σ/Cz	4	730	323	4	$5.811 \cdot 10^{-7}$	31	115.0
Wariant 3								
15	$\tanh/Zw$	4	672	339	4	$5.216 \cdot 10^{-7}$	28	121.6
15	σ/Zw	4	691	246	4	$2.936 \cdot 10^{-7}$	31	126.0
Wariant 4								
186	$\tanh/Zw$	2	951	324	4	$1.192 \cdot 10^{-7}$	16	128.3
186	σ/Zw	2	961	320	4	$2.896 \cdot 10^{-7}$	19	128.1

Porównanie otrzymanych rezultatów z literaturą ma miejsce w tabeli 3.14, gdzie poniżej podwójnej linii wymienione zostały wyniki z niniejszej pracy. W niektórych przypadkach z prezentowanej literatury autorzy nie podali informacji dotyczących wykorzystywanego sprzętu, zużywanych zasobów, szczegółów implementacyjnych, czy sposobu implementacji, nie są one zatem zawarte w analizie. W prezentowanej literaturze spotkać można różne podejścia implementacyjne, które w niektórych przypadkach pozwalały osiągnąć wysoką dokładność odwzorowania funkcji aktywacji. Zastosowanie tablicy wartości, jak ma to miejsce w pierwszym wierszu tabeli, pozwala w prosty sposób odwzorować wiele różnych funkcji, natomiast osiągana dokładność nie jest wysoka. Dokładność można poprawić poprzez zastosowanie interpolacji, co jest pokazane w drugim wierszu tabeli. Brak informacji o zużyciu zasobów przez autorów odnotowano stosownie w tabeli.

W przypadku metody opisanej w trzecim wierszu tabeli 3.14 zastosowano aproksymację wielomianami 2. stopnia oraz arytmetykę zmiennoprzecinkową, osiągając przy tym stosunkowo dużą wartość błędu, tj. $7.4 \cdot 10^{-3}$. Aproksymacja liniowo-odcinkowa została zaprezentowana w pracy [83] oraz [17], osiągając przy tym wartość błędu powyżej 10^{-5}, podobną dokładność osiągnięto w [78]. W pracach nie przedstawiono szczegółów implementacyjnych i zapotrzebowania na zasoby sprzętowe.

Tabela 3.14. Dokładność aproksymacji tangensa hiperbolicznego w literaturze oraz w niniejszej pracy

Metoda	Błąd maksymalny	Komentarz
tablica wartości[86]	$1.2 \cdot 10^{-4}$	stały przecinek, brak danych dot. wykorzystanych zasobów i czasu obliczeń
interpolacja liniowa[86]	$1.6 \cdot 10^{-7}$	stały przecinek, brak danych dot. wykorzystanych zasobów i czasu obliczeń
aproksymacja wielomianem 2 stopnia [97]	$7.4 \cdot 10^{-3}$	zmienny przecinek, brak danych dot. wykorzystanych zasobów i czasu obliczeń
aproksymacja liniowo-odcinkowa [17] 256 przedziałów	$2.18 \cdot 10^{-5}$	8 cykli zegara, zmienny przecinek, brak danych dot. wykorzystanych zasobów
CORDIC do obliczeń tryg., z których liczono eksponentę a następnie tanh() [76]	$1.7 \cdot 10^{-7}$	64 bity, stały przecinek, 42 bity cz. dziesiętnej 1.57 ms (x>=0) / 2.73 ms (x<0), brak danych dot. wykorzystanych zasobów
aproksymacja liniowo-odcinkowa przedziałów [83]	$1.92 \cdot 10^{-4}$	zmienny przecinek, brak danych dot. wykorzystanych zasobów i czasu obliczeń
interpolacja DCT [78]	$1.00 \cdot 10^{-5}$	zmienny przecinek, 64 bity, brak danych dot. wykorzystanych zasobów i czasu obliczeń
interpolacja McLaurina[88]	$1.79 \cdot 10^{-7}$	87 cykli, 1916 LUT, 792 FF, 4 DSP, 89.2 MHz
z punktu 3.3.2	$2.473 \cdot 10^{-7}$	67 - 80 cykli, 1305 LUT, 262 FF, 4 DSP, 33.0 MHz, zmienny przecinek, 32 bity
z punktu 3.3.3, U2 8. stopień, w. Czeby.	$5.595 \cdot 10^{-11}$	20 cykli, 771 LUT, 331 FF, 8 DSP, 98.7 MHz kodowanie U2
z punktu 3.3.3, IEEE754 5. stopień, w. Czeby.	$1.192 \cdot 10^{-7}$	34 cykli, 757 LUT, 353 FF, 4 DSP, 116.9 MHz zmienny przecinek
z punktu 3.3.4, stały przecinek 8. stopień wielomianu	$6.733 \cdot 10^{-11}$	18 cykli, 690 LUT, 161 FF, 8 DSP, 102.3 MHz stały przecinek
z punktu 3.3.4, IEEE754 5. stopień wielomianu	$6.656 \cdot 10^{-8}$	34 cykli, 726 LUT, 339 FF, 4 DSP, 125.5 MHz zmienny przecinek

Wysoką dokładność osiągnięto w przypadku zastosowania algorytmu CORDIC i interpolacji McLaurina, tj. na poziomie 10^{-7}. W przypadku zastosowania algorytmu CORDIC wykorzystano arytmetykę stałoprzecinkową o długości słowa 64 bity i osiągnięto stosunkowo długi czas obliczeń tj. $1.57\,ms$. W porównaniu z wymienioną literaturą, zaprezentowane w pracy podejścia implementacyjne wypadają bardzo dobrze w przypadku osiąganej dokładności. Prezentowane podejścia są też szeroko opi-

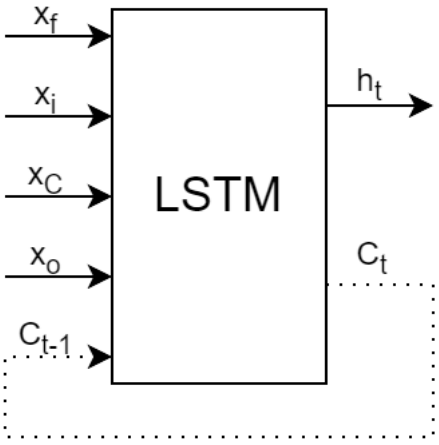
sane, uwzględniając czas obliczeń, maksymalną częstotliwość taktowania i wykorzystanie zasobów sprzętowych. Można również zauważyć, że prezentowane podejścia z zastosowaniem aproksymacji zwykłymi wielomianami lub wielomianami Czebyszewa pozwalają na uzyskanie wysokiej dokładności w stosunkowo prosty sposób i przy stosunkowo niskim zużyciu zasobów sprzętowych. Zdecydowanie lepsze wyniki można osiągnąć przy bezpośrednim aproksymowaniu obu funkcji aktywacji niż, na przykład, przy podejściu obliczającym najpierw wartość funkcji eksponenty i następnie wykorzystujących zależności matematyczne.

3.4. Implementacja komórki LSTM

W niniejszym punkcie przedstawiono budowę komórki LSTM, która została wykorzystana w dalszej części pracy. Zaprezentowano architekturę, strukturę logiczną oraz sposób organizacji obliczeń w module. Przedstawiono również sposób użycia opisywanego modułu w przypadku implementacji dowolnie dużej warstwy LSTM. Przedstawiana implementacja wykorzystuje funkcje aktywacji przedstawione w 3.3 i stanowi sprawdzenie wykorzystania przygotowanych modułów funkcji aktywacji w szerszym kontekście. Przedstawienie literatury w zakresie implementacji komórki LSTM wykonane jest w punkcie 3.5.1 razem z literaturą dotyczącą implementacji całych sieci LSTM. Porównanie z prezentowanymi tam podejściami ma miejsce w punkcie 3.5.7.

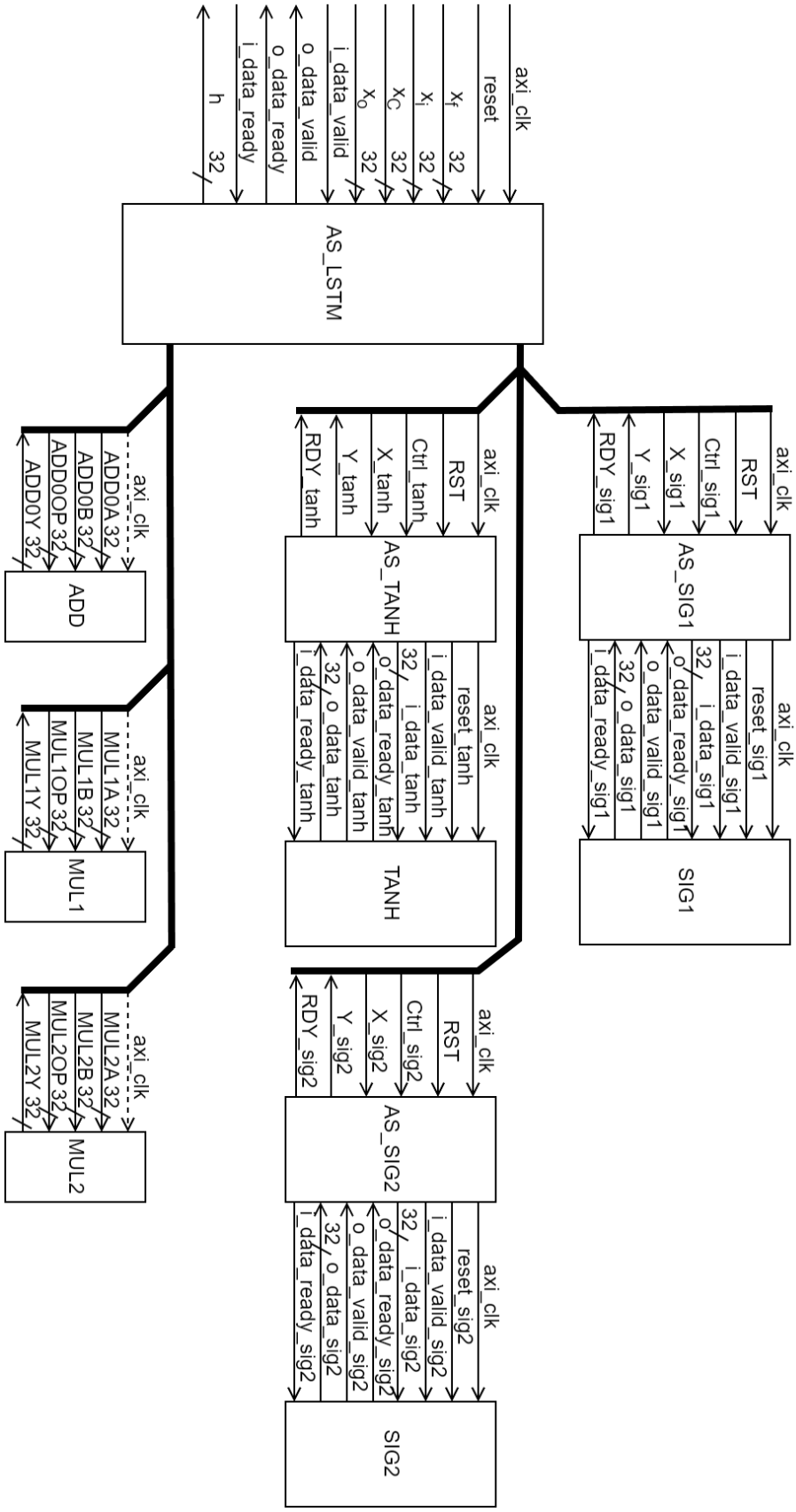
3.4.1. Opis implementacji

Opisywana architektura komórki LSTM wykonuje obliczenia zgodnie z równaniami (3.1)-(3.6), pomijając na obecnym etapie przemnażanie przez odpowiednie wagi i sumowanie sygnałów wejściowych oraz przesunięć (zostaną uwzględnione na dalszym etapie). Sumator sygnałów wejściowych potraktowany został jako osobny moduł, ponieważ jest jedynym elementem zmiennym w architekturze komórki (ogólna zasada implementacji przedstawiona jest w punkcie 3.4.2, a przykład implementacji w projekcie przedstawiony jest w punkcie 3.5.4). Niniejsza implementacja modułu komórki koncentruje się na przepływie sygnałów w obrębie komórki. Oznacza to, że argumenty funkcji aktywacji z równań (3.1) - (3.6) są przekazywane poprzez interfejs modułu komórki (rysunek 3.30), a proces akumulacji sygnałów i przemnażania ich przez wagi został wyodrębniony i wykonywany w oddzielnym module.



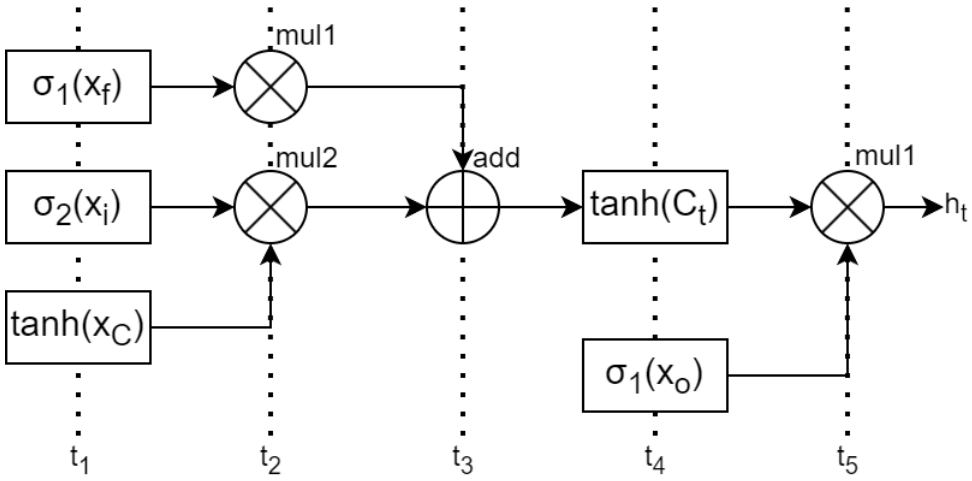
Rysunek 3.30: Przepływ sygnałów w obrębie modułu komórki LSTM bez sumatora

Przedstawiany moduł został podzielony na sześć bloków funkcyjnych (struktura logiczna została pokazana na rysunku 3.31): dwa bloki realizujące obliczenia wartości sigmoidalnej funkcji aktywacji, jeden blok realizujący obliczenia wartości funkcji tangensa hiperbolicznego, dwa bloki mnożące oraz jeden sumujący. Poza wymienionymi blokami w strukturze logicznej występował główny blok AS_LSTM, który był odpowiedzialny za obsługę komunikacji oraz zarządzanie obliczeniami, oraz bloki pośredniczące w obliczaniu funkcji aktywacji, które odpowiadały za koordynację obliczeń i komunikację z głównym blokiem. W ramach obliczeń, prezentowana komórka wykorzystywała dwukrotnie niektóre z bloków, według sekwencji załączeń przedstawionej na rys. 3.32, minimalizując dzięki temu zużycie zasobów sprzętowych. Przepływ danych w komórce uniemożliwia jednoczesne obliczenie wszystkich parametrów, przykładowo przy obliczaniu C_t korzysta się z wcześniej obliczonych f_t, i_t oraz $\hat{C}_t$, dlatego ponowne użycie bloków nie wprowadza dodatkowego opóźnienia. Ze względu na nieliniowość funkcji aktywacji oraz konieczność zachowania dużej dokładności obliczeń, największe zużycie zasobów oraz największe opóźnienie wprowadzane były przez bloki implementujące funkcję tangensa hiperbolicznego oraz funkcję sigmoidalną [91, 98, 20, 75, 9]. Implementacja komórki LSTM została przedstawiona w czterech wariantach tj. dla różnych funkcji aktywacji zaprezentowanych w punkcie 3.3.5. Warianty różniły się jedynie podejściem do implementacji funkcji aktywacji, pozostawiając pozostałe elementy komórki niezmienione. Każdy z wariantów operował na arytmetyce zmiennoprzecinkowej, zgodnej z IEEE754.



Rysunek 3.31: Struktura logiczna modułu komórki LSTM bez sumatora

Wejście do modułu komórki składało się z pięciu sygnałów tj. x_f, x_i, x_C, x_o oraz C_{t-1} (rys. 3.30), które były wykorzystywane w komórce, w sposób opisany równaniami (3.1) - (3.6), gdzie poszczególne sygnały wejściowe stanowiły wartość argumentu funkcji aktywacji oraz wartość stanu wewnętrznego z poprzedniego kroku obliczeniowego (C_{t-1}). W pierwszym kroku t_1 równolegle wykonane zostały dwie funkcje sigmoidalne odpowiednio dla sygnału f_t i i_t oraz funkcja tangensa hiperbolicznego dla sygnału C_t. Działanie funkcji jest synchronizowane, ponieważ jak widać w tabeli 3.13 długość obliczeń zależy od obliczanej funkcji. Po zakończeniu obliczeń z kroku t_1 wykonane zostały dwa iloczyny. Pierwszym jest iloczyn sygnału f_t oraz C_{t-1}, a drugim iloczyn sygnałów i_t oraz $\hat{C}_t$.



Rysunek 3.32: Sekwencja wykorzystania bloków w komórce

W trzecim kroku sumowane zostały poprzednio obliczone sygnały. Kroki t_2 i t_3 wykorzystały bloki arytmetyczne w funkcjach aktywacji, przez co minimalizowane jest zużycie zasobów sprzętowych. W kroku t_4 jednocześnie wykonywane były obliczenia dla ostatniego z wejść tj. funkcja sigmoidalna dla sygnału o_t oraz funkcja tangensa hiperbolicznego mająca jako argument wynik obliczeń z kroku t_3. Ostatnim krokiem było przemnożenie wyników z kroku t_4. Wartością wyjściową z komórki był sygnał h_t, który przekazywany został dalej w obrębie warstwy oraz po zakończeniu obliczeń do warstwy następnej. Równie istotnym sygnałem był sygnał C_t, również jest wykorzystywany w kolejnym kroku obliczeniowym, lecz tylko w obrębie komórki, której dotyczy i który w ramach opisywanej implementacji jest zapisywany pomiędzy cyklami wewnątrz komórki.

W celu zaznaczenia przepływu tego sygnału na rysunku 3.30 zostały uwzględnione porty C_{t-1} i C_t, a sam przepływ sygnału został naniesiony linią przerywaną, co oznacza że przepływ ten ma miejsce w obrębie komórki. W celu określenia dokładności obliczeń przygotowano, jako referencję, analogiczną implementację komórki LSTM na komputerze klasy PC, wykorzystującą pakiet NumPy. Wygenerowane dane wejściowe zostały oddzielnie przygotowane dla czterech sygnałów podawanym na wejścia bramek oraz dla sygnału stanu z poprzedniego cyklu. Dane testowe zawierały się w przedziale $< -6,6 >$. Ze względu na mnogość zmiennych i tym samym liczbę kombinacji, ograniczono w tym przypadku rozdzielczość do 100 punktów, co daje $100^5 = 10^{10}$ punktów testowych dla każdego z wariantów. Wyniki eksperymentu przedstawiono w tabeli 3.15.

Tabela 3.15. Implementacje komórki LSTM bez sumatora (Artix-7 XC7A100T-3CSG384)

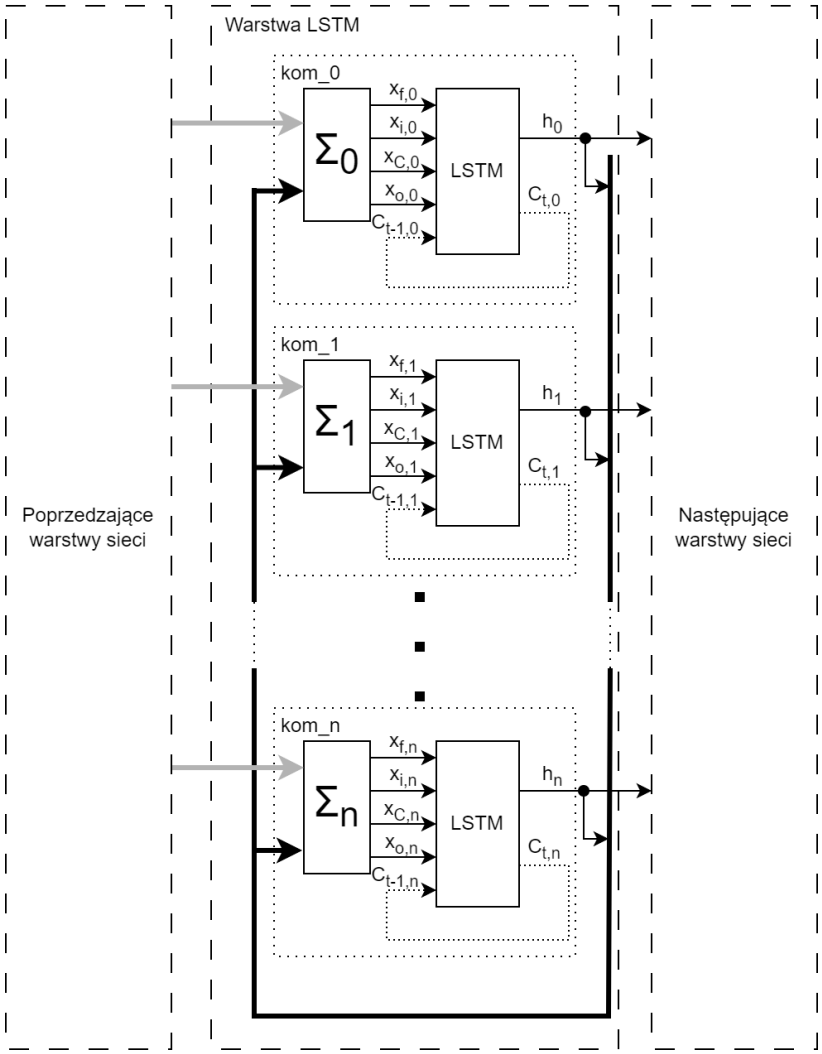
Wariant	LUT	FF	DSP	Błąd maksymalny h_t	Błąd maksymalny C_t	Liczba taktów zegara	Maksymalna częstotliwość zegara[MHz]
$Wariant\,1$	4292	974	12	$5.282 \cdot 10^{-6}$	$5.245 \cdot 10^{-6}$	$152 - 178$	37.1
$Wariant\,2$	2835	1025	12	$4.258 \cdot 10^{-6}$	$4.061 \cdot 10^{-6}$	80	124.0
$Wariant\,3$	2320	928	12	$7.489 \cdot 10^{-6}$	$7.398 \cdot 10^{-6}$	80	137.2
$Wariant\,4$	4553	960	12	$7.372 \cdot 10^{-6}$	$7.398 \cdot 10^{-6}$	56	121.3

Najmniejszy błąd uzyskano dla wariantu 2, czyli przy zastosowaniu wielomianów Czebyszewa. Błąd jest najmniejszy zarówno dla sygnału h_t, jak i C_t. Jak można było się spodziewać, najszybszym wariantem był wariant 4, wykonujący obliczenia w 56 cykli i mogący pracować do częstotliwości 121.3 MHz. Najmniej zasobów zużywał wariant 3: 2320 tablic LUT oraz 928 FF, wariant ten może pracować przy najwyższej częstotliwości maksymalnej zegara, natomiast spośród rozważanych wariantów osiąga względnie największy błąd. Nie stanowi dużej wady, gdyż wszystkie wartości błędów maksymalnych są bardzo zbliżone. Najwolniejszym i zużywającym relatywnie dużo zasobów okazał się wariant 1. Liczba cykli potrzebnych na wykonanie obliczeń zawierała się w przedziale $< 152,178 >$. Wariant ten może pracować też przy najmniejszej maksymalnej częstotliwości zegara wynoszącej 37.1 MHz.

3.4.2. Wykorzystanie modułu komórki LSTM

Opisany w punkcie 3.4.1 moduł komórki LSTM realizuje dużą część obliczeń z równań (3.1) - (3.6), pomijając jedynie obliczenie argumentów funkcji aktywacji. W ni-

niejszej pracy przyjęto, że to zadanie będzie realizowane poprzez oddzielny moduł określany jako sumator i oznaczany dalej jako Σ. Moduł ten różni się w zależności od implementacji sieci, w nim bowiem znajdują się wagi połączeń, a ich liczba zależy od rozmiaru warstwy ukrytej oraz liczby połączeń z poprzedniej warstwy. Sposób zliczania iloczynów sygnałów i wag zależny jest od liczby parametrów oraz dostępnych zasobów w układzie. Mając dostępną dużą liczbę zasobów można część obliczeń wykonać równolegle, przez co proces zliczania będzie zajmował mniej kroków, ale wykorzysta większą ilość zasobów sprzętowych. Decyzje dotyczące sposobu implementacji mogą się znacząco różnić. W punkcie 3.5.4 przedstawiony został przykład implementacji sumatora, który można dowolnie wykorzystać dostosowując do rozwiązywanego problemu i ograniczeń sprzętowych. Mając moduł sumatora oraz moduł komórki można przystąpić do budowy warstwy LSTM, której schemat ogólny przedstawiono na rysunku 3.33.



Rysunek 3.33: Wykorzystanie modułu komórki LSTM

W warstwie LSTM każdy z n elementów sieci został zbudowany z modułu sumatora oraz modułu komórki LSTM. Do sumatora wysyłane są sygnały z poprzedniej warstwy oraz sygnały rekurencyjne, a na jego wyjściu wystawiane są wartości argumentów dla wszystkich czterech funkcji aktywacji, które następnie są kierowane na moduł komórki LSTM. Na schemacie przedstawiono pętle sygnałową z wyjścia $C_{t,n}$ do wejścia $C_{t-1,n}$, która w rzeczywistości jest wykonywana wewnątrz modułu, na schemacie zaznaczona linią przerywaną. Wyjściem z modułu jest sygnał wyjściowy h_n, grupowany na schemacie wraz z sygnałami z pozostałych komórek (pogrubiona czarna linia) i przekazywany na wejście każdej komórek jako zbiór sygnałów rekurencyjnych. Sygnał wyjściowy z każdej komórki był również przekazywany do warstwy następującej po LSTM w sieci. Przedstawione na rysunku 3.33 podejście pozwala na implementację dowolnie dużej warstwy LSTM, ograniczonej tylko dostępnymi zasobami sprzętowymi, dostosowującej jedynie blok sumatora m.in. poprzez wgranie odpowiednich parametrów wag i przesunięć oraz ustawienie odpowiedniej liczby sygnałów w sieci.

3.4.3. Wnioski z implementacji komórki LSTM

W podrozdziale przedstawiono wykonanie implementacji komórek LSTM z różnymi funkcjami aktywacji, ze szczególnym uwzględnieniem implementacji funkcji aktywacji. Materiał zawarty w podrozdziale jest przede wszystkim sprawdzeniem wykorzystania modułów funkcji aktywacji w szerszym kontekście. Wzięto pod uwagę metodę obliczającą najpierw wartość eksponenty według algorytmu wzorowanego na CORDIC, a następnie obliczającą wartość funkcji wykorzystując zależności matematyczne pomiędzy eksponentą a funkcjami aktywacji. Uwzględniono również metodę aproksymującą z wykorzystaniem zwykłych wielomianów dla różnych stopni wielomianów i różnego podziału dziedziny na odcinki oraz metodę aproksymującą wykorzystującą wielomiany Czebyszewa dla różnego maksymalnego stopnia wielomianów oraz różnego podziału dziedziny na odcinki. Metody zgrupowano w cztery warianty przedstawione w tabeli 3.13. Metody bezpośrednio aproksymujące funkcje aktywacji okazały się zdecydowanie szybsze zarówno w zakresie liczby cykli jak i maksymalnej częstotliwości zegara. Szczególnie dobre wyniki, w sensie szybkości obliczeń, można osiągnąć poprzez wykorzystanie małego stopnia wielomianu, ale za to dużej liczby przedziałów dziedziny (wariant 4). Niesie to za sobą zwiększenie zużycia zasobów sprzętowych ze względu na większą liczbę współczynników, które muszą być przetrzymywane w pamięci układu. Zmiana rodzaju wielomianów nie wpływa znacząco na wyniki, Zarówno wykorzystanie

zasobów sprzętowych, jak i dokładność obliczeń jest na podobnym poziomie. Wariant 1 ze względu na iteracyjny charakter obliczeń umożliwia łatwe wykonanie sprawdzenia wpływu dokładności na pracę układu w którym zostanie umieszczony. Nie wymaga zmian w strukturze, lecz jedynie określenia liczby iteracji algorytmu. Biorąc pod uwagę pozostałe aspekty, ustępuje jednak pozostałym wariantom tak pod względem liczby cykli, jak i maksymalnej częstotliwość zegara. W tabeli 3.15 przedstawiono wyniki implementacji komórek LSTM bez uwzględnienia sumatora, w oparciu o wcześniej przyjęty podział na warianty. Wykonanie szeregu obliczeń w komórce zauważalnie wpłynęło na osiąganą dokładność, która dla obu parametrów wyjściowych h_t oraz C_t spadła o rząd wielkości dla badanego zakresu zmiennych wejściowych. Nieoczywistym rezultatem jest mniejsze zużycie zasobów dla wariantu 1 w stosunku do 4, natomiast analizując liczbę cykli można zauważyć, że wariant 4 jest wielokrotnie szybszy od wariantu 1. Najbardziej oszczędny w sensie zasobów jest wariant 3. Osiąga też najwyższą częstotliwość zegara. Należy tu mieć na uwadze, że przedstawiane wyniki zostały wygenerowane dla układu Artix-7 XC7A100T-3CSG384 z wykorzystaniem oprogramowania ISE Design Suite i mogą się różnić przy próbie implementacji na innych układach lub przy wykorzystaniu innego oprogramowania. Oprogramowanie, a więc wykorzystane w nim algorytmy i techniki optymalizacyjne, może wpływać na wykorzystanie i sposób rozmieszczania wykorzystywanych elementów, co ma bezpośrednie przełożenie m.in. na maksymalną częstotliwość zegara.

Przedstawienie podejść spotykanych w literaturze wykonane zostało razem z prezentacją literatury dotyczącej sieci LSTM w punkcie 3.5.1. Porównanie z prezentowanymi tam podejściami implementacyjnymi zostało przedstawione w punkcie 3.5.7 razem z porównaniem implementacji całych sieci LSTM.

3.5. Wykorzystanie implementacji sieci LSTM na FPGA do zadania klasyfikacji w procesie kucia na zimno

Obecnie można zaobserwować dążenia do zmiany sposobu wytwarzania, zgodnego z koncepcją Smart Manufacturing oraz koncepcją Industry 4.0, które zakładają wysoki poziom automatyzacji, przetwarzania wielu danych oraz szybkiego dopasowywania do nowych wymogów projektowych. Nowa filozofia zarządzania wiąże się z tzw. inteligentnymi maszynami, które pozwalają na monitorowanie wielu parametrów pracy

i w oparciu o nie, przewidywanie ilości wytwarzanych produktów, problemów z jakością, potrzebnych napraw i związanych z nimi przestojów [6, 7]. Szczególnie planowanie przestojów na przezbrajanie lub naprawę jest kluczowe i pozwala znacząco zredukować koszty. Powstaje wiele analiz dotyczących zastosowania uczenia maszynowego. Przykładem jest [6], w którym przedstawiono studium użycia uczenia maszynowego w przemyśle, biorąc pod uwagę zadania związane z przetwarzaniem języka naturalnego. Pomimo szybkiego rozwoju sieci komputerowych w przemyśle, wciąż można dostrzec zastosowania wymagające np. instalacji klasyfikatora wykrywającego wady produktu zaraz przy maszynie, minimalizując przy tym czas reakcji i uniezależniając się od obciążenia sieci i działania serwerów. Szczególnie przy szybkich procesach typu kucie na zimno, np. główek śrub ma to znaczenie. Klasyfikator zbudowany w oparciu o FPGA wydaje się być tu adekwatnym rozwiązaniem.

3.5.1. Implementacja głębokich sieci neuronowych na układach FPGA w literaturze badawczej

Zastosowanie FPGA do realizacji zadań przemysłowych jest chętnie podejmowanym tematem przez badaczy. W [13] skupiono się na zastosowaniu FPGA do realizacji obliczeń związanych z sieciami konwolucyjnymi. Wykorzystano w tym celu płytkę ewaluacyjną Xilinx Zedboard wyposażoną w układ typu ZYNQ, który jest połączeniem procesora ARM z FPGA w jednej obudowie. Prezentowana sieć jest bardzo mała i przyuczona do zbioru MNIST. Pomimo prostej budowy i wykorzystaniu prostych funkcji aktywacji, sieć zużywa 85.5% DSP, 47.8% tablic LUT i 45.8% BRAM. Osiągnięto przy tym szybkość obliczeń porównywalną z referencyjnym programem korzystającym z GPU.

W [12] również przeanalizowano wykorzystanie FPGA w zadaniu analizy obrazu ze zbioru MNIST. Autorzy wykonali implementacje sieci CNN na platformie ZYNQ oraz Artix 7, nie podając przy tym szczegółów implementacyjnych poza wymienieniem funkcji aktywacji, którą okazała się ReLU.

Praca [99] jest kolejnym podejściem do wykorzystania FPGA, konkretnie Xilinx Virtex UltraScale+ XCVU9P, do zadania analizy obrazu z wykorzystaniem sieci CNN. W tym przypadku praca opierała się na analizie dużych zdjęć z kompresją JPEG, co wpływało na wykorzystanie zasobów: 274 tys. tablic LUT, 273 tys. FF oraz 2.4 tys. DSP. Autorzy natomiast osiągnęli bardzo wysoką przepustowość.

Opublikowanych prac związanych z wykorzystaniem układów FPGA do imple-

mentacji sieci LSTM jest zdecydowanie mniej. Jedną z takich prac jest [100], gdzie autorzy zwracają uwagę na znaczący atut FPGA, jakim jest energooszczędność. Przedstawiona tam implementacja wykonana została z użyciem płyty ewaluacyjnej Zedboard. Porównana została następnie z procesorami Core i5 oraz Cortex-A9. Przy wykonaniu takiego samego zadania, energia wydzielona przez FPGA okazała się zdecydowania niższa. Implementacja na FPGA okazała się również szybsza. W pracy rozważano sieci o różnym rozmiarze warstwy LSTM tj. o 32, 64, 128 i 256 komórkach, gdzie obliczenia były przeprowadzane z użyciem pipeliningu, a w strukturze sprzętowej zaimplementowana była jedna komórka. Parametry sieci były wcześniej obliczone przy wykorzystaniu biblioteki Tensorflow. Użycie pipeliningu dotyczyło głównie modułu wykonującego obliczenia przemnażania macierzy wag z wektorami sygnałów wejściowych i rekurencyjnych, tj. pełniącego rolę sumatora. Następnie, obliczone wartości wykorzystywane były do wyliczenia wartości funkcji aktywacji. Wszystkie cztery funkcje wykonywane były równolegle i opierały się o aproksymację liniowo-odcinkową. Wartości sygnałów wykorzystywane były do obliczeń stanu wewnętrznego oraz sygnału wyjściowego. Artykuł nie zawiera informacji o miejscu i sposobie wykonania obliczeń związanych z wartością wyjściową komórki, na którą składa się ponowne wywołanie funkcji tangensa hiperbolicznego.

Praca [101] prezentuje akcelerator dla sieci LSTM zbudowany ze szczególną uwagą poświęconą zoptymalizowaniu działania oraz komunikacji. Autorzy zwracają uwagę, że obliczenia związane z dużymi rekurencyjnymi sieciami neuronowymi nie mogą być wydajnie wykonywane na CPU lub GPU i proponowane jest stosowanie FPGA w tym zakresie, jako rozwiązania bardziej dopasowanego do problemów i pobierającego mało energii elektrycznej. Publikacja traktuje o zadaniach związanych z rozpoznawaniem mowy. Zaprezentowana implementacja korzysta z liniowej aproksymacji funkcji aktywacji, dla której osiąga 0.63% błędu oraz z reprezentacji zmiennoprzecinkowej liczb. Organizacja obliczeń związanych z komórką LSTM polega na podziale obliczeń na 5 kroków: obsługa bufora wejściowego; przemnażanie przez wagi i sumowanie sygnałów wejściowych oraz rekurencyjnych; wykonanie obliczeń funkcji aktywacji dla czterech bramek komórki LSTM; wykonanie obliczeń wewnętrznych; obsługa bufora wyjściowego. Nie podano informacji o sposobie obliczeń sygnału wyjściowego, który według równań wymaga obliczenia funkcji tangensa hiperbolicznego. W artykule nie zaprezentowano analizy jakości klasyfikacji ani architektury sieci, natomiast podano

informację na temat zużytych zasobów: 198 tys. tablic LUT, 182 tys. FF oraz 1176 DSP dla układu Xilinx Virtex7-485.

W [102] została zaprezentowana implementacja sieci na FPGA, składającej się z dwóch warstw i 128 komórek LSTM. Sieć na FPGA okazała się 21 razy szybsza w porównaniu z analogicznym modelem na procesorze ARM Cortex-A9. Autorzy zasugerowali również, że podobne implementacje mogą być wykorzystywane jako koprocesory w telefonach komórkowych. W artykule wykorzystano arytmetykę Q8.8, komunikacja z układem FPGA odbywała się poprzez port DMA. Zaimplementowany model został wcześniej przygotowany na komputerze klasy PC, z użyciem Torch7. Komórka LSTM wykorzystana w artykule składa się z modułów mnożąco-akumulujących, modułów funkcji aktywacji, kolejki FIFO raz modułu nazwanego Ewise. Moduł mnożąco-akumulujacy przyjmuje na wejściu dwa wektory: z wartościami wejściowymi oraz z wartościami wag. Wartości są przemnażane element po elemencie i sumowane, a następnie przekazywane do modułów funkcji aktywacji. W komórce zaimplementowano dwa moduły obliczające wartość funkcji sigmoidalnej i jeden obliczający wartość funkcji tangensa hiperbolicznego. Wyniki obliczeń podawane były do kolejki FIFO, a następnie do Ewise, podłączonego do modułów mnożąco-akumulujacych, pełniących również funkcję routera. Dzięki temu możliwe było wykonywanie obliczeń wewnątrz komórki, którymi nadzorował moduł Ewise. Wynikiem tych obliczeń była wartość sygnału wewnętrznego komórki oraz sygnał wyjściowy. Funkcje aktywacji aproksymowano metodą liniowo-odcinkową. Średni błąd dla C_t wyniósł 3.9%, zaś dla h_t 2.8%.

Implementacja sieci LSTM, którą warto przywołać, znajduje się również we wspominanej już pracy [75], w której autorzy próbowali osiągnąć lepsze rezultaty obliczeń na FPGA niż na komputerze PC. W efekcie implementacja na FPGA okazała się 251 razy szybsza w porównaniu z analogicznym rozwiązaniem uruchamianym na procesorze i7-3770k, realizowanym w Pythonie. Autorzy wykorzystali aproksymację wielomianami w celu implementacji funkcji aktywacji oraz arytmetykę stałoprzecinkową Q6.11. Podczas aproksymacji osiągnęli maksymalny błąd $1.4 \cdot 10^{-3}$ dla funkcji sigmoidalnej oraz $1.21 \cdot 10^{-2}$ dla funkcji tangensa hiperbolicznego. W komórce LSTM zaimplementwano moduł obliczający iloczyn wag i sygnałów oraz sumujący wyniki razem z wartością przesunięciem. Efektem pracy modułu były wartości wejściowe sygnałów dla każdej z bramek, które dalej były połączone z dwoma multiplekserami, podłączonymi do modułów funkcji aktywacji. Odpowiednie wysterowanie multiplekse-

rów określało dla jakiej bramki wykonywane były obliczenia. Na wejście multipleksera podłączonego do modułu tangensa hiperbolicznego znajdował się również sygnał stanu wewnętrznego komórki, przez co możliwe było wykonanie wszystkich obliczeń z wnętrza komórki za pomocą dwóch modułów funkcji aktywacji. Wyjścia z modułów funkcji aktywacji podawane były dalej na układ mnożący i sumujący, co umożliwiło wykonanie obliczeń na sygnałach, a w efekcie obliczenie wspomnianego stanu wewnętrznego komórki oraz wartości wyjścia komórki. Autorzy przy pomocy sieci LSTM próbowali rozwiązać problem dodawania dwóch liczb 8 bitowych. Autorzy zrezygnowali z pełnej równoległości obliczeń i ograniczyli liczbę bloków realizujących funkcję aktywacji do dwóch - po jednym dla każdej z funkcji. Pomimo ograniczeń, zużycie tablic LUT doszło do 91% przy 32 komórkach dla płyty ewaluacyjnej Virtex-7 VC707. Nie podano wyników pracy sieci oraz czy zakładana dokładność jest wystarczająca do prawidłowego działania sieci.

W publikacji [103] zauważono, że układy FPGA mogą być wartościową alternatywą przy złożonych obliczeniach np. dotyczących sieci LSTM dla CPU lub GPU. Autorzy deklarują, że w przypadku wykorzystania układu, wydajność w stosunku do CPU i GPU wzrosła odpowiednio o 8.8 i 2.2 raza, a poprawa efektywności energetycznej o odpowiednio 16.9 i 9.6 raza. Do obliczeń skorzystano z FPGA, model Kintex UltraScale XCKU115-2FLVB2104E, z uwzględnieniem arytmetyki stałoprzecinkowej Q5.6. Funkcje aktywacji zostały zaimplementowane jako tablice z wartościami o wielkości 4096 elementów. Autorzy nie podali analizy dokładności uzyskanej w takim podejściu. Układ został zintegrowany z obliczeniami na komputerze, wykonywanymi w Caffe (platforma uczenia głębokiego opracowana na Uniwersytecie Kalifornijskim w Berkeley, napisana w C++ i posiadająca interfejs Pythona), gdzie przed wysłaniem danych na FPGA, dane były konwertowane do arytmetyki stałoprzecinkowej, a następnie po wykonaniu obliczeń z powrotem do arytmetyki zmiennoprzecinkowej. Autorzy nie precyzują, czy konwersje zostały uwzględnione w pomiarze czasu obliczeń na FPGA ani czy pomiary uwzględniały czas transmisji. Zaimplementowana sieć składała się z dwóch warstw: wbudowanej (bez opisu struktury w tekście publikacji) oraz LSTM, obie składały się z 512 elementów. Obliczenia związane z komórką LSTM podzielone były na 3 etapy, którym odpowiadał osobny moduł logiczny: obliczenia macierzowe (funkcjonalność sumatora), obliczanie funkcji aktywacji, obliczenia wewnętrzne komórki. Moduły połączone były poprzez dwa bufory. Pierwszy składał się z wartości wejściowych dla

bramek komórki, a drugi z sygnałów wewnętrznych komórki. W prezentowanym rozwiązaniu obliczenia dla wszystkich czterech bramek były wykonywane jednocześnie, a moduł odpowiedzialny za obliczenia funkcji tangensa hiperbolicznego znajdował się również w module, w którym wykonywane były obliczenia wewnętrzne komórki. Moduł wykonujący obliczenia wewnętrzne wykorzystywał dodatkowo 3 układy mnożące oraz jeden sumujący. W publikacji nie uwzględniono porównania efektów obliczeń z rozwiązaniami opartymi o CPU lub GPU oraz raportu z wykorzystania zasobów. Brak również danych dotyczących dokładności.

W artykule [104] przedstawiono akcelerator sieci LSTM oparty na FPGA. Celem autorów była optymalizacja prędkości i dokładności istniejących modeli. Implementacja została wykonana na układzie Xilinx's Virtex-7 VC707, z wykorzystaniem narzędzi do generowania struktury logicznej - w tym przypadku Vivado HLS. Ze względu na to, że nie jest zaimplementowana w dostarczanych bibliotekach, sigmoidalna funkcja aktywacji została silnie zlinearyzowana poprzez aproksymację odcinkami w trzech przedziałach. Obliczenia funkcji aktywacji w komórce LSTM odbywały się jednocześnie dla czterech bramek wejściowych. Moduł tangensa hiperbolicznego był ponownie wykorzystywany na dalszych etapach obliczeń. Zadanie związane z przemnażaniem sygnałów wejściowych i rekurencyjnych przez wagi oraz akumulacja iloczynów zostało zaimplementowane w bramkach komórki. Implementowana sieć składała się z 19 elementów w warstwie wejściowej, 10 komórek LSTM i 19 elementów w warstwie wyjściowej. Sieć została zamodelowana i przyuczona na komputerze, z wykorzystaniem Pythona oraz biblioteki Keras, spełniając również funkcję programu referencyjnego, względem którego porównywano czas obliczeń. Prezentowana implementacja zużyła 285558 LUT, 194318 FF, 1730 DSP i 6 BRAM. W pracy nie ma informacji o przyjętej arytmetyce oraz o sposobie projektowania struktury logicznej z wykorzystaniem narzędzi HLS. Praca nie prezentuje analizy dokładności pomimo zwrócenia uwagi przez autorów na istotę dokładnego odwzorowywania pracy sieci po zaimplementowaniu na FPGA. Osiągnięte przyspieszenie obliczeń w stosunku do referencyjnych wyniosło 28.76 razy.

Praca [105] miała na celu realizację modułu wykonującego operacje mnożenia macierzy przez wektory oraz maksymalizację wykorzystania równoległości w wykonywaniu operacji matematycznych. Do modułu akceleratora LSTM oddzielnie doprowadzano trzy bufory: z wartościami sygnałów wejściowych, z wartościami wag oraz

dwukierunkowy bufor z wartościami sygnałów wyjściowych. W module akceleratora autorzy umieścili moduły obliczeniowe na które składały się: bufor wartości wag, 4 układy mnożące, 4 układy dodające. Każda para układów arytmetycznych (jeden układ mnożący i jeden układ dodający) odpowiadała za obliczenia wartości sygnału wejściowego dla poszczególnych bramek komórki LSTM. Wykorzystano arytmetykę 16-bitowa, lecz nie podano, czy stałoprzecinkową czy też zmiennoprzecinkową. Funkcje aktywacji zostały zaimplementowane w postaci tablic wartości, a w ramach komórki zostały zaimplementowane 2 moduły: jeden obliczający wartość funkcji sigmoidalnej, a drugi tangensa hiperbolicznego. Obliczenia wewnątrz komórki odbywały się w 7 etapach: obliczenie sygnału dla bramki wejściowej oraz aktywacji wejścia; iloczyn sygnałów obliczonych w poprzednim etapie; obliczenie wartości sygnału dla bramki zapominającej; iloczyn sygnału z bramki zapominającej oraz wartości poprzedniego stanu wewnętrznego komórki; suma wartości obliczonych w kroku 2 i 4, czego wynikiem jest nowa wartość stanu wewnętrznego; obliczenie wartości sygnału dla bramki aktywującej wyjście oraz wartości funkcji tangensa hiperbolicznego z wartości stanu wewnętrznego; iloczyn wartości z poprzedniego etapu, czego wynikiem jest wartość wyjściowa komórki. W ramach pracy modelowana została sieć z warstwą LSTM, składająca się z 32 komórek, oraz z warstwą gęsto połączoną, o nieokreślonej w opisie strukturze. Sieć została zrealizowana na układzie Virtex-7 VC707 i wykorzystała 23473 LUT, 21389 FF, 142 DSP oraz 165 BRAM. Nie podano liczby komórek faktycznie zaimplementowanych w strukturze FPGA, nie przedstawiono analizy działania sieci oraz tego czy osiągana dokładność jest wystarczająca do poprawnego działania systemu. Prezentowana implementacja okazała się 43 razy szybsza od CPU (Intel Core i5-6500) oraz 3 razy szybsza niż GPU (GeForce GTX 1660). Programy referencyjne zostały przygotowane w Pythonie z użyciem biblioteki Keras.

W literaturze przedmiotu znaleźć można prace, dotyczące tematyki implementacji sieci LSTM na układach FPGA, których celem jest prezentacja metody optymalizacyjnej, a nie zastosowanie w konkretnym problemie. Nie porównują również w swoim zakresie różnych platform uruchomieniowych. Przykładem jest praca [106] dotycząca kompresji. Artykuł prezentuje metodę kompresji SIBBS (Shared Index Bank-Balanced Sparsity). Wiersze macierzy wag dla warstwy LSTM podzielono na wiele klastrów, aby zbalansować niezerowy rozkład wag. Elementy klastra dzieliły ze sobą indeksy. Zastosowanie metody pozwoliło na skrócenie długości obliczeń o 1.47-79.5 razy. Dodatkowo

zastosowana metoda pomijania obliczeń w oparciu o podobieństwa, pozwalała zaoszczędzić 10% operacji dla prezentowanej implementacji. Całość optymalizacji wpływała jednak na dokładność, której spadek autorzy odnotowali jako dochodzący do 4%. W implementacji wykorzystano arytmetykę stałoprzecinkową 8-bitową dla wag i 12-bitową dla funkcji aktywacji. W ramach prac zaimplementowano trzy różne sieci, z których najmniejsza oznaczona została jako Google LSTM i wykorzystywała 327683 LUT, 364447 FF, 546 BRAM i 4224 DSP, na układzie Xilinx XCKU115. Dokładnej architektury sieci nie przedstawiono. Implementacja wykorzystywała liniowo-odcinkową metodę aproksymacji, a sama sieć została najpierw wytrenowana na PC z użyciem biblioteki PyTorch.

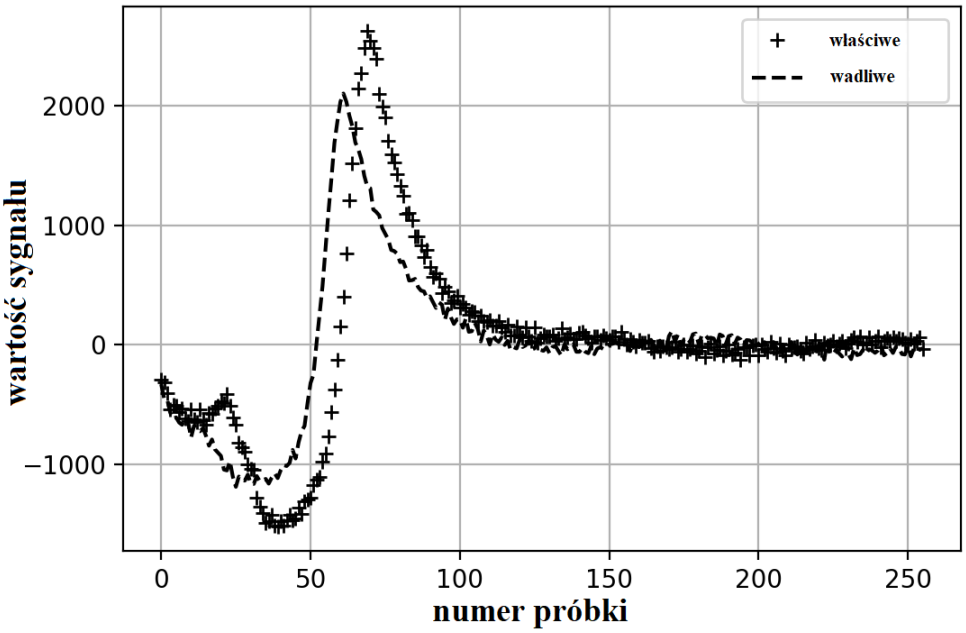
Prezentowane w ramach przeglądu literatury implementacje komórki LSTM głównie wykonywały obliczenia dla każdej z bramek jednocześnie lub przy wykorzystaniu dwóch modułów (jednego modułu na rodzaj funkcji aktywacji). Pierwsze rozwiązanie sprawia, że wymagane jest zaimplementowanie 4 modułów, spośród których tylko jeden może zostać użyty ponownie (funkcja tangensa hiperbolicznego przy obliczaniu wartości wyjściowej). Drugie z kolei wymaga sekwencjonowania obliczeń dla każdej z bramek osobno, co znacząco wydłuża obliczenia, chociaż pozwala na zaoszczędzenie wykorzystania zasobów na FPGA. Zaprezentowana w ramach niniejszej pracy komórka LSTM (punkt 3.4) implementuje natomiast 2 moduły funkcji sigmoidalnej i jeden tangensa hiperbolicznego. Wykorzystane ponownie zatem zostają 2 moduły, które wykonują obliczenia równolegle (rysunek 3.32), przez co zaoszczędzenie wykorzystanych zasobów nie odbywa się kosztem wydłużenia czasu obliczeń. Przy założeniu takich samych modułów funkcji aktywacji, można zauważyć, że niniejsze podejście jest oszczędniejsze niż w przypadku spotykanego w literaturze rozwiązania pierwszego i tak samo szybkie. W porównaniu z drugim, prezentowane w ramach niniejszej pracy rozwiązanie jest szybsze, lecz zużywa więcej zasobów sprzętowych.

Pomimo istnienia literatury dotyczącej implementacji sieci LSTM na FPGA, nie zawiera ona dokładnego opracowania, porównującego ze sobą implementacje na FPGA z innymi platformami obliczeniowych, w publikacjach często nie podaje się szczegółów implementacyjnych i nie weryfikuje się, jak podejmowane w trakcie implementacji decyzje wpływają na działanie sieci. Sprawia to, że niemożliwe jest jednoznaczne określenie w jakich okolicznościach implementacja na FPGA może być korzystnym wyborem i jakie są takiego wyboru konsekwencje.

3.5.2. Proces kucia na zimno

W niniejszym punkcie oraz w dalszej części pracy, w celu wytrenowania sieci oraz sprawdzenia jej działania, wykorzystane zostaną dane udostępnione przez autorów [2], za co autor wyraża głęboką wdzięczność.

Proces kucia na zimno jest często stosowanym, wydajnym i szybkim procesem kształtowania metalu. W trakcie procesu kształt obiektu jest zmieniany w temperaturze otoczenia. Ze względu na dużą dokładność, zakres możliwych kształtów jest szeroki, a wytwarzane elementy cechują się lepszą stabilnością wymiarów, lepiej wykończoną powierzchnią oraz niższymi kosztami produkcji niż w przypadku np. kucia na gorąco, zaś w porównaniu z obróbką mechaniczną zapewnia mniejszy odpad produkcyjny. W pracy [2] autorzy przeanalizowali popularny, jedno-matrycowy wariant kucia na zimno z dwoma uderzeniami, zastosowany do produkcji główek śrub. W analizowanym procesie, drut pobierany jest z podajnika i cięty na określoną długość, a następnie podawany dalej w celu nadania kształtu główki śruby w dwóch następujących po sobie uderzeniach. Pierwsze uderzenie nadaje zgrubny kształt, zaś drugie ostateczny. Po drugim uderzeniu element jest wyrzucany z maszyny i proces rozpoczyna się od początku. Maszyna była wyposażona w czujnik piezoelektryczny, mierzący siły oddziałujące na obrabiany element. Sygnał odczytywany i zapisywany był poprzez sterownik PAC z analogowym wejściem z możliwością nadpróbkowywania i częstotliwością próbkowania 10 μs. Przebiegi podzielono na 4 klasy: poprawny, wadliwe pierwsze uderzenie, wadliwe drugie uderzenie oraz brak drugiego uderzenia. W pracy [2] rozważono trzy warianty: z 256 atrybutami, z 3 atrybutami, z jednym atrybutem po przekształceniu dyskretną transformatą Fouriera oraz 6 metod klasyfikacji: pojedyncze drzewo decyzyjne (SDT), probabilistyczna sieć neuronowa (PNN), maszyna wektorów nośnych (SVM), wielowarstwowy perceptron (MLP), liniowa analiza dyskryminacyjna (LDA), metoda K średnich (K-Means). Metody te były porównywane z użyciem procedury dziesięciokrotnej walidacji krzyżowej. Wyniki eksperymentów pokazały, że najlepszej klasyfikacji dokonywały SVM oraz PNN, osiągając dokładność na poziomie 99%. Zarys rozwiązania omawianego problemu autor przedstawił w pracy [107], która przedstawia kolejny wariant rozwiązania oraz jego implementację na FPGA opartą na komórce LSTM zawierającą funkcje aktywacji przedstawione w rozdziale 3.3.2. Porówano szybkość oraz dokładność w stosunku do implementacji na PC oraz do wyników uzyskanych w pracy [2]. Komputer PC dysponował procesorem 7-8550U, 8GB pamięci RAM



Rysunek 3.34: Sygnał drugiego uderzenia z czujnika piezoelektrycznego

oraz posiadał system operacyjny Windows 10. Dane dostarczone przez autorów [2] były pomniejszone o przypadki wadliwego pierwszego uderzenia oraz braku drugiego uderzenia, co ograniczyło problem do klasyfikacji binarnej przypadków poprawnego procesu oraz wadliwego drugiego uderzenia. Przykładowy przebieg sygnału dla obu przypadków znajduje się na rysunku 3.34. Dane składały się z 264 przebiegów, z których każdy zawiera 256 próbek. 131 przebiegów reprezentowało przypadki wadliwe, zaś 133 przypadki poprawne.

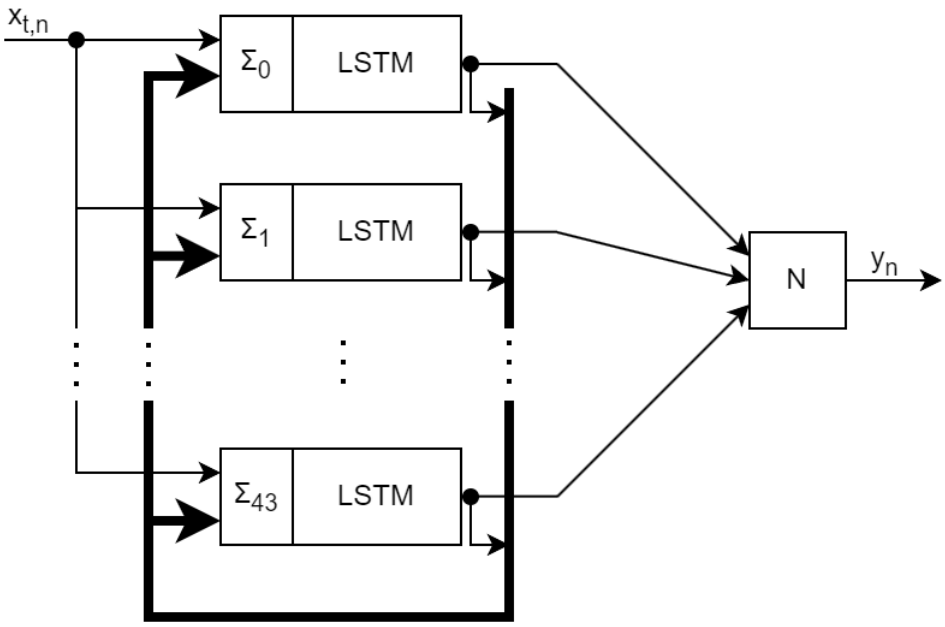
W niniejszej pracy przedstawiono znacznie rozszerzony zakres tematyczny. Przedstawione zostały cztery warianty implementacyjne, wykorzystujące komórki LSTM opisane w 3.3.5. Pierwszym wariantem była komórka LSTM oparta o ulepszoną wersję funkcji aktywacji, w której sposób obliczeń eksponenty wzorowany był na algorytmie CORDIC (ale bezpośrednio z niego nie korzystał). Ulepszenie dotyczyło zmniejszenia zużywanych zasobów FPGA oraz zwiększenia szybkości działania funkcji aktywacji. Drugim wariantem była implementacja z wykorzystaniem funkcji aktywacji z użyciem wielomianów Czebyszewa 4. stopnia oraz 15 przedziałów. W trzecim wariancie wykorzystano funkcje aktywacji oparte o zwykłe wielomiany 4. stopnia oraz 15 przedziałów. W czwartym wariancie wykorzystano również zwykłe wielomiany, natomiast stopień

wielomianów ograniczono do 2., a liczbę przedziałów zwiększono do 186. Warianty zostały przedstawione w punkcie 3.3.5 i 3.4.

3.5.3. Architektura sieci LSTM

Ze względu na ograniczone zasoby sprzętowe w układach FPGA, głównym założeniem podczas określania architektury sieci było wykorzystanie minimalnej liczby neuronów wystarczającej do wykonania poprawnie zadania klasyfikacji. Detekcja wadliwego uderzenia może być rozpatrywana jako problem klasyfikacji binarnej, gdzie jedna klasa oznacza poprawne uderzenie, a druga wadliwe. Jak wspomniano w punkcie 3.5.2, dane na których operowano były zorganizowane w szeregi czasowe o długości 256 próbek. Pierwszym etapem było rozdzielenie 264 przebiegów na dane uczące i testowe. Przyjęto podział na 176 przebiegów uczących i 88 testowych. Drugim etapem było ograniczenie szeregów do mniejszej liczby próbek. Analizując przebiegi sygnałów, których przykłady przedstawione zostały na rys. 3.34, zauważyć można, że najwięcej wartościowych informacji jest w początkowej fazie i dotyczy próbek o indeksie poniżej 100. Powyżej tego indeksu poszczególne przebiegi nie różnią się znacząco. Wobec tego, wpierw postanowiono ograniczyć szeregi do indeksów poniżej 100, a następnie zmniejszać gęstość próbek tak, żeby utrzymać efekt uczenia sieci na zadowalającym poziomie. Na tym etapie eksperymenty wykonywane były na sieci składającej się z dwóch warstw (rys. 3.35): pierwszej LSTM, składającej się z 64 elementów i drugiej wyjściowej, składającej się z jednego neuronu z sigmoidalną funkcją aktywacji. W trakcie uczenia wykorzystywano funkcję straty w postaci binarnej entropii krzyżowej, metryki "acc" odpowiadającej dokładności oraz algorytmu optymalizacyjnego Adam. Uczenie wykonano za pomocą pakietu Keras i języka Python.

W rezultacie eksperymentów okazało się, że minimalną liczbą punktów jakie wystarczają do utrzymania efektów uczenia sieci na poziomie 100% jest 15 równo rozmieszczonych punktów poniżej indeksu 100 (co siódmy element szeregu). Kolejnym etapem było znalezienie minimalnej liczby neuronów w warstwie LSTM. W tym celu dla różnych algorytmów optymalizacyjnych przyuczano sieci składające się z różnej liczby neuronów. Początkowo liczba elementów w warstwie wynosiła 64 po czym zmniejszano co 2. Dla każdej liczby neuronów i dla każdego algorytmu optymalizującego wykonywano 12 cykli uczenia, w których mierzono dokładność klasyfikacji na szeregach testowych. Do porównania wybierano najwyższą wartość dokładności osiągniętą dla każdej z kombinacji. Wyniki z eksperymentu zamieszczono w tabeli 3.16.



Rysunek 3.35: Diagram połączeń bloków w implementowanej sieci

Tabela 3.16. Dokładności klasyfikacji sieci dla różnych optymalizatorów i różnej liczny neuronów

Liczba Neuronów	Adam	Rmsprop	SGD	Adamax	Nadas
58	100%	100%	100%	100%	100%
56	100%	100%	98%	100%	100%
54	100%	100%	57%	100%	100%
52	100%	100%	53%	100%	100%
50	100%	100%	47%	100%	100%
48	100%	100%	53%	100%	100%
46	100%	81%	47%	100%	100%
44	100%	86%	53%	100%	81%
42	95%	69%	47%	97%	83%
40	86%	61%	47%	53%	53%

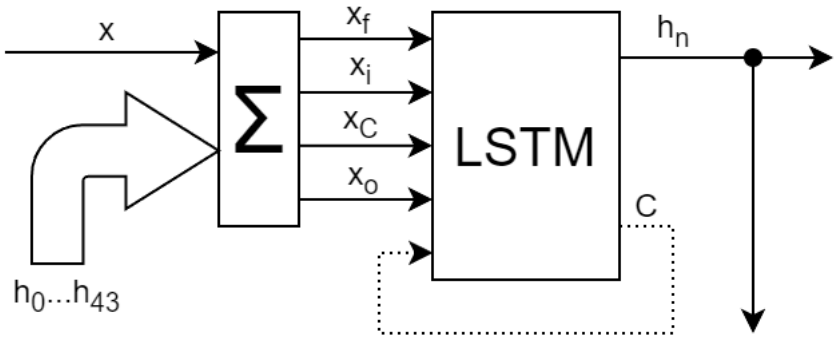
Dla liczby neuronów większej lub równej 58 różnica nie była zauważalna i dlatego przypadków > 58 nie umieszczono w tabeli. Dopiero od wartości 56 można zauważyć spadek dokładności dla algorytmu SGD. W przypadku Rmsprop spadek nastąpił już poniżej 48 neuronów, a dla Nadas poniżej 46. Najmniejszą liczbę neuronów udało się osiągnąć dla algorytmu Adam i Adamax, i wyniosła ona 44.

Na podstawie eksperymentów, w dalszej części pracy przyjęto 44 neurony w warstwie LSTM oraz długość sekwencji wynoszącą 15 punktów.

3.5.4. Implementacja sieci

Wszystkie warianty implementacyjne zbudowane są w oparciu o taką samą architekturę sieci i korzystają z takiej samej reprezentacji liczbowej, zgodnej z IEEE754. W implementacji zrezygnowano z poprzednio wykorzystywanego oprogramowania ISE Design Suite oraz układu Artix-7 XC7A100T-3CSG384, ze względu na spodziewany rozmiar logiki, na rzecz układu XCU250-FIGD2104-2L-E z płyty Alveo U250 Data Center Accelerator Card oraz programu Vivado, który wspiera tworzenie logiki dla tego układu. Sieć składała się z warstwy rekurencyjnej, zbudowanej z 44 komórek, na którą podawany był analizowany, wcześniej obrobiony sygnał oraz warstwy wyjściowej, zbudowanej z jednego bloku, do którego wchodziły sygnały z każdej z komórek LSTM. Układ bloków uwidoczniono na rysunku 3.35. Sygnały wyjściowe (co charakterystyczne dla sieci rekurencyjnych) były również skierowane na wejścia komórek LSTM i wykorzystywane w obliczeniach w kolejnych iteracjach, równolegle z analizowanym sygnałem wejściowym. Na początku obliczeń sygnały rekurencyjne miały wartość zero, natomiast w każdym kolejnym kroku należało je uwzględnić w obliczeniach zgodnie z równaniami (3.1) - (3.4), w których każdy z sygnałów był przemnażany przez odpowiadającą mu wagę. Przemnażany przez swoją wagę był również analizowany sygnał, całość sumowano i dodawano do wyniku odpowiednią wartość przesunięcia. Każda z bramek w komórce LSTM posiadała swój zestaw wag dla każdego sygnału i swoją wartość przesunięcia. Opisywane obliczenia wykonywano w bloku sumatora, który jest widoczny na rys. 3.36, oznaczony jako Σ. Ze względu na liczbę działań, które muszą zostać wykonane, blok sumatora może wprowadzać duże opóźnienie lub duże zużycie zasobów. Obliczenia wykonywane w sumatorze nie kolidują z tymi, wykonywanymi w komórce LSTM, więc możliwe jest wykorzystanie bloków arytmetycznych przeznaczonych do obliczeń funkcji aktywacji.

Jak przedstawiono w punkcie 3.3, każda z wykorzystywanych funkcji aktywacji wykorzystywała blok dodająco-odejmujący oraz mnożący, a bloków realizujących funkcje aktywacji było 3 w każdej z komórek, jak przedstawiono w punkcie 3.4. Wykorzystano zatem 3 pary bloków arytmetycznych, do których można było dodać kolejne, zastosowane jedynie w sumatorze. W tabeli 3.17 przedstawiono trzy podejścia: z dodaniem 1 pary bloków arytmetycznych (tak żeby każda para realizowała obliczenia



Rysunek 3.36: Komórka LSTM z sumatorem

dla każdej z bramek z osobna), z dodaniem 5 oraz 13, co odpowiada odpowiednio 4, 8 i 16 parom bloków. Sposób wykonywania obliczeń dla 8 par przedstawiono jako algorytm 3.1. Należy mieć na uwadze, że algorytm 3.1 realizował obliczenia związane tylko z jedną bramką, na którą wykorzystano 2 pary bloków arytmetycznych. Dla pozostałych trzech bramek zastosowano pozostałe pary, a obliczenia były analogiczne. Pula wejść została podzielona na dwie części, w których każda była osobno obliczana, a następnie wyniki były ze sobą łączone. Pętla 1 i pętla 2 wykonywane były równolegle.

Algorytm 3.1. Obliczanie wartości argumentu funkcji aktywacji

Wejście: tablica sygnałów na wejściu komórki S[ ];
tablica wartości wag sygnałów wejściowych w[ ];
liczba neuronów n; wartość przesunięcia b;

Wyjście: Argument funkcji aktywacji Xtemp

1.**Ustaw** Xtemp1 = 0; **Ustaw** Xtemp2 = 0;
2.**for** i = 0 < n/2 **do** //pętla 1
3. $Xtemp1 = Xtemp1 + S[i] \cdot w[i]$;
4.**endfor**
5.**for** j = 0 < n/2 **do** //pętla 2
6. $Xtemp2 = Xtemp2 + S[n/2 + j] \cdot w[n/2 + j]$;
7.**endfor**
8.$Xtemp = Xtemp1 + Xtemp2 + b$;

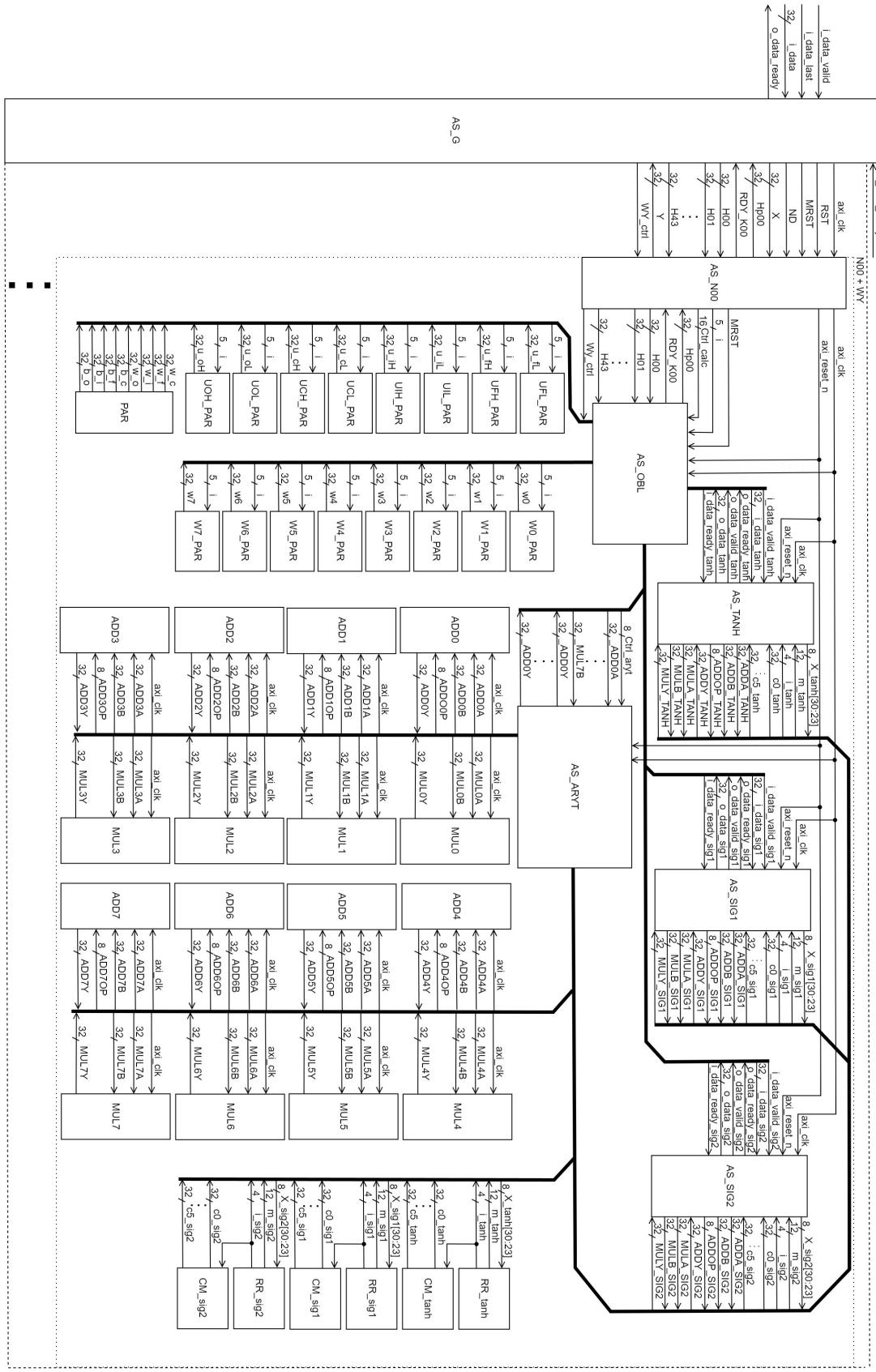
Tabela 3.17. Porównanie sumatorów (XCU250-FIGD2104-2L-E)

Liczba par bloków	LUT	FF	DSP	Liczba taktów zegara
16	8020	1467	64	43
8	4954	831	32	79
4	2423	401	16	142

Tabela 3.17 pokazuje, iż decyzja ta skutkowała mniejszym czasem obliczeń niemal o połowę. Liczba cykli dla 8 par była porównywalna z liczbą cykli potrzebnych na przeprowadzenie obliczeń w komórce LSTM dla wariantu 2 i 3. Wykorzystano przy tym 4954 tablic LUT, 831 FF i 32 DSP. W przypadku 4 par, zużycie zasobów sprzętowych było znacznie mniejsze, lecz liczba cykli zegara była za duża w porównaniu z liczbą cykli potrzebnych na wykonanie obliczeń w samej komórce. Z kolei przy 16 parach wykorzystanie zasobów było zdecydowanie za duże. W związku z tym, w dalszej części wykorzystywany był sumator oparty o 8 par bloków arytmetycznych - niezależnie od wykonywanego wariantu.

Poza implementacją sumatora i komórki LSTM, ostatnim elementem pozostającym do zdefiniowania był element z warstwy wyjściowej przedstawiony na rys. 3.35 jako element *N*. Jest to prosty komponent składający się z sumatora oraz funkcji aktywacji. Sumator zaimplementowany w tym elemencie był prostszy od tych implementowanych w warstwie LSTM, z jednym wyjściem, bez potrzeby uwzględniania bezpośrednio analizowanego sygnału. Ze względu na to, że obliczenia w warstwie wyjściowej wykonywane były sekwencyjnie po wykonaniu wszystkich obliczeń w warstwie rekurencyjnej, można zatem było wykorzystać którąś z komórek LSTM do wykonania obliczeń. Wykorzystane przy tym mogą zostać istniejące bloki arytmetyczne oraz blok funkcji sigmoidalnej. Dołożenie tej funkcjonalności sprowadziło się do rozbudowania automatu sekwencyjnego oraz dodania oddzielnych sygnałów sterujących, decydujących o wykonywanej funkcji w jednej z komórek, co też zostało wykonano dla komórki o indeksie 0.

Jak wspominano, architektura sieci została zaprojektowana oraz przyuczona najpierw na komputerze PC, korzystając z biblioteki Keras języka Python. Wobec tego, obliczone wagi i przesunięcia mogły zostać wyeksportowane i przeniesione do struktury FPGA. Parametry sieci zostały więc zaszyte w poszczególnych sumatorach. Przedstawione w tabeli 3.17 sumatory zawierały już pakiet parametrów dla jednej komórki LSTM.



Rysunek 3.37: Struktura logiczna pełnej sieci LSTM

Schemat logiczny modułu sieci przedstawiony został na rys. 3.37. Przed rozpoczęciem pracy z modułem należało na jeden cykl zegara ustawić sygnał axi_reset_n w celu początkowej inicjalizacji sieci. Interfejs wejściowy modułu był zgodny ze standardem AXI4-Stream i był typu slave. Na wejście i_data podano kolejno wartości z analizowanego szeregu. Podanie kolejnej wartości musi zostać potwierdzone wystawieniem sygnału i_data_valid rozpoczynającego obliczenia. O zakończeniu obliczeń związanych z kolejnymi wartościami sieć sygnalizuje, wystawiając sygnał o_data_ready, wtedy to właśnie następna wartość może zostać podana i potwierdzona sygnałem i_data_valid. Podanie ostatniego elementu z sekwencji było sygnalizowane poprzez ustawienie portu i_data_last w trakcie trwania przesyłu ostatniej sekwencji. Jest to informacja, dotycząca wykonania obliczeń związanych z tą wartością, po której moduł może zacząć obliczenia związane z warstwą wyjściową. Po zakończeniu tychże wykorzystywany był interfejs wyjściowy, również zgodny z AXI4-Stream, lecz typu master. Po ustawieniu wartości wynikowej na porcie o_data, ustawiany był w stan wysoki port o_data_valid. W przypadku, jeśli i_data_ready był ustawiony w stan aktywny, można było rozpocząć przesył. Po zakończeniu przesyłu danych, sygnał o_data_valid był ustawiany w stan niski.

Poza interfejsami, na rysunku 3.37 przedstawiono zestaw bloków dla pojedynczej, rozszerzonej komórki. Prócz wykonywania obliczeń, jak w przypadku pozostałych komórek, realizowała ona również funkcjonalność warstwy wyjściowej. Ze względu na dużą liczbę elementów związanych z pozostałą liczbą komórek (łącznie 44 komórki LSTM) nie uwzględniano ich na schemacie. Głównym blokiem odpowiedzialnym za obsługę interfejsów oraz koordynowanie pracy komórek był blok AS_G, który składał się z głównego automatu sekwencyjnego, w ramach którego aktualizowane były wartości rekurencyjne w sieci, a obliczenia synchronizowane w komórkach. W nim bowiem odczekuje się na zakończenie pracy każdej komórki i ustawienie nowej, poprawnej wartości sygnału rekurencyjnego $Hpxx$. Wartość ta później była przepisywana do Hxx i użyta w kolejnym kroku. Synchronizacja odbywała się w oparciu o sygnały RDY_Kxx, zaś początek nowej tury obliczeń sygnalizowany był poprzez ustawienie sygnału ND, który był jednakowy dla każdego modułu komórki LSTM. Pierwsza komórka, ze względu na wbudowanie funkcjonalności warstwy wyjściowej posiadała dodatkowy sygnał sterujący WY_ctrl, który sygnalizował potrzebę przełączenia funkcjonalności przez moduł. Pozostałe komórki również wyposażone były w ten port,

natomiast jego zadaniem było w tym przypadku przekazanie informacji o ignorowaniu sygnału ND, jeśli WY_ctrl był w stanie aktywnym. Do każdego modułu komórki doprowadzone zostały sygnały rekurencyjne oraz sygnał wejściowy. Głównym modułem logicznym komórki był blok AS_Nxx, odpowiedzialny przede wszystkim za obsługę interfejsów komórki i zarządzanie obliczeniami. Obliczenia wykonywane były w bloku AS_OBL, do którego przekazywano wartości sygnałów oraz sygnał sterujący pracą komórki $Ctrl\_calc$. Blok ten funkcjonował w oparciu o automat sekwencyjny, bezpośrednio połączony z 8 parami bloków arytmetycznych (8 układów dodająco-odejmujących oraz 8 układów mnożących) za pomocą których wykonywano wszystkie obliczenia. Blok skomunikowany był z modułami odpowiedzialnymi za obliczenia wartości funkcji aktywacji. Na schemacie przedstawione są bloki realizujące obliczenia związane z funkcjami aktywacji działające w oparciu o wielomiany Czebyszewa (wariant 2), natomiast w trakcie eksperymentów sprawdzono wszystkie cztery warianty. Modułów realizujących funkcje aktywacji było trzy: 2 moduły liczące wartość funkcji sigmoidalnej oraz 1 liczący wartość funkcji tangensa hiperbolicznego. Bloki wyposażono w interfejsy zgodne z AXI4-Stream, poprzez które następowała komunikacja. Wykorzystano w nich również współdzielone z AS_ARYT bloki arytmetyczne. W związku ze współdzieleniem zasobów komunikacja odbywała się za pośrednictwem AS_ARYT, pośredniczącym w dystrybucji sygnałów do oraz z bloków arytmetycznych. Parametry wykorzystywane w trakcie obliczania wartości funkcji aktywacji były pobierane z bloków $RR\_$ i $CM\_$, zaś wartości wag połączeń wchodzących i rekurencyjnych oraz przesunięć z bloku PAR i U_PAR. Ze względu na pełne wykorzystanie bloków arytmetycznych i podział puli sygnałów rekurencyjnych na 2 dla każdej z bramek bloków U_PAR było 8. W przypadku wykonywania obliczeń związanych z warstwą wyjściową potrzebne parametry pobierane były z bloków W_PAR, gdzie pula sygnałów podzielona została na 8 części.

Sprawdzenie poprawności działania projektowanej logiki odbywało się na drodze symulacji w module symulacyjnym programu Vivado, gdzie porównano wyuniki z uzyskanymi przez program referencyjny napisany w Pythonie, na różnych etapach uruchamiania. Uruchamianie rozpoczęto od pojedynczej komórki o indeksie 00. Początkowo sprawdzono działanie sumatora, którego funkcjonalność zawarto w bloku AS_OBL. W tym celu wyodrębniono w symulacji model całej komórki (blok wyznaczony przerywaną linią i etykietą "$N00 + WY$" na rysunku 3.37), a następnie określono warto-

ści na każdym wejściu danych tj. na wszystkich wejściach sygnałów rekurencyjnych $H00...H43$ oraz X, następnie określono wartości sygnałów sterujących tak, żeby zasymulować cykl komunikacyjny. W tym czasie, na wejściu zegarowym tj. axi_clk podawany był symulowany sygnał zegara. Ze względu na znane wartości zliczanych sygnałów na wejściu, można było tym samym prześledzić pracę sumatora i sprawdzić wartości koljenych zliczanych sygnałów. W omawianej implementacji sumator zliczał sygnał wejściowy dla każdej bramki osobno. Test na tym etapie wykonano dla 100 różnych zestawów wartości sygnałów wejściowych i porównano z analogicznymi obliczeniami wykonanymi w programie referencyjnym. Kolejnym krokiem było pominięcie sumatora i ustawienie na jego wyjściu konkretnych wartości sygnałów, a następnie sprawdzenie działania bloków funkcji aktywacji. Automat sekwencyjny z bloku AS_OBl startował wtedy od stanu, w którym następowało przesłanie danych do bloków funkcji aktywacji i wysterowanie ich do rozpoczęcia obliczeń. Wartości na wejściach dobierano tak, żeby pokrywały się z wykorzystywanymi w punktach 3.3.2, 3.3.3 oraz 3.3.4, tak żeby można było wykorzystać dane tam uzyskane jako wartości docelowe. Sprawdzenie odbywało się na 100 różnych zestawów wartości. Dalszym krokiem było uruchomienie w całości bloku komórki, również dla 100 różnych zestawach danych wejściowych (innych niż poprzednio) oraz po przywróceniem stanu kodu sprzed zmian dla testu bloków funkcji aktywacji. Dla każdego zestawu danych inicjowano komunikację oraz sprawdzano wartość sygnału $Hp00$ oraz wartość stanu wewnętrznego komórki. Kolejnym krokiem było sprawdzenie funkcjonalności modułu dla realizacji obliczeń związanych z warstwą wyjściową. Ze względu na to, że bloki funkcji aktywacji zostały już przetestowane, skupiono się na tym etapie na sprawdzeniu sumatora, co odbyło się w sposób analogiczny do poprzedniego, a następnie do sprawdzenia działania całego modułu w tym ustawieniu. Ostatnim etapem weryfikacji było sprawdzenie całej sieci, które polegało na sprawdzeniu wartości sygnałów wyjściowych komórek po każdym kroku obliczeniowym oraz po przetworzeniu całej sekwencji. Takie testy wykonano dla 10 sekwencji testowych za każdym razem porównując wyniki z programem referencyjnym.

3.5.5. Wyniki implementacji

Wyniki implementacji przy wykorzystaniu programu Vivado dla układu XCU250-FIGD2104-2L-E z Alveo U250 Data Center Accelerator Card przedstawiono w tabeli 3.18. Dla każdej implementacji osiągnięto taki sam poziom dokładności klasyfikacji, który został ujęty w tablicę pomyłek, w tabeli 3.19. Spośród 88 danych testowych do klasy 1 należało 41 przebiegów spośród których wszystkie zostały prawidłowo przypisane do klasy 1, do klasy 0 należało 47 przebiegów spośród których wszystkie zostały poprawnie przypisane do klasy 0.

Tabela 3.18. Porównanie wykorzystanie zasobów sprzętowych (XCU250-FIGD2104-2L-E)

Wariant	LUT	FF	DSP
Wariant 1	308977	84455	1408
Wariant 2	219972	136299	1408
Wariant 3	195431	126647	1408
Wariant 4	250629	127245	1408
Dostępne	1728000	3456000	12288

Tabela 3.19. Tablica pomyłek

	True class	
	Positives:	Negatives:
Predicted True	41	0
Predicted False	0	47

Zużycie zasobów różni się znacząco pomiędzy poszczególnymi wariantami. Najwięcej tablic LUT wykorzystane zostało przy implementowaniu wariantu 1. Jednocześnie wykorzystano wtedy najmniej przerzutników. W docelowym układzie FPGA jest znacznie więcej przerzutników niż tablic LUT (liczba dostępnych zasobów sprzętowych znajduje się w tabeli 3.18, a szerszy opis wykorzystanych układów znajduje się w punkcie 4.5) przez co, przy projektowaniu, większą wagę powinno się zwrócić na wykorzystanie tablic LUT. W porównaniu z wariantem 3, który zużywa najmniej tablic LUT, wariant 1 zużywa ich 1.58 razy więcej.

Tabela 3.20. Porównanie czasów obliczeń

Wariant	Czas obliczeń	Liczba taktów zegara	Częstotliwość zegara [MHz]	Estymowana moc[W]	Błąd maksymalny
Wind+Keras+GPU1	30.8 ms	-	-	-	-
Ubun+Keras+GPU2	32.0 ms	-	-	-	-
Python+CPU	46.9 ms	-	-	-	-
Raspberry Pi 3	11.2 ms	-	-	-	-
Wariant 1	65.9 μs	3033	46	3.9	$5.6 \cdot 10^{-3}$
Wariant 2	19.2 μs	2473	129	11.1	$2.6 \cdot 10^{-3}$
Wariant 3	19.2 μs	2473	129	10.4	$2.9 \cdot 10^{-3}$
Wariant 4	17.5 μs	2101	120	10.6	$3.8 \cdot 10^{-3}$
Z artykułu [107]	251.8 μs	7553	30	-	-

W tabeli 3.20 zawarto zestawienie czasów obliczeń oraz informację na temat pobieranej mocy (odczytaną z Vivado) dla prezentowanych wariantów. Zawarto również informacje na temat czasu obliczeń referencyjnych wykonywanych na Raspberry Pi 3 z użyciem C++, na komputerze PC z wykorzystaniem Pythona i CPU oraz na dwóch różnych komputerach PC wykorzystujących bibliotekę Keras i GPU. Ze względu na czytelność, w oddzielnej tabeli 3.21, przedstawiono stosunki poszczególnych czasów obliczeń. Pomiar czasu dotyczył jednego przebiegu, co miało odzwierciedlić rzeczywistą sytuację, gdy obliczenia były wykonywane zaraz po skompletowaniu pojedynczej sekwencji danych. Sprzęt na którym wykonywano obliczenia referencyjne został opisany w punkcie 2.

Tabela 3.21. Zestawienie czasów obliczeń

Wariant	Czas obliczeń	$\frac{T_{GPU1}}{T_x}$	$\frac{T_{GPU2}}{T_x}$	$\frac{T_{PC}}{T_x}$	$\frac{T_{RPi3}}{T_x}$
Wind+Keras+GPU1	30.8 ms	1.00	1.04	1.52	0.36
Ubun+Keras+GPU2	32.0 ms	0.96	1.00	1.47	0.35
Python+CPU	46.9 ms	0.66	0.68	1.00	0.24
Raspberry Pi 3	11.2 ms	2.75	2.86	4.19	1.00
Wariant 1	65.9 μs	467	486	712	170
Wariant 2	19.2 μs	1604	1667	2443	583
Wariant 3	19.2 μs	1604	1667	2443	583
Wariant 4	17.5 μs	1760	1829	2680	640

Można zauważyć, że poprawiona implementacja z funkcją aktywacji wzorowaną na CORDIC była w stosunku do wersji z [107] znacznie szybsza i operowała na wyższej częstotliwości zegara. Pomimo osiągnięcia ogólnej poprawy wyników na drodze optymalizacji pierwotnego rozwiązania, lepsze wyniki zostały osiągnięte w przypadku wariantów 2-4. Najkrótszy czas obliczeń uzyskał wariant 4 i wyniósł 17.5 μs. Nieznacznie dłużej wykonywane były obliczenia w przypadku wariantów 2 i 3, bo w 19.2 μs. Należy przy tym zwrócić uwagę, że w przypadku wariantu 3, nieznacznie dłuższe obliczenia w stosunku do wariantu 4 wiążą się ze znacznie mniejszym zużyciem zasobów - 55 tys. tablic LUT mniej. Porównując otrzymane wyniki z tymi osiągniętymi przez programy referencyjne zauważalne jest, że implementacje na FPGA są dużo szybsze, co pokazano w tabeli 3.21. Najwolniejszy z wariantów na FPGA jest wciąż 467 razy szybszy niż obliczenia za pośrednictwem GPU1 oraz 486 razy szybszy niż GPU2, natomiast już tylko 170 razy szybszy niż obliczenia w C++ na Raspberry Pi 3. Jak można było się spodziewać, najdłużej obliczenia wykonywane były z wykorzystaniem Pythona i CPU. Zaskakującym wynikiem cechowało się wykorzystanie języka C++ na Raspberry Pi 3, które poskutkowało lepszym czasem obliczeniowym niż w przypadku biblioteki Keras i GPU. Pomimo tego, iż spośród obliczeń referencyjnych implementacja oparta na C++ była najszybsza, to w porównaniu z najszybszą implementacją na FPGA i tak okazała się 640 razy wolniejsza. Zaletą pakietu Vivado jest możliwość analizy i estymacji pobieranej mocy przez zaprojektowaną logikę.

W przypadku wariantu 1, który przy analizie pod kątem czasu wypadł najgorzej, w przypadku analizy estymacji pobieranej mocy wypada najlepiej - 3.9 W. To kilkukrotnie mniejsza pobierana moc niż w pozostałych wariantach, które pobierały moc nieznacznie powyżej 10 W. Należy przy tym pamiętać, że wartość pobieranej mocy również była wynikiem obliczeń programu Vivado.

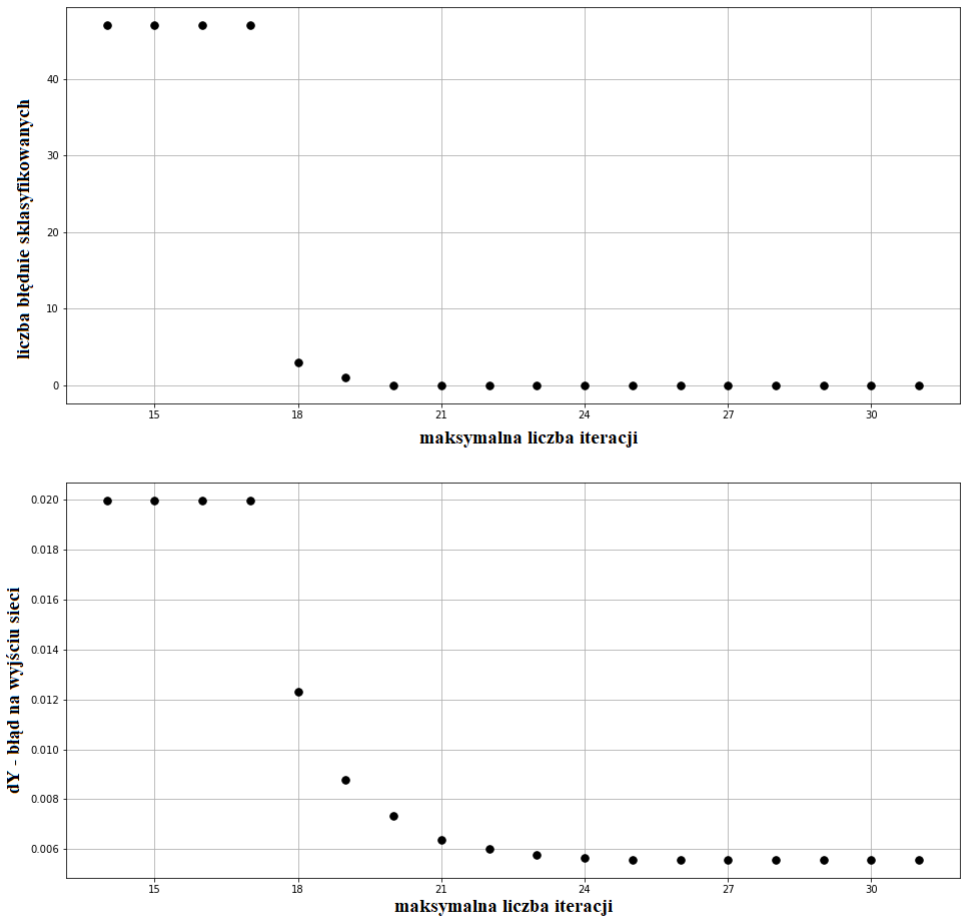
Przedstawione wyniki są bardzo obiecujące w kontekście potencjalnych zastosowań. Pomimo, że w eksperymencie rozważano ogólne zastosowanie (bez dodawania interfejsów zewnętrznych typu I2C lub SPI, próbkowania sygnałów, zbierania i obróbki danych), to otrzymane wyniki pozwalają sądzić, że układy FPGA mogą konkurować z popularnymi obecnie podejściami do obliczeń związanych z rekurencyjnymi sieciami neuronowymi. Mniej korzystnie wygląda cena zastosowania bardzo dużych układów FPGA, jakim miedzy innymi jest wykorzystany tutaj układ XCU250-FIGD2104-2L-E z Alveo U250 Data Center Accelerator Card, co zostało przedstawione w punkcie

4.5, gdzie odniesiono wyniki otrzymywane w opisywanej pracy do układów dostępnych na rynku oraz ich cen.

3.5.6. Wpływ dokładności aproksymacji na klasyfikację

Ze względu na uczenie sieci na komputerze PC i przeniesienie wartości wag oraz przesunięć na FPGA, konieczne było zapewnienie odpowiedniej dokładności obliczeń dla implementacji na FPGA. Szczególnie ważne było to ważne ze względu na rekurencyjny typ implementowanej sieci, w której błąd mógłby znacząco narosnąć i pogorszyć jakość klasyfikacji. W trakcie eksperymentów zauważono, że maksymalna wartość wyjścia dla poprawnego uderzenia wynosiła 0.4973246, zaś minimalna wartość dla wadliwego uderzenia 0.50793844. Wartości te różniły się o 0.0106138, co prawdopodobnie łatwo przekroczyć przy mało dokładnych implementacjach funkcji aktywacji.

Korzystając z zalety implementacji funkcji aktywacji z wariantu 1, polegającej na tym, że dokładność obliczeń może zostać łatwo zmieniona poprzez ograniczenie liczby iteracji, możliwe było wykonanie eksperymentu sprawdzającego, jaka dokładność funkcji aktywacji jest wymagana w celu przeprowadzenia poprawnej klasyfikacji oraz sprawdzenie tego, jak wpływa ona na obliczenia. Dla liczby iteracji od 14 do 31 badano wyniki klasyfikacji oraz wartość liczbową wyjścia. Wyniki znajdują się na rys. 3.38. Liczbę iteracji można łatwo odnieść do osiąganej dokładności wspomagając się rys. 3.20 oraz rys. 3.21. Poniżej 18 iteracji (dokładność na poziomie 10^{-4}) błędnie sklasyfikowanych było 47 szeregów na 88. Dla 19, błędnie sklasyfikowany był tylko jeden szereg, natomiast od 20 w górę osiągnięto zakładaną dokładność klasyfikacji. Wartość liczbową odniesiono później do wartości zwracanych przez model pracujący na PC z użyciem biblioteki Keras i zamieszczono na dolnym wykresie na rys. 3.38. Wartość dY jest maksymalną wartością spośród różnic wartości na wyjściu sieci pomiędzy LSTM implementowanym na FPGA i z użyciem Keras, dla całego zestawu danych testowych przy konkretnym ograniczeniu liczby iteracji. Różnica dla liczby iteracji większej od 20 była na poziomie $5.6 \cdot 10^{-3}$ pomimo, że same obliczenia funkcji aktywacji osiągały wartość błędu na poziomie 10^{-5} i mniej. Warte zaznaczenia jest, iż końcowa dokładność obliczeń (dla $i = 31$) dla pozostałych wariantów była bardzo podobna (tabela 3.5). Błąd maksymalny dla wariantu 2 wyniósł $2.6 \cdot 10^{-3}$, dla wariantu 3 - $2.9 \cdot 10^{-3}$, zaś dla wariantu 4 - $3.8 \cdot 10^{-3}$. Takie same wyniki otrzymano niezależnie od szerokości dziedziny, zarówno dla $< -6, 6 >$ jak i dla $< -10, 10 >$.



Rysunek 3.38: Analiza wymaganej dokładności obliczeń

Biorąc pod uwagę przyrost błędu zaprezentowany w tym punkcie oraz realizowany później (w punkcie 4) moduł uczenia sieci, który może wykonywać znacznie więcej operacji matematycznych, przyjęto na początku pracy, że docelowa dokładność funkcji aktywacji powinna być na poziomie 10^{-7} dla zakresu $< -6,6 >$. Z rozważań w tym punkcie wynika, że jest to nieco zawyżona wartość. W punkcie 4 rozważania zostały rozszerzone do procesu uczenia sieci, co pozwoliło na pełne zweryfikowanie tego założenia.

3.5.7. Podsumowanie

W podrozdziale przedstawiono implementacje sieci LSTM będących klasyfikatorami do procesu kucia na zimno. W każdym z prezentowanych wariantów otrzymano dużą dokładność obliczeń względem programów referencyjnych oraz 100% dokładność klasyfikacji. Uzyskano również znacznie krótszy czas obliczeń pojedynczej klasyfikacji, niż w przypadku programów referencyjnych - nawet w stosunku do obliczeń na GPU. Najszybszy wariant (wariant 4) był szybszy 1760 razy w stosunku do GPU1 (NVIDIA GeForce GTX 1060) i 1829 od GPU2 (NVIDIA GeForce RTX 3080 Ti). Powodem tego był mały wymiar obliczeń, który nie pozwolił na wykorzystanie w pełni możliwości kart graficznych. Ukazało to, iż prowadzenie obliczeń na GPU ma swoje ograniczenia, które wyraźnie ustępują obliczeniom na FPGA w niektórych przypadkach. Każdy z prezentowanych wariantów różnił się przede wszystkim funkcją aktywacji, natomiast w przypadku wariantu 1 dodatkową różnicę stanowiło ograniczenie liczby iteracji podczas eksperymentów związanych z badaniem wpływu dokładności. W przypadku zadania klasyfikacji, wariant 1 wypadł najsłabiej spośród wszystkich wariantów, natomiast ze względu na łatwą kontrolę nad osiąganą dokładnością aproksymacji funkcji aktywacji, umożliwił przeprowadzenie analizy wpływu dokładności na działanie sieci. W omawianym przypadku okazało się, że dokładność aproksymacji jest istotna i zadowalające wyniki osiągano już dla maksymalnego błędu na poziomie 10^{-5}. Porównując między sobą warianty 3 i 4, które są wykonywane według tego samego podejścia, lecz w przypadku wariantu 4 zredukowano stopień wielomianu, a zwiększono liczbę przedziałów, można zauważyć że różnica pomiędzy podejściami maleje jeśli chodzi o czas obliczeń, natomiast porównanie zużycia zasobów wypada zauważalnie lepiej dla wariantu 3. Biorąc pod uwagę wyniki implementacji, można zauważyć użyteczność tego oraz podobnych rozwiązań. Szczególnie szybkość i estymowana moc zasługują na uwagę. Ograniczeniem przy projektowaniu systemu mogą być dostępne układy FPGA, ich cena oraz ich dostępne zasoby sprzętowe. Podczas implementacji zrezygnowano z poprzedniego docelowego układu FPGA, na który budowano funkcje aktywacji, ze względu na niewystarczającą liczbę bramek LUT, przerzutników i bloków DSP. Zdecydowano się na układ będący częścią karty obliczeniowej Alveo U250. Znajdujący się tam układ jest obecnie jednym z większych obsługiwanych domyślnie przez Vivado. Wzrastające ostatnio zainteresowanie układami FPGA, a szczególnie dobre wyniki przy wykorzystaniu tych układów do realizacji modeli uczenia maszynowego,

pozwala sądzić, że dostępność układów FPGA oraz dostępne w nich zasoby będą wzrastać i przedstawione w tym punkcie implementacje będą z powodzeniem realizowane.

W opisanych przypadkach mierzono czas samych obliczeń, istotne opóźnienie może natomiast nastąpić jednak na interfejsach. Korzystając z FPGA można ten problem rozwiązać poprzez np. dopasowanie struktury logicznej tak, by bezpośrednio współpracowała z urządzeniami peryferyjnymi oraz uniezależnić proces klasyfikacji od zajętości układu obliczeniowego innymi procesami. Porównując rozmiar sieci oraz biorąc pod uwagę, że układy FPGA są wciąż rozwijane, widoczny jest potencjał w implementacjach znacznie większych sieci i realizowaniu bardziej złożonych zadań. W przemysłowych zastosowaniach posiadanie oddzielnego modułu, odpowiedzialnego za detekcję zużycia narzędzia, który np. informowałby kontroler o wyniku klasyfikacji, jest dużym atutem ze względu na łatwe modyfikacje i utrzymanie systemu.

Porównując otrzymane wyniki do literatury przedmiotu zauważalne są korzyści z zastosowania prezentowanej implementacji. W przypadku pracy [75], porównanie z programem referencyjnym napisanym w Pythonie pokazało, że implementacja była nawet do 251 razy szybsza, dla implementacji w sprzęcie 4 komórek LSTM, przy zastosowaniu funkcji aktywacji aproksymowanej wielomianami, opierającej się na arytmetyce stałoprzecinkowej. W pracy wykonano również inne implementacje osiągając gorszy współczynnik przyspieszenia, ale wciąż zauważalny. Dla implementacji 4 komórek zużyto 0.18% LUTRAM oraz 3.39% FF, zaś dla 32 17.66% LUTRAM oraz 26.45% FF. Implementacje te były wykonane na układzie XC7Z020, ale w pracy przedstawiono również większe sieci implementowane na płycie ewaluacyjnej Virtex-7 VC707 zajmując 24.22% LUTRAM, 9.14% FF oraz 24.22% LUT dla 64 komórek, jednak nie podano wyników dla tych implementacji. Autorzy nie podali wyników analizy jakościowej klasyfikacji, ani informacji, czy zmiana platformy obliczeniowej wpłynęła na jakości klasyfikacji.

W pracy [101] nie przedstawiono dokładnie implementowanej sieci, natomiast wspomniano o przyspieszeniu obliczeń na Virtex7-VX485 o 20.18 razy w stosunku do obliczeń referencyjnych wykonywanych na jednym wątku CPU (Intel Xeon CPU E5-2430), napisanych w C/C++, przy zajmowaniu 198 tys. tablic LUT, 182 tys. przerzutników oraz 1176 DSP na układzie Xilinx Virtex7-VX485.

W artykule [102] zaprezentowano implementację sieci, której model składa się z dwóch warstw LSTM po 128 komórek każda, natomiast w strukturze FPGA zaimplemento-

wana była jedna komórka oraz logika odpowiednio kierująca obliczeniami. Taki sposób projektowania struktury logicznej pozwolił na przyspieszenie obliczeń 21 razy w porównaniu z procesorem ARM. Autorzy nie odnotowali, czy zmiana platformy obliczeniowej i podjęte wybory projektowe wpłynęły na jakość klasyfikacji.

W [103] wykorzystano funkcje aktywacji w postaci tablic wartości o 4096 elementach oraz arytmetykę stałoprzecinkową Q5.6 osiągając przyspieszenie o 16.9 razy szybsze w stosunku do CPU i 9.6 razy w stosunku do GPU, przy zaimplementowanej jednej komórce wraz z logiką nadzorującą obliczenia. Modelowana sieć była częścią systemu współpracującego z Caffe (platformą uczenia głębokiego opracowaną na Uniwersytecie Kalifornijskim w Berkeley, napisana w C++ i posiadającą interfejs Pythona), w której sieć dysponowała warstwą LSTM z 512 komórkami. W pracy nie podano jednoznacznego raportu dotyczącego zużycia zasobów oraz analizy wpływu zmiany platformy obliczeniowej.

Prezentowane w tym punkcie podejście do organizacji obliczeń wewnątrz komórki różni się od podejść spotykanych w literaturze. W niniejszej pracy, w komórce zaimplementowano 2 moduły funkcji sigmoidalnej oraz 1 moduł tangensa hiperbolicznego, co pozwoliło na wykonanie obliczeń wykorzystujących dwukrotnie 2 z modułów. W prezentowanej w tym punkcie literaturze spotykane są podejścia z wykorzystaniem 2 (1 moduł funkcji sigmoidalnej i 1 tangensa hiperbolicznego) lub 4 modułów (3 moduły funkcji sigmoidalnej i 1 tangensa hiperbolicznego). Przy założeniu stałej liczby cykli zegara dla każdego z modułów oraz stałej ilości wykorzystywanych zasobów oraz porównaniu jedynie podejść implementacyjnych, można zauważyć, że rozwiązanie prezentowane w niniejszej pracy jest bardziej oszczędne w sensie zasobów, niż w przypadku wykorzystania 4 modułów, będąc przy tym tak samo szybkim. Przy porównaniu z podejściem wykorzystującym 2 moduły, podejście wykorzystywane w niniejszej pracy jest dużo szybsze, ale wykorzystujące większą liczbę zasobów.

W niniejszej pracy, przy zachowaniu dokładności klasyfikacji, w odniesieniu do obliczeń wykonywanych w Pythonie implementacja była nawet do 2680 razy szybsza, co jest zauważalnie lepszym wynikiem w stosunku do literatury przedmiotu. W kontekście zużywanych zasobów prezentowana implementacja jest porównywalna do zużycia spotykanego w literaturze (np. [101]), a prezentowane podejście daje się łatwo zaadaptować do dowolnej sieci składającej się z warstwy LSTM. Nie ma informacji w literaturze przedmiotu na temat analizy wpływu zmiany platformy obliczeniowej, doboru parame-

trów, czy sposobu implementacji funkcji aktywacji, na jakość pracy sieci np. w zadaniu klasyfikacji. Często prezentowane opisy implementacyjne nie zawierają szczegółów pozwalających na odwzorowanie eksperymentu, co utrudnia analizę.

3.6. Implementacja klasyfikatora na platformie ZYNQ

W niniejszym punkcie przedstawiono implementację klasyfikatora do zadania detekcji zużycia narzędzia w procesie kucia na zimno z wykorzystaniem platformy ZYNQ. Ze względu na konieczność dopasowania klasyfikatora do wykorzystywanego układu, a zatem uwzględniając ograniczoną liczbę zasobów sprzętowych, prezentowana w tym podrozdziale implementacja różni się znacząco od prezentowanej w punkcie 3.5.

3.6.1. Powiązane prace dotyczące implementacji sieci LSTM na układach ZYNQ

Zainteresowanie układami ZYNQ jest zauważalne w nauce, chociaż liczba publikacji z uwzględnieniem opisu szczegółów technicznych jest mała. Zainteresowanie badaczy dotyczy głównie transmisji i analizy obrazu. W [108] autorzy zbudowali układ na bazie platformy ZYNQ z układem XC7Z020-1CLG400 do przesyłu obrazu. Obraz był przesyłany przez interfejs Gigabitowego Ethernetu do urządzenia, wyświetlany oraz odsyłany z powrotem. Sprawdzano przede wszystkim jakość przetwarzania oraz opóźnienie transmisji. Część z procesorem była odpowiedzialna za otrzymanie obrazu oraz zapisanie go w pamięci, natomiast programowalna logika odpowiadała za odczyt z pamięci poprzez DMA, kompresję oraz wyświetlenie obrazu na monitorze podłączonym przez interfejs HDMI. Obraz po kompresji był w tym samym czasie przesyłany z powrotem za pośrednictwem interfejsu Gigabitowego Ethernetu przy użyciu części PS. Podczas projektowania logiki starano się wykorzystać zalety FPGA tj. możliwości wykonywania obliczeń równolegle. Według autorów zbudowany przez nich system cechował się dużą przepustowością danych oraz dużą skalowalnością. W pracy nie ma danych dotyczących innych podejść do problemu np. w pełni programowym lub w pełni opartym na FPGA.

Przykładem zastosowania układów typu ZYNQ w komunikacji jest [109], gdzie autorzy podjęli pracę związaną z implementacją dekodera G.723.1, który można spo-

tkać przy transmisji głosu przez internet (VOIP). Implementacja została przygotowana w języku C, a następnie korzystając z narzędzi generujących opis sprzętu dostępnych jako Verilog HLS, został wygenerowany kod Verilog. W tekście nie zawarto próby określenia jakości wygenerowanej struktury logicznej i tego, czy rozwiązania tworzone bezpośrednio wypadają gorzej, czy może też lepiej w porównaniu z generowanymi z języka C.

Artykuł [110] to również próba wykorzystania układu typu ZYNQ do zadań związanych z komunikacją, a dokładniej (za autorem) z komunikacją satelitarną. Praca obejmowała wykonanie dekodera realizującego algorytm Viterbiego wraz z generatorem losowego wzoru binarnego, enkoderem konwolucyjnym, modulatorem QPSK demodulatorem QPSK oraz kwantyzatorem na układzie typu ZYNQ. Obliczenia były rozdzielone pomiędzy PS i PL. PL służył tu do realizacji zadań dekodera, podczas gdy w PS zaimplementowano pozostałe elementy systemu. Według autora efekt działania układu był zadowalający, nie podano natomiast porównania implementacji oraz uzasadnienia wyboru układu ZYNQ w tym zastosowaniu.

Praca [111] również dotyczy zagadnień telekomunikacyjnych, a dokładniej estymacji kanału dla radia NR pracującego w technologii 5G, z wykorzystaniem głębokiego uczenia maszynowego. Próbki do uczenia zostały przygotowane za pośrednictwem platformy SDR, która posłużyła do wyznaczenia nieznanych wartości odpowiedzi kanału przy wykorzystaniu znanych wartości lokalizacji pilota. Wydajność działania została porównana z innymi wykorzystywanymi technikami. Wynik porównania pokazał, że użycie metod konwolucyjnych w połączeniu z platformą ZYNQ jest tak samo dobre jak pozostałych, powszechnie używanych metod, natomiast daje dodatkowe możliwości wykorzystania modelu uczenia maszynowego do elastycznego dostosowania w zależności od obserwowanej natury kanałów. Autorzy postulują, że późniejsze prace powinny skupić się na minimalizacji zużycia zasobów części PL, co jest przesłanką, że stworzenie wydajnej struktury sprzętowej zużywającej możliwie mało zasobów nie jest zadaniem trywialnym, a wręcz wymagającym.

Artykuł [112] jest kolejną pracą wykorzystującą układ typu ZYNQ do zagadnień telekomunikacyjnych, a dokładniej do szerokopasmowej analizy widma (WSS) w oparciu o uczenie głębokie (konwolucyjne sieci neuronowe). WSS zyskał na popularności w ramach rozwoju technologii 5G i reguł podziału spektrum częstotliwościowego. Podobnie jak w innych pracach, PL posłużyła do implementacji sieci CNN z funk-

cjami aktywacji ReLU oraz sigmoidalną funkcją na wyjściu sieci. Autorzy nie podają szczegółów implementacyjnych, w tym dotyczących poszczególnych funkcji. Poszczególne warstwy sieci wynik obliczeń zapisują bezpośrednio w pamięci za pośrednictwem DMA. Wstępne przetwarzanie danych również ma miejsce w PL, gdzie wykonywane jest szereg obliczeń arytmetycznych na macierzach, a wyniki również zapisywane są bezpośrednio do pamięci przy wykorzystaniu DMA. W wyniku implementacji zużyto duże ilości zasobów dostępnych w PL dochodząc do dostępnego limitu, co pokazuje, że złożone algorytmy uczenia maszynowego wymagają dużych zasobów przy implementacji w strukturach FPGA.

Autorzy [113] podjęli tematykę wykorzystania układów ZYNQ do zadań nawigacyjnych. Urządzenie było w założeniu przeznaczone jedynie do małych samolotów i integrowało w sobie funkcje sterowania, odczytu z czujników, akwizycji obrazu oraz sterownika serwa. Część PL była odpowiedzialna głównie za akwizycję obrazu, komunikację z czujnikami oraz wysterowanie serwa za pomocą PWM, do części PS były podłączone wszystkie elementy, które nie musiałyby pracować w czasie rzeczywistym, takie jak interfejsy zewnętrzne, ale i odczyty z IMU. Autorzy twierdzą, że wykorzystanie układów ZYNQ pozwoliło na kompaktową budowę, małe zapotrzebowanie na energię oraz stabilne działanie.

W [114] autorzy twierdzą, że systemy oparte o FPGA, a w szczególności oparte o układy typu ZYNQ, pomimo bardziej złożonego i przez to dłuższego procesu projektowania wydają się bardziej odpowiednie dla niektórych problemów np. implementacja filtru Kalmana w wersji UKF, gdzie zwiększenie liczby monitorowanych parametrów znacząco zwiększa zapotrzebowanie na moc obliczeniową. Obliczenia związane z filtrem Kalmana powinny być wykonywane możliwie szybko, ponieważ niosą ze sobą informację dotyczącą stanu obiektu, na podstawie której określa się sterowanie. Autorzy podkreślają, że w przypadku samolotów bezzałogowych, systemy ZYNQ pozwalają na połączenie zalet procesora oraz logiki programowalnej. W omawianym systemie wykorzystywano architekturę zmiennoprzecinkową pojedynczej precyzji.

Praca [115] przedstawia próbę wykorzystania układów typu ZYNQ do przyspieszenia obliczeń wielomianowego mnożenia macierzy, w której część PL służyła jako zewnętrzny moduł obliczeniowy specjalnie zaprojektowany w tym celu. W części PS zainstalowany został PetaLinux. W rezultacie eksperymentów zauważono, że wykorzystanie zewnętrznego modułu pozwoliło na przyspieszenie obliczeń nawet do 67 razy przy

architekturze mieszanej: zmiennoprzecinkowej pojedynczej precyzji dla większości elementów, zaś stałoprzecinkowej wewnątrz jednego bloku mnożącego. Artykuł pokazuje, że układy typu ZYNQ mogą być dobrym wyborem w przypadku, gdy projekt obejmuje niestandardowe obliczenia, albo realizuje je inaczej, wtedy zastosowanie PL jako przeznaczonego do konkretnego zadania modułu wewnętrznego wydaje się naturalnym wyborem.

W pracy [116] podjęto próby implementacji na platformie ZYNQ-7000 zewnętrznego modułu testowego silnika asynchronicznego z obciążeniem. Założeniem projektowym było, żeby układ symulował działanie silnika i był używany w podejściu MIL (Model In the Loop) oraz HIL (Hardware In the Loop) podczas weryfikacji działania poszczególnych elementów systemu współpracującego z silnikiem asynchronicznym. Struktura PL została wygenerowana za pomocą Vivado HLS i obejmowała model silnika. Model masy obciążającej silnik został zakodowany w PS. Zbudowany model spełniał wymagania podczas porównania z analogicznym, stworzonym w Simulinku i pozwolił na wykonywanie testów w czasie rzeczywistym. W pracy nie ma informacji dotyczącej porównania struktury wygenerowanej narzędziami HLS ani danych z porównania wydajności i czasu działania z innymi podejściami np. w pełni programowym lub w pełni zrealizowanym na FPGA. Przedstawiona implementacja wykorzystuje architekturę stałoprzecinkową z 26 bitami, w tym 24 bitami części ułamkowej w przypadku PL.

W [117] wykorzystano układ typu ZYNQ do zadania rozpoznawania sygnałów EMG (elektromiografii) w protezie kończyny i wykonywania na ich podstawie odpowiednich ruchów protezą. Na części PL zostały zaimplementowane dwa moduły: pierwszy wydobywający informacje z sygnału EMG, którego źródłem była opaska zamontowana na przedramieniu, a drugi wykonujący klasyfikację w oparciu o metodę k najbliższych sąsiadów (kNN). W efekcie zastosowania platformy ZYNQ i przeniesienia części obliczeń na część PL uzyskano przyspieszenie o 5 razy dla etapu wydobywania danych oraz 2.15 raza dla predykcji. Dokładność rozpoznania gestów plasowała się na poziomie 96.4%. Podobnie jak w poprzednich pracach autorzy zdecydowali się na generowanie opisu sprzętu z kodu (tutaj C++) zamiast bezpośredniej implementacji. Warto zauważyć, że autorzy poczynili szereg optymalizacji wygenerowanej struktury dla modułu ekstrakcji cech, co pozwoliło zmniejszyć zapotrzebowanie na przerzutniki o 16%, a na tablice LUT o 15%, natomiast zwiększyło czas obliczeń. Dla

modułu predykcji starano się zoptymalizować czas obliczeń, co pozwoliło go zmniejszyć prawie 7 razy, natomiast odbyło się to kosztem znacznego zapotrzebowania na zasoby tj. 4.47 razy zwiększone zapotrzebowania na przerzutniki oraz 3.19 razy zwiększone zapotrzebowanie na tablice LUT. Autorzy nie wykonali modułów samodzielnie od podstaw. Perspektywa ta mogłaby być wartościowym porównaniem narzędzi syntezy. Nie dokonano porównania z innymi podejściami np. w pełni programowymi. Nie podano uzasadnienia wykorzystania platformy ZYNQ do rozwiązania opisywanego problemu.

Praca [118] prezentuje wykorzystanie układu typu ZYNQ do zadania klasyfikacji gazów w urządzeniach alarmujących. Klasyfikator oparty był o sieć neuronową w postaci perceptronu wielowarstwowego. Dane uczące zostały zgromadzone za pomocą czujnika z tlenkiem cyny. Implementacja klasyfikatora została wykonana na części PL z nakierowaniem na minimalizację czasu obliczeń. Sieć neuronowa składała się z 12 wejść, 3 neuronów ukrytych i 1 wyjścia. Wszystkie neurony wyposażone były w funkcję aktywacji w postaci tangensa hiperbolicznego, który był implementowany jako tablica punktów. Osiągnięto dokładność klasyfikacji na poziomie 97.4%, przy implementacji stałoprzecinkowej (16 bitów przy 14-bitowej części ułamkowej).

W artykule [13] przedstawiono implementację sieci konwolucyjnej wykorzystującej układ ZYNQ. Sieć składała się z 2 warstw konwolucyjnych, 2 warstw łączących oraz 1 gęsto połączonej i była wytrenowana dla zadania rozpoznawania ręcznie pisanych liczb oraz dla jądra o rozmiarach 3x3. Część PS była odpowiedzialna za sterowanie logiką oraz przechowywała dane testowe. Cała sieć została zaimplementowana na PL. Pomimo implementacji stosunkowo prostej sieci, zużycie zasobów sprzętowych było znaczne: 85.5% DSP, 47.8% tablic LUT i 45.8% BRAM, a implementacja osiągnęła porównywalny czas pracy do obliczeń na GPU. Praca nie przedstawia szczegółów implementacyjnych i sposobu organizacji logiki w PL.

Jedną z niewielu prac poruszających temat implementacji sieci rekurencyjnych na układach ZYNQ jest [102], w której autorzy podjęli próbę implementacji sieci LSTM z 2 warstwami i 128 komórkami do rozpoznawania liter. Implementacja opierała się o arytmetykę stałoprzecinkową Q8.8. Funkcje aktywacji zostały zaimplementowane jako aproksymacja liniowo-odcinkowa 13 odcinkami, a w strukturze sprzętowej zaimplementowana została pojedyncza komórka LSTM. Nie ma informacji o sposobie konfiguracji części PL. Nie podano szczegółów dotyczących DMA oraz nie zdefiniowano sposobu komunikacji z DMA. W wykorzystywanym w niniejszej pracy oprogramowaniu

nie udało się odwzorować przedstawianej w [102] architektury, ani konfiguracji DMA. Autorzy [102] osiągnęli stosunkowo duże wartości błędów na wyjściu sieci tj. maksymalnie 7.1%, co nie jest problemem w zadaniu generowania tekstu, natomiast przy problemie klasyfikacji ma duże znaczenie i mogłoby być dyskwalifikujące.

Artykuł [119] prezentuje trzy podejścia do implementacji sieci rekurencyjnych na układach FPGA, wykorzystujące przy tym układy typu ZYNQ. Implementacje były testowane na modelu języka na poziomie znaków, z użyciem arytmetyki stałoprzecinkowej Q8.8. Autorzy nie przedstawiają informacji o sposobie realizacji funkcji aktywacji, a przedstawiane podejścia dotyczą sposobu pobierania i przetwarzania danych, implementacji pojedynczej komórki oraz organizacji obliczeń w sposób taki, żeby wykonać obliczenia dla całej sieci przy modelowaniu zaledwie jednej komórki LSTM. Zużycie zasobów pomiędzy podejściami różni się znacząco i maksymalnie wynosi 73% dostępnych zasobów. Autorzy osiągają też stosunkowo duże błędy dla parametrów wyjściowych, tj. średnio 3.9% dla C_t oraz 2.8% dla h_t. Rozwiązanie nie zostało porównane z innymi podejściami np. z implementacją w pełni programową. W artykule nie ma też dokładnej analizy wyników, wykazano jedynie wydajność na poziomie $1W$ w zestawieniu z kilkoma innymi implementacjami na FPGA.

Z przedstawionej analizy literatury wynika, że układy ZYNQ w wielu przypadkach nie były porównywane z innymi podejściami implementacyjnymi i nie wykazano stosowności wykorzystania takiego podejścia w odniesieniu np. do rozwiązań w pełni opartych na układach FPGA lub realizowanych wyłącznie na procesorze. Z pewnością na uwagę zasługuje uniwersalność układu, który pozwala m.in. na wykorzystanie systemu operacyjnego i delegowanie części obliczeń do części PL. W przypadku niewielkich układów dużym problemem jest mała dostępność zasobów FPGA, szczególnie przy implementacji obliczeń związanych z uczeniem głębokim. W literaturze niewiele jest prób implementacji sieci rekurencyjnych z wykorzystaniem układów ZYNQ, częściej spotkać można jedynie implementacje CNN, kNN i MLP. Cytowane prace, dotyczące sieci LSTM na ZYNQ, wciąż pozostawiają wiele kwestii do rozważenia, szczególnie w kontekście porównania z innymi podejściami implementacyjnymi. W zaprezentowanych artykułach brak szczegółów implementacyjnych, co może mieć związek z generowaniem struktury sprzętowej przez wybrane narzędzia. Sprawia to, że trudno jest ocenić adekwatność i jakość prezentowanego podejścia. Dodatkowym aspektem wartym dalszego zbadania jest porównanie implementacji wygenerowanych i napisanych ręcznie,

ze szczegółowym uwzględnieniem wskazań do użycia jednego lub drugiego w zależności od wymagań projektowych.

3.6.2. Implementacja klasyfikatora

W celu przeprowadzenia implementacji i eksperymentów, wykorzystana została płyta ewaluacyjna Digilent Zybo Z7-20, wyposażona w układ XZ7020 posiadający 53.2 tys. tablic LUT, 106.4 tys. przerzutników oraz 220 DSP. Głównym kryterium wyboru była dostępność płyty ewaluacyjnej oraz licencji na narzędzia. Wykorzystany układ był zbyt mały by pomieścić całą sieć klasyfikatora, jaka została przedstawiona w punkcie 3.5, dlatego konieczna była zmiana podejścia oraz gruntowne przeprojektowanie logiki. Przyjęte w tym punkcie podejście polegało na zbudowaniu modułu będącego elementem peryferyjnym procesora, do którego oddelegowano obliczenia związane z siecią LSTM. Moduł miał odpowiadać funkcjonalnie pojedynczej komórce LSTM lub komórce wyjściowej. Ze względu na występujące podobieństwa tj. element sumatora oraz funkcję aktywacji, możliwe było zamknięcie obu funkcjonalności w jednym module opisanym w języku Verilog. Korzystanie z modułu odbywało się za pośrednictwem interfejsu DMA, zaś wybór rodzaju wykonywanych obliczeń (obliczenia dla komórki LSTM, czy obliczenia dla neuronu wyjściowego) wykonywany był port w taki sposób, żeby wykorzystać minimalną liczbę zasobów. Dane wejściowe najpierw umieszczono w buforze przez procesor, a następnie uruchomiono transmisję, w trakcie której część PL odbierało dane, wykonywało obliczenia oraz zwracało wynik obliczeń. Wynik zapisano w drugim buforze, z którego dane mogła pobrać PS. Po zakończeniu transmisji wystawiony został sygnał informujący o przerwaniu. Odpowiedzialność za poprawne przygotowanie bufora spoczywało na PS.

Rozważono cztery podejścia implementacyjne. Pierwsze podejście zakładało minimalne wykorzystanie zasobów sprzętowych. W tym podejściu, w buforze umieszczono wartości przesunięć oraz poszczególnych sygnałów, wraz z odpowiednimi wagami połączeń. Jak widać na rysunku 3.35, na wejście każdej z komórek LSTM podawany był sygnał wejściowy oraz sygnały rekurencyjne. Każda z bramek posiada osobny zestaw wag, ze względu na to, że komórka LSTM posiadała 4 bramki (o czym więcej w punkcie 3.1) liczba wag potrzebnych do przesłania była czterokrotnością liczby komórek w warstwie. W przypadku neuronu wyjściowego, w buforze osadzono wartości sygnałów z wyjść komórek oraz odpowiadające im wagi. Ze względu na to, że protokół

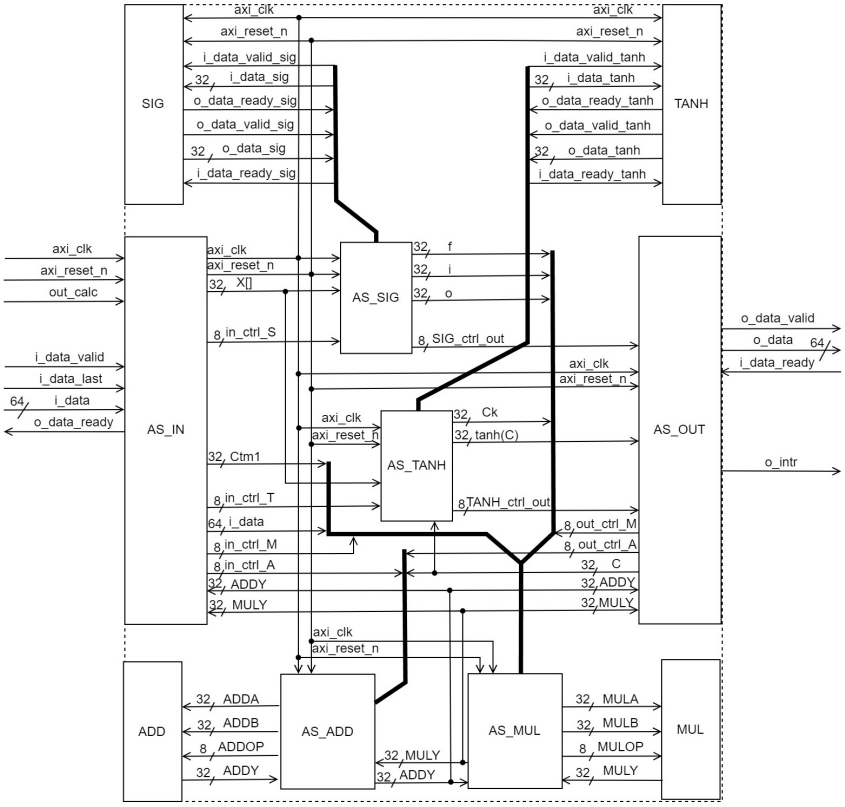
AXI4-Stream oraz DMA wspierają szerokość pola danych nawet do 1024 bitów, zdecydowano się przesyłać wartości sygnałów wraz z odpowiadającymi im wagami. Sygnały takie jak wartość stanu komórki z poprzedniego cyklu obliczeniowego, czy przesunięcia, przesyłano z polem odpowiadającym wadze, ustawionym na wartość 1. Podobnie jak w poprzednich przypadkach, zdecydowano się na zastosowanie arytmetyki zmiennoprzecinkowej pojedynczej precyzji, zgodnej z IEEE754. Przyczyną ku temu było pominięcie etapu konwersji, który przy takiej ilości danych byłby czasochłonny dla procesora, oraz uwydatnienie wpływu zmiany platformy obliczeniowej. Pojedyncze pole danych zajmowało zatem 64 bity. Pozwoliło to na wykonywanie po stronie PL mnożenia przy każdej zmianie danych na wejściu i dalsze sumowanie w trakcie odbierania danych. Dane przesyłane były w następujący sposób:

$$
\begin{array}{lllllll}
[Xin, wi], & [H00, w00i], & [H01, w01i], & ..., & [H43, w43i], & [bi, \text{"1"}], & [Ctm1, \text{"1"}], \\
[Xin, wf], & [H00, w00f], & [H01, w01f], & ..., & [H43, w43f], & [bf, \text{"1"}], & \\
[Xin, wc], & [H00, w00c], & [H01, w01c], & ..., & [H43, w43c], & [bc, \text{"1"}], & \\
[Xin, wo], & [H00, w00o], & [H01, w01o], & ..., & [H43, w43o], & [bo, \text{"1"}]. &
\end{array}
$$

gdzie: Xin to wartość sygnału wejściowego dla rozpatrywanego kroku; wi, wf, wc i wo to wagi do poszczególnych bramek od poprzedzającej warstwy; $H00...H43$ wartości wyjściowe każdej z komórek; $w00...w43$ wagi połączeń rekurencyjnych; bi, bf, bc, bo przesunięcia dla poszczególnych bramek; a $Ctm1$ jest wartością stanu wewnętrznego z poprzedniego kroku obliczeniowego.

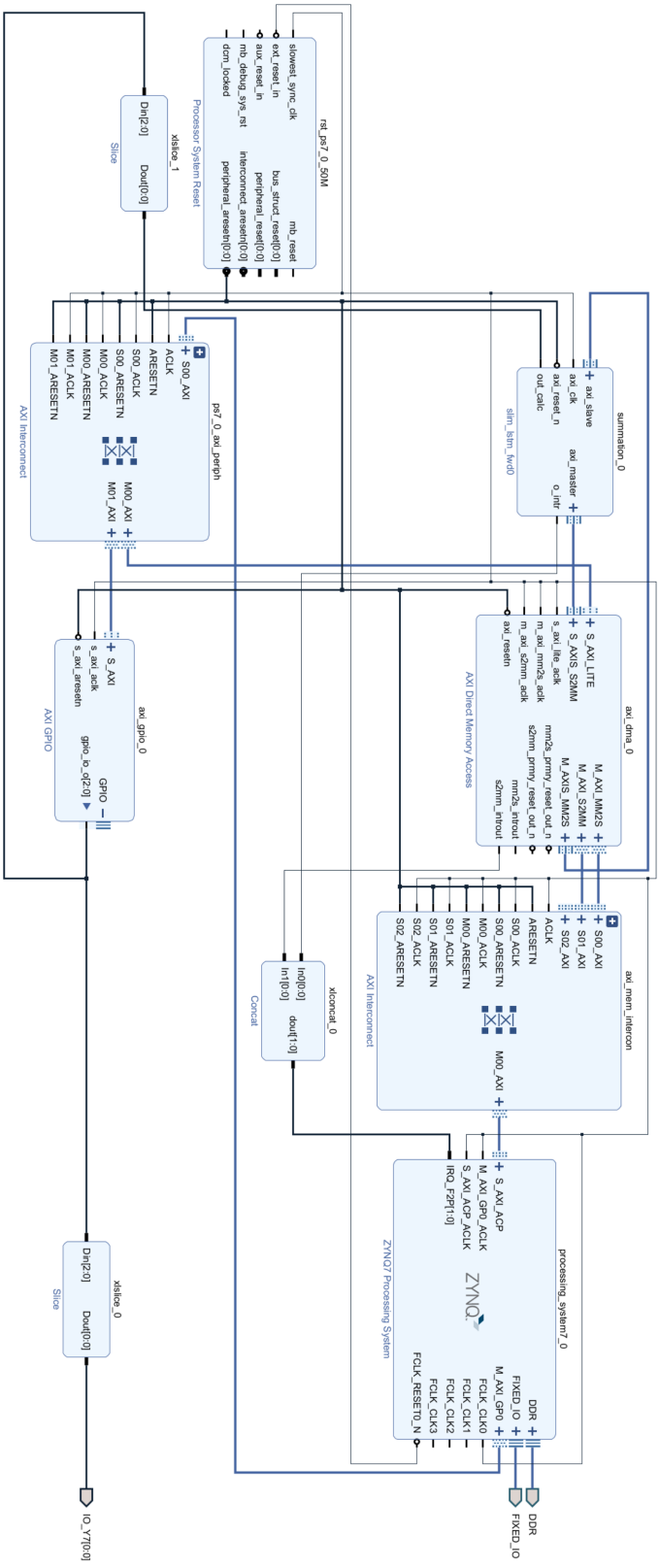
Dało to łączenie 185 elementów do przetransmitowania. W przypadku neuronu wyjściowego było to 45 elementów. Transmisji związanych z komórkami LSTM było 15 (długość sekwencji, punkt 3.5) * 44 (liczba komórek w warstwie) = 660. Ponieważ dane zorganizowano w konkretnych grupach odpowiadającym bramkom komórki i wysyłano są za koleją, możliwe było m.in. rozpoczęcie obliczeń wartości na wyjściu bramki wejściowej tuż po przetransmitowaniu elementu $[bi, \text{"1"}]$ i analogicznie w przypadku pozostałych bramek. Sumy iloczynów par przetransmitowanych danych zostały podane na wejście jednej z funkcji aktywacji, zaimplementowanych zgodnie z opisanym w punkcie 3.3.5, wariantem 3. Schemat logiczny implementacji przedstawiony został na rysunku 3.39. W komórce LSTM zaimplementowany został jeden moduł sigmoidalnej funkcji aktywacji oraz jeden moduł tangensa hiperbolicznego, które dysponowały własnymi układami sumującymi oraz mnożącymi, przez co nie wpływały na proces przetwarzania danych wejściowych. Mając wartości sygnałów f_t, i_t oraz $\hat{C}_t$

przystąpiono do obliczenia nowej wartości stanu komórki C_t zgodnie z równaniem (3.5). Jedynie obliczenie wartości sygnału o_t oraz wyjścia tj. h_t zgodnie z równaniem (3.6) następowało po pełnym przetworzeniu danych wejściowych, co sprawiło, że cały proces obliczeniowy trwał niewiele dłużej niż przetwarzanie całego strumienia danych. Wartości obliczone w komórce zwracane były w pojedynczym polu danych jako złączone ze sobą $[ht, Ct]$. Opisywany moduł został przedstawiony na rysunku 3.40 jako blok *slim_lstm_fwd0*. Podczas projektowania struktury logicznej PL w Vivado firmy Xilinx skorzystano z edytora graficznego, w którym poszczególne elementy prezentowane są w formie bloków, a następnie ze sobą łączone.



Rysunek 3.39: Struktura logiczna sieci LSTM zaimplementowanej na układzie ZYNQ z parametrami po stronie PS

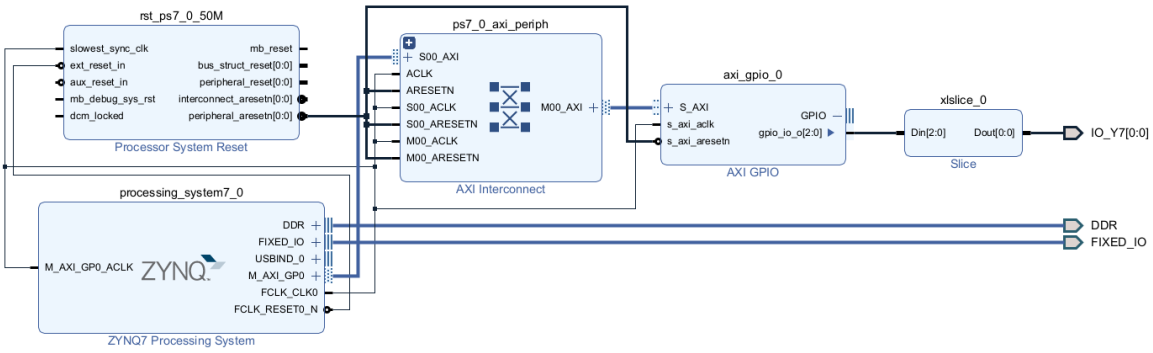
Przypomina to projektowanie układów elektronicznych lub język programowania sterowników przemysłowych FBD. Poszczególne bloki to albo przygotowane w trakcie projektowania bloki, albo wcześniej przygotowane bloki dostarczone razem z oprogramowaniem. Głównym blokiem na rysunku 3.40 jest *ZYNQ7 Processing System*, który odpowiada za konfigurację wewnętrzną układu ZYNQ. W nim definiuje się m.in.



Rysunek 3.40: Schemat wykorzystanej konfiguracji połączeń pomiędzy PL i PS w układzie typu ZYNQ

częstotliwości taktowania (może być kilka), to z jakich interfejsów będzie się korzystać, czy też definiuje się przerwania. Blok jest też źródłem sygnału reset ponieważ o konfiguracji PL decyduje PS. Sygnał reset jest doprowadzony do bloku *Processor System Reset*, a następnie dystrybuowany po innych blokach. Zdefiniowane interfejsy pojawiają się jako wyprowadzenia bloku, nieaktywne pozostają niewidoczne, co sprawia, że konfiguracja jest bardziej przejrzysta.

Do zdefiniowania rodzaju obliczeń w bloku $slim\_lstm\_fwd0$ wykorzystano interfejs $AXI\ GPIO$, który był wykorzystywany m.in. do wysterowywania wyprowadzeń na obudowie układu. Jedno z połączeń zamiast być podłączone do wyprowadzenia na obudowie zostało przekierowane na wejście out_calc bloku $slim\_lstm\_fwd0$, w wyniku czego wykorzystując 1 bit interfejsu można było odpowiednio konfigurować obliczenia. Wyłuskanie pojedynczego bitu odbywało się za pośrednictwem bloku *Slice*. W projektowanej strukturze wykorzystany został też drugi bit z $AXI\ GPIO$, który to został wykorzystany do wysterowywania wyprowadzenia na obudowie, a dokładniej wyprowadzenia $Y7$. Wyprowadzenie posłużyło do pomiaru czasu obliczeń za pośrednictwem oscyloskopu.



Rysunek 3.41: Schemat konfiguracji wykorzystanej do pomiaru czasu obliczeń wykonywanych jedynie w PS

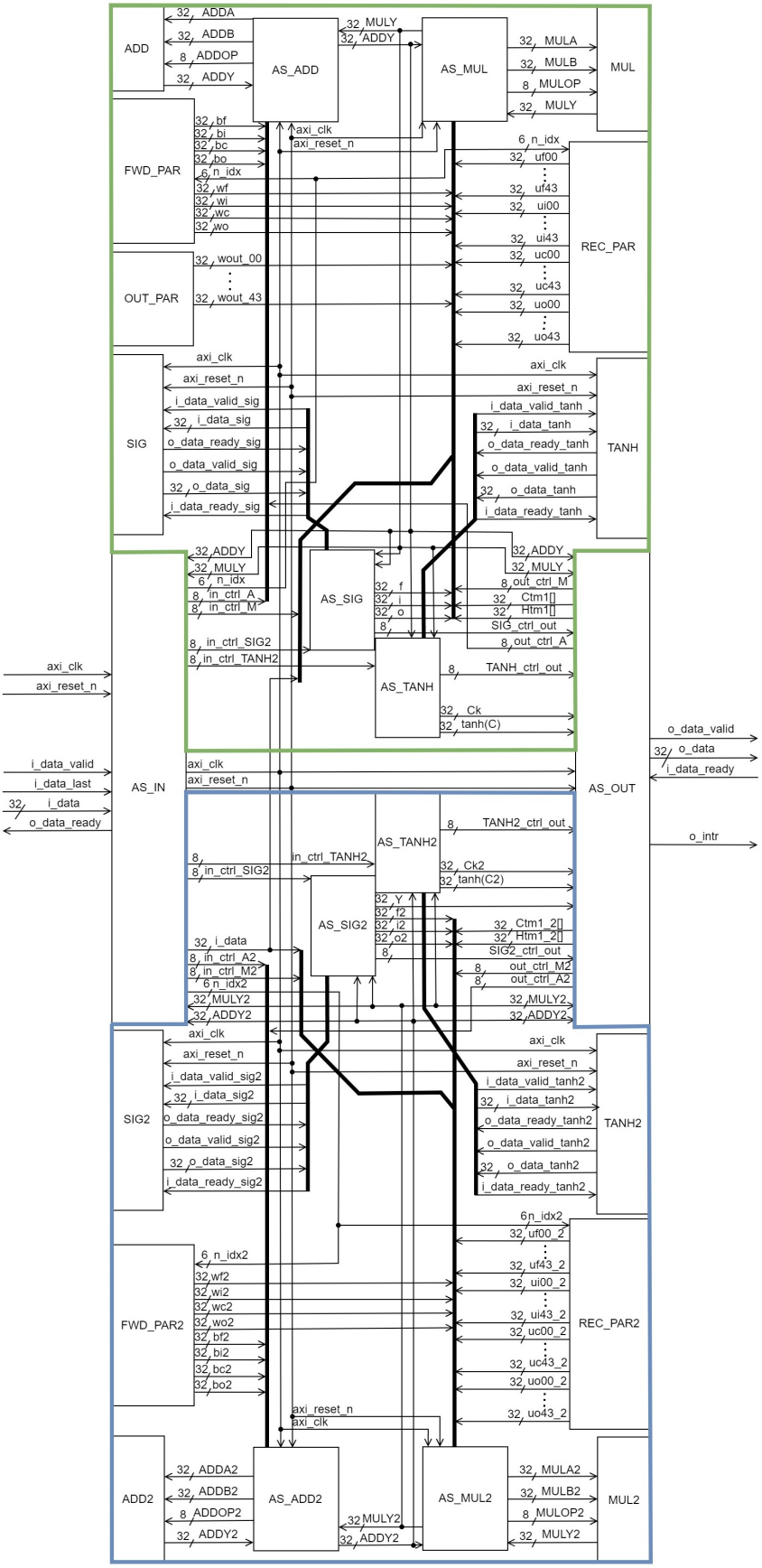
Posłużenie się oscyloskopem do pomiaru czasu obliczeń było podyktowane potrzebą ujednolicenia sposobu pomiaru czasu obliczeń przy wykorzystaniu PL, jak i podczas obliczeń referencyjnych, będących jednocześnie drugim podejściem implementacyjnym. W części PL skonfigurowane było jedynie wyprowadzenie do pomiaru czasu. Struktura wykorzystywana podczas obliczeń referencyjnych jest przedstawiona na rysunku 3.41. Tym sposobem jedyną zmienną podczas porównywania pomiarów czasu

była metoda obliczeń. Jednym z pozostałych elementów na rysunku 3.40 jest blok *AXI Direct Memory Access*, który odpowiadał za transfer danych z pamięci przez DMA. To w nim definiowano parametry przesyłanych danych, takie jak szerokość pola danych, czy wielkość bufora danych. Wszystkie bloki połączono za pośrednictwem bloków typu *AXI Interconnect*, na schemacie występują dwa. Jeden w całości przeznaczony do odbioru danych w transmisji opartej o DMA i dalej podłączony do portu slave głównego bloku, drugi zaś podłączony do wyprowadzenia master głównego bloku i odpowiadający zarówno za odbiór danych przeznaczonych dalej do $slim\_lstm\_fwd0$ jak i za wysterowywanie portu *AXI GPIO*.

Współpracujące z opisanym systemem oprogramowanie wykorzystywało tylko jeden rdzeń z dwóch dostępnych i nie wykorzystywało żadnego systemu operacyjnego. Kod został napisany w języku C i był wgrywany bezpośrednio na procesor. Pomimo, że w głównym bloku *ZYNQ7 Processing System* sprecyzowane zostało, jakie interfejsy będą wykorzystywane oraz to, że będą wykorzystywane przerwania, osobno określono konfigurację po stronie PS i odpowiednio zainicjowano poszczególne interfejsy. Pisząc oprogramowanie w programie Vitis firmy Xilinx (co miało miejsce w tym przypadku), który jest oparty o popularne środowisko Eclipse (obecnie rekomendowanym programem do pisania oprogramowania na platformę ZYNQ), istnieje możliwość wybrania szablonu projektowego, w którym podczas konfiguracji określa się ścieżki do przygotowanych przez firmę Xilinx bibliotek i tym samym umożliwia korzystanie z funkcji i makr znacząco ułatwiających proces programowania. Podczas wspomnianej konfiguracji dołączono wcześniej wyeksportowany plik z opisem konfiguracji PL. Umożliwiło to późniejsze wgranie konfiguracji z poziomu środowiska Vitis. Inicjalizacja wykorzystywanych interfejsów miała miejsce na początku i obejmowała interfejs GPIO, kontroler DMA, kontroler przerwań oraz sprecyzowanie, jakie przerwania z jakim priorytetem będą wykorzystywane. W opisywanym systemie wykorzystywane było jedynie przerwanie zakończenia transakcji przez DMA, które oznacza, że dane zostały wysłane, blok wykonujący obliczenia dane otrzymał, przetworzył, wysłał z powrotem, a dane z wynikiem zostały poprawnie odebrane. Chcąc korzystać z tego przerwania, należy go również włączyć w kontrolerze DMA, a podczas konfiguracji kontrolera przerwań przypisuje się funkcję, jaka ma zostać wykonana w przypadku pojawienia się przerwania. Podczas inicjalizacji interfejsu GPIO określa się jedynie, czy dana linia będzie wykorzystywana jako wejście, czy jako wyjście. W prezentowanym przypadku były to 2

wyjścia, co odpowiada ustawieniu wartości tych bitów w masce kierunkowej na 0. W przypadku obliczeń realizowanych w całości po stronie PS, czyli w przypadku podejścia drugiego, miała miejsce jedynie konfiguracja GPIO i ustawienie bitu wykorzystywanego do pomiaru czasu. Obliczenia realizowane były według równań (3.1) - (3.6), iterując po wszystkich neuronach w warstwie oraz po danych wejściowych. Po zakończeniu obliczeń w warstwie LSTM, nastąpiło zsumowanie przemnożonych przez odpowiadające wagi wyjść komórek z warstwy LSTM oraz dodanie przesunięcia, a następnie obliczenie wartości klasyfikacji w oparciu o funkcję sigmoidalną. Podczas obliczeń programowych wykorzystane zostały standardowe funkcje matematyczne dostępne w bibliotece *math.h*. W przypadku tangensa hiperbolicznego wykorzystano bezpośrednio funkcję *tanh*(), natomiast funkcja sigmoidalna była obliczana zgodnie z (3.7), gdzie funkcja *exp*() wzięto ze wspomnianej biblioteki. Podczas implementacji wykorzystano wagi i dane opisane w punkcie 3.5.2.

Trzecie podejście implementacyjne zakładało przeniesienie wszystkich parametrów sieci tj. wartości wag i połączeń dla wszystkich komórek oraz dla warstwy wyjściowej na stronę PL. Takie podejście wiązało się ze zużyciem większej ilości zasobów sprzętowych w związku z koniecznością przechowania danych oraz obsłużenia ich, lecz znacząco zredukowało liczbę transmitowanych danych pomiędzy PL i PS. Zmniejszyło również liczbę samych transmisji do jednej transmisji na początku obliczeń, w celu przesłania analizowanego szeregu, oraz jednej po zakończeniu obliczeń. W związku z tym, strumień danych w przypadku transmisji inicjującej składał się z 15 elementów transmitowanych kolejno, a port wejściowy *i_data* miał 32 bity. Port wejściowy był zgodny z AXI4-Stream i wraz z resztą struktury logicznej został zaprezentowany na rysunku 3.42. Transmisja była koordynowana przez DMA, dzięki czemu nie wstrzymywano pracy procesora. Przyjęto w związku z tym, że logika nie będzie magazynowała wartości sekwencji, tylko będą one kolejno wykorzystywane w obliczeniach, a informacja o podaniu kolejnej wartości będzie przekazywana zgodnie ze sposobem działania interfejsów AXI4-Stream, czyli poprzez transakcję w oparciu o sygnały *i_data_valid* oraz *o_data_ready*. Dodatkowo wykorzystany został sygnał *i_data_last* w celu oznaczenia końca sekwencji i przekazania informacji o możliwości rozpoczęcia obliczeń związanych z warstwą wyjściową zaraz po zakończeniu obliczeń związanych z wartościami z sekwencji.



Rysunek 3.42: Struktura logiczna modułu sieci LSTM zaimplementowanej na układzie ZYNQ z parametrami po stronie PL i operującej na dwóch komórkach równolegle

W ramach transakcji kończącej proces obliczeń wysyłana była, przez interfejs wyjściowy, tylko jedna wartość będąca wynikiem klasyfikacji, a dokładniej wartością wyjściową funkcji aktywacji z warstwy wyjściowej. Schemat z rysunku 3.42 przedstawia strukturę logiczną odpowiadającą implementacji sieci LSTM dla przypadku, gdy po stronie PL znajdują się dwie komórki LSTM wykorzystywane w obliczeniach oraz każda z komórek wyposażona jest w jedną parę bloków arytmetycznych. Bloki przyporządkowane poszczególnym komórkom zostały wzięte w obrys zielony (pierwsza komórka LSTM) oraz niebieski (druga komórka LSTM). W ramach eksperymentów wykonano różne konfiguracje logiki PL: dla 1, 2 lub 4 komórek LSTM w strukturze logicznej oraz dla 1, 2 lub 4 par bloków arytmetycznych przypadających na jedną komórkę LSTM. W przypadku dwóch komórek LSTM i jednej parze bloków arytmetycznych (przypadek z rys. 3.42) można zauważyć, że każda z komórek ma oddzielny zestaw bloków, z których pobierane są wartości wag i przesunięć. Są to bloki REC_PAR oraz REC_PAR2 dla wag połączeń rekurencyjnych, FWD_PAR oraz FWD_PAR2 dla wag połączeń wejściowych oraz przesunięć, oraz OUT_PAR dla wag połączeń wyjściowych, który znajduje się tylko w jednej z komórek. Ze względu na sposób organizacji obliczeń, zawartość tych bloków nie dubluje się. Jedna komórka wykonuje obliczenia jako co druga komórka w warstwie. Dopiero po zakończeniu pełnego cyklu obliczeniowego aktualizowana jest tablica wartości rekurencyjnych i stanów wewnętrznych, i rozpoczyna się kolejny cykl dla kolejnej wartości z sekwencji pobranej z wejścia. Niezależnie od rozważanej liczby komórek w PL, tylko jedna z komórek pełni dodatkową rolę warstwy wyjściowej. W przypadku większej liczby bloków arytmetycznych obliczenia są rozkładane, zgodnie z zasadą rozwijania pętli (punkt 2.2.2) na dwie lub cztery części, obliczenia wykonują się wtedy równolegle dla kilku kolejnych elementów. Rozwinięcie pętli ma miejsce głównie przy okazji sumowania iloczynów wartości połączeń rekurencyjnych oraz ich wag. Moduły, które realizowały obliczenia związane z funkcjami aktywacji, posiadały wewnątrz oddzielne bloki arytmetyczne, przez co były w stanie wykonywać działania w trakcie obliczania wartości wejściowej dla kolejnej bramki, usprawniając w ten sposób cykl obliczeniowy. W proponowanym rozwiązaniu wykorzystano implementację funkcji aktywacji z wariantu 3 (punkt 3.3).

Schemat połączeń pomiędzy blokami, wewnątrz PL, jest w tym przypadku taki sam, jak w podejściu pierwszym i może być opisany schematem z rysunku 3.40. Jedyną różnicą są nastawy bloków DMA, które zmieniono na odpowiadające znacząco

zredukowanej transmisji. Analogicznie do poprzednich przypadków, czas obliczeń był mierzony na wyprowadzeniu $Y7$. Program pracujący po stronie PS został znacząco uproszczony i zredukowany do konfiguracji PL, przygotowania sekwencji, na której miała zostać wykonana klasyfikacja, zainicjowania komunikacji z PL, do odbioru wartości wyjściowej sieci oraz odpowiedniego wysterowania sygnału pomiarowego.

Czwarte podejście implementacyjne z wykorzystaniem platformy ZYNQ zakładało użycie narzędzi do generowania struktury logicznej z kodu programu w języku C/C++. Wykorzystano w tym celu program Xilinx Vitis HLS i kod napisany w C++. Przygotowanie programu mającego być podstawą do generowania struktury logicznej różniło się znacząco od pisania programu wykonywalnego na procesorze. Funkcjonalność jaka ma być uzyskana po stronie PL ma postać funkcji, która może mieć dowolną nazwę. Funkcję główną, która ma zostać użyta jako baza do wygenerowania struktury logicznej określa się w ustawieniach projektu. Funkcja główna może korzystać z innych funkcji zdefiniowanych w projekcie w sposób taki, jak ma to miejsce w języku C++. Funkcja *main* definiowana jest podczas tworzenia narzędzi testowych dla projektowanej funkcjonalności. Funkcja, która została zdefiniowana jako główna, może zostać załączona w plikach narzędzi testowych i wywołana z poziomu funkcji *main* w trakcie testów symulacyjnych. Definiowanie struktury logicznej odbywa się przede wszystkim w dyrektywach preprocesora *#pragma*, gdzie definiowane są m.in. interfejsy lub konwersja pętli, czyli np. to czy ma być zastosowany pipelining, a jeśli tak, to w jakim stopniu. Poprzez kod definiuje się zachowanie projektowanej struktury. Narzędzia do generowania operują na wielu wstępnie określonych parametrach, które wymagają korekcji, gdy nie są w stanie wygenerować struktury logicznej spełniającej m.in. wymagania czasowe, co ma często miejsce w przypadku większych projektów. Przykładowo, zastosowanie pipeliningu jest zależne od domyślnie ustawionej strategii, co w przypadku projektu sieci wymagało korekty dla każdej pętli i wydłużenia czasu pomiędzy krokami. Definiowanie interfejsu takiego jak AXI4-Stream, który został użyty w tym przypadku, nie wymaga tworzenia go od podstaw, ponieważ środowisko dostarcza gotowe do użycia biblioteki. W opisywanym przypadku zdefiniowano trzy interfejsy, dwa AXI4-Stream: jeden do odbioru danych i drugi do wysłania wyniku obliczeń oraz jeden AXI4-Lite, który jest wymagany do kontroli nad modułem, czyli m.in. do wystartowania modułu. Podobnie jak w przypadku podejścia trzeciego, dane z analizowanej sekwencji były analizowane pojedynczo. W trakcie jednego kroku obliczenia wykony-

wane były dla całej warstwy rekurencyjnej, po czym następowała aktualizacja tablic wartości stanu wewnętrznego komórek oraz sygnałów rekurencyjnych i przystąpienie do kolejnego kroku obliczeniowego. Wartości wag i przesunięć umieszczone zostały w strukturze logicznej i mogą być bezpośrednio wykorzystane przez moduł. Obliczenia związane z funkcjami aktywacji wykorzystywały funkcje biblioteczne $hls::tanh()$ oraz $hls::expf()$ (zgodnie z równaniem (3.7)) z biblioteki $hls\_math.h$, która wykorzystuje algorytmy dostarczone przez firmę Xilinx, dopasowane do generowania struktury sprzętowej. W implementacji wykorzystywano arytmetykę zmiennoprzecinkową pojedynczej precyzji typu float.

Po przygotowaniu kodu, narzędzia do generowania struktury logicznej dostarczają w pełni funkcjonalny moduł do wykorzystania w schemacie blokowym, w Vivado. Połączenie modułu w obrębie PL było analogiczne jak w poprzednich przypadkach, tj. zgodne ze schematem połączeń z rysunku 3.40, z dodaniem interfejsu AXI4-Lite dołączonego do bloku $ps7\_0\_axi\_periph$. Po stronie PS program działał w sposób analogiczny jak w przypadku podejścia trzeciego.

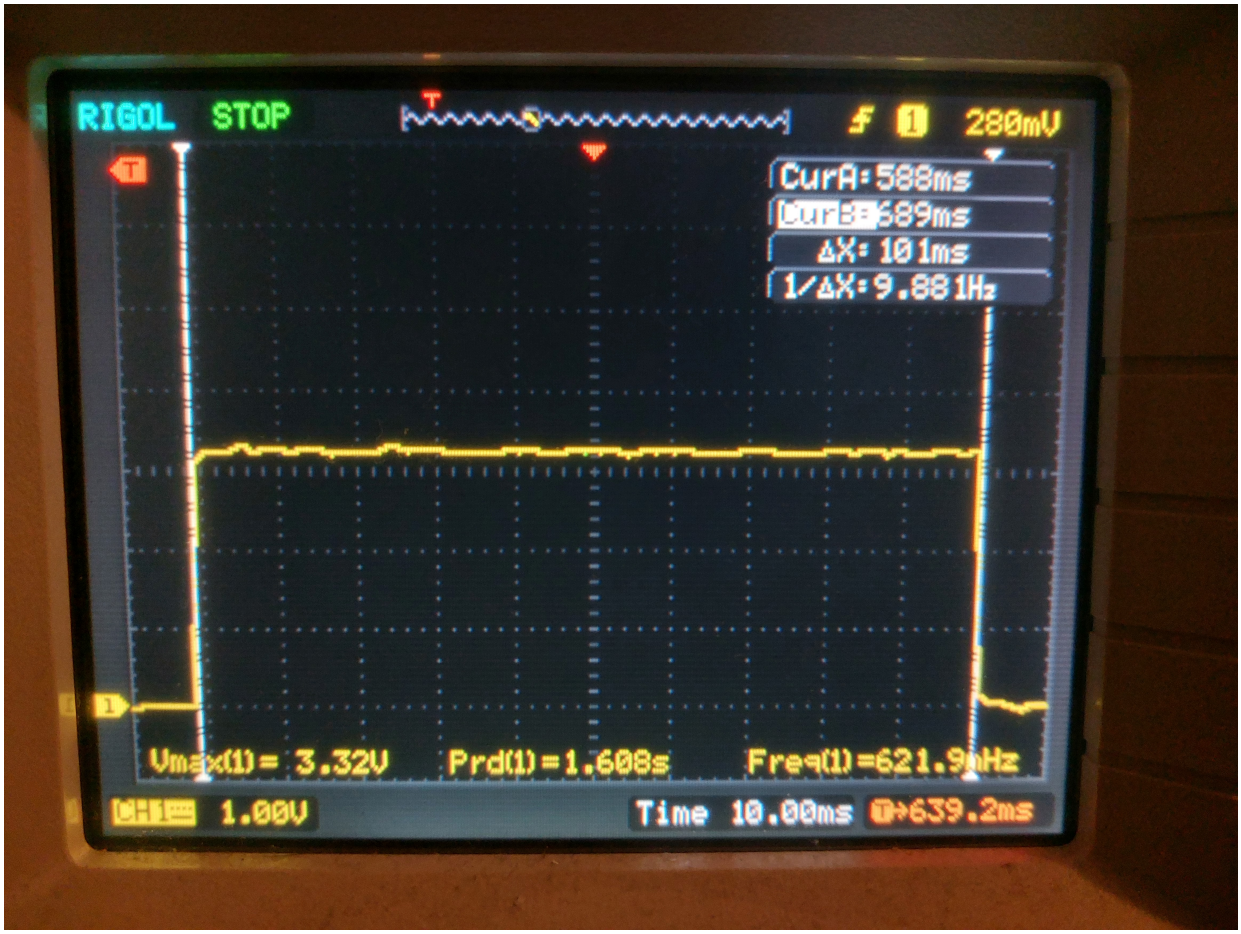
Weryfikacja funkcjonowania modułów przed pomiarami odbywała się w oparciu o symulacje działania logiki w module symulacyjnym programu Vivado, jak i na drodze analizy komunikacji pomiędzy PS i PL, możliwej do zarejestrowania oraz wyświetlenia przy wykorzystaniu modułu do debugowania programu Vivado. W przypadku podejścia z wykorzystaniem narzędzi do generowania struktury logicznej weryfikacja działania modułów opierała się głównie na analizie zarejestrowanej komunikacji pomiędzy PS i PL oraz na testach wykonanych w Vitis HLS, na podstawie napisanego tam zestawu testów. Weryfikacja w oparciu o symulację różniła się w zależności od podejścia implementacyjnego i dotyczyła głównie podejścia pierwszego i trzeciego. W przypadku podejścia pierwszego przygotowano 100 przykładowych zestawów danych wejściowych, które były kolejno transmitowane do badanego modułu symulując transakcję pomiędzy PS i PL. Początkowo sprawdzano sam sumator tj. to, czy zliczane wartości były poprawne oraz czy następowało wysterowywanie bloków liczących funkcje aktywacji. Następnie sprawdzano całość komórki pod kątem obliczanych wartości i inicjowania komunikacji zwrotnej. Otrzymane na etapie symulacji wyniki porównywano z programem referencyjnym napisanym w Pythonie. Po sprawdzeniu symulacyjnym sprawdzano działanie po wgraniu logiki do układu ZYNQ poprzez rejestracje komunikacji w programie Vivado. Sprawdzenia dokonano dla 10 sekwencji testo-

wych. Kod po stronie PS również był testowany funkcjonalnie i walidowany pod kątem poprawności, co miało miejsce dla każdego z rozważanych z przypadków, a wartości obliczone na poszczególnych etapach były porównywane z uzyskanymi przez program referencyjny. Dla przypadku podejścia trzeciego proces weryfikacji funkcjonalności był dużo bardziej złożony ze względu na liczbę rozważanych wariantów oraz złożoność projektowanej logiki w każdym z nich. Poszczególne opcje były sprawdzane w całości, a w ramach testów przygotowano 10 sekwencji testowych, które były transmitowane do testowanego modułu element po elemencie. Na początku sprawdzano poprawność działania bloków zwracających wartości wag i przesunięć dla konkretnego elementu sieci, sprawdzając przy tym poprawność zawartych tam danych oraz to, czy zwracane w danym etapie obliczeniowym wartości przyporządkowane są odpowiedniemu indeksowi elementu. Kolejnym krokiem było sprawdzenie działania sumatora oraz wartości poszczególnych sygnałów podawanych na wejścia bloków funkcji aktywacji. Sprawdzano też czy w poprawny sposób i czy we właściwym dla obliczeń momencie następowała komunikacja z blokami odpowiadającymi za obliczanie wartości funkcji aktywacji. Następnie sprawdzono działanie całej warstwy, tj. wyników uzyskanych po wykonaniu obliczeń dla całej warstwy LSTM oraz po przetworzeniu całej sekwencji. Poza testami symulacyjnymi sprawdzano komunikację pomiędzy modułami PS i PL oraz wartości otrzymywane po przesłaniu wyniku obliczeń do PS. Na każdym etapie porównywano wartości z programem referencyjnym napisanym w Pythonie.

3.6.3. Pomiary i zestawienie wyników

W celu wykonania pomiarów czasu obliczeń wykorzystano oscyloskop RIGOL DS1022C, który został podłączony bezpośrednio do wyprowadzeń płytki ewaluacyjnej o oznaczeniu JB. Są one dwurzędową listwą złączową, do której doprowadzone jest m.in. wyprowadzenie $Y7$ układu ZYNQ. W przypadku pomiaru czasu obliczeń klasyfikatora mierzony był czas pojedynczej klasyfikacji. Sygnał na wyprowadzeniu $Y7$ ustawiany był w stan wysoki tuż przed rozpoczęciem obliczeń, a z powrotem w stan niski od razu po zakończeniu obliczeń i odebraniu strumienia danych. Zmierzono czasy dla czterech podejść implementacyjnych przedstawionych w punkcie 3.6.2. Podejście pierwsze cechowało się lokalizacją parametrów sieci po stronie PS. W trakcie eksperymentów związanych z pierwszym podejściem, dodatkowo w kolejnych punktach pomiarowych zmieniano nastawy bloków arytmetycznych w strukturze PL od 0 cykli do 4. Podejście drugie oparte było na programie uruchamianym w całości po stronie

PS. Trzecie podejście cechowało przeniesienie parametrów sieci na stronę PL. Rozważono wersję z różną liczbą zaimplementowanych komórek oraz różną liczbą par bloków arytmetycznych. Ostatnie, czwarte podejście, polegało na wykorzystaniu narzędzi do generowania struktury logicznej i sprawdzeniu oraz porównaniu parametrów wygenerowanej sieci z siecią zaprojektowaną ręcznie. Czas obliczeń był mierzony w taki sam sposób, niezależnie od podejścia implementacyjnego. Na rysunku 3.43 przedstawiono przykładowy przebieg uchwycony w czasie pomiaru czasu obliczeń dla pierwszego podejścia w przypadku, gdy bloki arytmetyczne ustawione były na 2 takty zegara.



Rysunek 3.43: Pomiar czasu obliczeń

Wyniki pomiarów oraz parametry implementacji znajdują się w tabeli 3.22. Kolumna *Rodzaj* opisuje to jakiego podejścia i wariantu dotyczą wyniki. Poszczególne podejścia są od siebie odseparowane poziomą linią. Kolumna *Częstotliwość taktowania* zawiera rzeczywistą częstotliwość taktowania logiki, która jest skwantowana do konkretnych wartości ze względu na wystąpienie podzielnika częstotliwościowego. Jest to maksymalna osiągalna częstotliwość taktowania logiki, która jest najbliższa, ale wciąż

mniejsza od maksymalnej częstotliwości taktowania wyliczonej przez Vivado. Kolumna *Czas obliczeń* zawiera wartości odczytane z oscyloskopu na podstawie obserwacji przebiegu na porcie $Y7$.

Tabela 3.22. Implementacja klasyfikatora na platformie ZYNQ dla czterech różnych podejść implementacyjnych (XC7Z020-1CLG400C)

Rodzaj	LUT	FF	DSP	BRAM	Częstotliwość taktowania [MHz]	Czas obliczeń [ms]
Param. na PS, 0 cykli	6962	8194	12	6	38.6	100
Param. na PS, 1 cykl	6961	8481	12	6	49.3	104
Param. na PS, 2 cykle	6773	8692	12	6	81.3	101
Param. na PS, 3 cykle	6767	8740	12	6	96.0	102
Param. na PS, 4 cykle	6826	8874	12	6	101.1	103
ZYNQ PS(ARM)	426	543	0	0	132.3	6.8
Param. na PL, 1LSTM, 1p_aryt	12910	10784	12	2	47.6	2.88
Param. na PL, 1LSTM, 2p_aryt	13865	10894	16	2	47.6	1.70
Param. na PL, 1LSTM, 4p_aryt	14915	11145	24	2	47.6	1.14
Param. na PL, 2LSTM, 1p_aryt	22988	12148	24	2	47.6	1.50
Param. na PL, 2LSTM, 2p_aryt	22534	12228	32	2	47.6	0.892
Param. na PL, 2LSTM, 4p_aryt	20346	12937	48	2	47.6	0.612
Param. na PL, 4LSTM, 1p_aryt	37182	14745	48	2	47.6	0.792
Param. na PL, 4LSTM, 2p_aryt	32412	13549	64	2	47.6	0.484
Param. na PL, 4LSTM, 4p_aryt	29195	15718	96	2	47.6	0.342
HLS	14555	11708	71	21	66.7	4.56

Porównując ze sobą wszystkie cztery podejścia, widać, że zastosowanie części PL do realizacji obliczeń może znacząco polepszyć czas obliczeń, natomiast przy źle dobranej architekturze i funkcjonalności PL, lub przy zbyt małej liczbie dostępnych zasobów, można uzyskać rezultat przeciwny do zamierzonego. W przypadku podejścia pierwszego, w którym dane były magazynowane po stronie PS, w celu zminimalizowania zużycia zasobów sprzętowych, osiągano czasy obliczeń na poziomie $100\,ms$, podczas gdy obliczenia wykonywane bezpośrednio na procesorze (podejście drugie) trwały $6.8\,ms$. Jest to bardzo duża różnica i stanowczo na niekorzyść podejścia pierwszego, które wiąże się z dużą liczbą transmisji pomiędzy PS i PL oraz bardzo dużą liczbą da-

nych w pojedynczej transakcji w porównaniu z pozostałymi podejściami, co jak widać po wynikach znacząco wpływa na czas obliczeń.

W trakcie eksperymentów sprawdzono implementacje oparte o różne nastawy opóźnienia w blokach arytmetycznych. Zmiana opóźnienia wpływa bezpośrednio na wygenerowaną strukturę logiczną bloku arytmetycznego, co widocznie przekłada się bezpośrednio na częstotliwość taktowania. Dla nastawy na 0 cykli częstotliwość taktowania wyniosła 38.6 MHz, podczas gdy przy nastawie na 4 cykle wyniosła ona 101.1 MHz. Tak duża zmiana częstotliwości taktowania nie przełożyła się jednak na szybkość obliczeń, która okazała się niemal jednakowa, niezależnie od nastawy. Zmiana nastawy wpłynęła też nieznacznie na ilość wykorzystywanych zasobów. Bardziej widoczne jest to na przerzutnikach, gdyż ich zużycie zwiększyło się z 8194 do 8874. Pomimo niekorzystnych wyników w dziedzinie czasu, podejście pierwsze cechuje bardzo małe zużycie zasobów sprzętowych.

Analizując wyniki dla podejścia trzeciego, w którym dane zostały przeniesione na stronę PL, zaobserwować można bardzo dobre rezultaty w sensie czasu obliczeń. Eksperymenty wykonano dla różnej liczby komórek oraz różnej liczby par bloków arytmetycznych przypadających na komórkę. Dla przypadku jednej komórki LSTM i jednej pary bloków arytmetycznych czas obliczeń wyniósł 2.88 ms, co jest wynikiem 2.36 razy szybszym niż w przypadku obliczeń na procesorze i 34.7 razy szybszym niż w przypadku obliczeń w podejściu pierwszym. Zwiększanie liczby komórek w strukturze logicznej oraz liczby bloków arytmetycznych skutkuje skróceniem czasu obliczeń jeszcze bardziej. Dla przypadku 4 komórek i 4 par bloków arytmetycznych czas obliczeń wynosiósł 0.342 ms, co jest wynikiem 19.9 razy szybszym niż w przypadku obliczeń wykonywanych na procesorze i 292.4 razy szybszym niż dla podejścia pierwszego. Lepsze wyniki w dziedzinie czasu wiążą się z większym zużyciem zasobów sprzętowych. Najmniejsza implementacja z podejścia trzeciego zużywała 1.85 razy więcej bramek LUT niż w przypadku podejścia pierwszego. Przyrost zużycia przerzutników nie był już tak duży, bowiem zwiększyło się ono 1.32 razy. Wraz ze zwiększaniem się liczby komórek LSTM i liczby par bloków arytmetycznych na komórkę znacząco wzrosło zużycie bloków DSP. W przypadku jednego neuronu i jednej pary bloków arytmetycznych zużycie bloków DSP było takie samo jak w przypadku podejścia pierwszego, natomiast dość szybko rosło i dla 4 komórek oraz 4 par bloków na komórkę osiągnęło 96 bloków DSP. Zauważyć można, że zmalało zużycie bloków BRAM w stosunku do podejścia

pierwszego. Jest to związane z tym, że logika związana z transmisją przez DMA nie potrzebuje tak dużych buforów. W podejściu trzecim przesyła się jedynie analizowany szereg i wynik obliczeń. Niezależnie od liczby komórek i liczby par bloków arytmetycznych osiągano w trakcie eksperymentów taką samą częstotliwość taktowania.

Ostatnim z rozważanych podejść była implementacja wykorzystująca narzędzia do generowania opisu struktury logicznej na bazie kodu i dyrektyw preprocesora. W tym podejściu również parametry sieci były przechowywane po stronie PL. Zauważyć można, że czas obliczeń uzyskany przy tym podejściu był nieznacznie mniejszy od obliczeń wykonywanych na procesorze i zauważalnie większy niż dla własnoręcznie wykonanych implementacji w podejściu trzecim pomimo wyższej częstotliwości taktowania. Zauważyć też można duże zużycie bloków BRAM i DSP. Porównywalna implementacja podejścia trzeciego, w sensie zużycia tablic LUT i przerzutników (oznaczona jako $Param.\ na\ PL$, $1LSTM$, $4p\_aryt$), była 4 razy szybsza i zużywała 2.96 razy mniej bloków DPS oraz 10.5 razy mniej bloków BRAM. Implementacja podejścia trzeciego zużywająca podobną liczbę bloków DSP (najbliższa jest oznaczona jako $Param.\ na\ PL$, $4LSTM$, $2p\_aryt$) była 9.42 razy szybsza, ale zużywała 2.23 razy więcej tablic LUT.

Tabela 3.23. Zestawienie czasów obliczeń dla zadania klasyfikacji

Wariant	Czas obliczeń	$\frac{T_{GPU1}}{T_x}$	$\frac{T_{GPU2}}{T_x}$	$\frac{T_{PC}}{T_x}$	$\frac{T_{RPi3}}{T_x}$
Wind+Keras+GPU1	30.8 ms	1.00	1.04	1.52	0.36
Ubun+Keras+GPU2	32.0 ms	0.96	1.00	1.47	0.35
Python+CPU	46.9 ms	0.66	0.68	1.00	0.24
Raspberry Pi 3	11.2 ms	2.75	2.86	4.19	1.00
Wariant 1	65.9 μs	467	486	712	170
Wariant 2	19.2 μs	1604	1667	2443	583
Wariant 3	19.2 μs	1604	1667	2443	583
Wariant 4	17.5 μs	1760	1829	2680	640
ZYNQ PS+PL, param. na PS	100 ms	0.31	0.32	0.47	0.11
ZYNQ PS (ARM)	6.8 ms	4.53	4.71	6.89	1.65
ZYNQ PS+PL, param. na PL	342 μs	90.1	93.6	137	32.7
HLS	4.56 ms	6.75	7.02	10.3	2.46

W tabeli 3.23 przedstawiono zestawienie czasów obliczeń uzyskanych za pomocą platformy ZYNQ do wyników prezentowanych w punkcie 3.5.5 oraz do wyników uzyskanych za pomocą programów referencyjnych. W tabeli zawarto najlepsze wyniki dla

każdego z rozważanych podejść. Wykorzystanie platformy ZYNQ, związana z tym dużo mniejsza pula zasobów sprzętowych i konieczność zmiany architektury sieci bezpośrednio przekłada się na czas obliczeń, co uwidoczniło się w wynikach pomiarów. Poprzednio najwolniejszy wariant 1 okazał się być i tak szybszy niż najszybsze z podejść wykorzystanych przy projektowaniu struktury logicznej na platformie ZYNQ, natomiast zajął dużo więcej zasobów sprzętowych. W porównaniu do programów referencyjnych podejście trzecie okazało się zauważalnie szybsze. W stosunku do GPU1 było 90.1 razy szybsze, a w stosunku do GPU2 o 93.6 razy. Podejście to było również 32.7 razy szybsze od implementacji na Raspberry Pi 3, które reprezentuje podejście wydające się być największą konkurencją jeśli chodzi o potencjalne zastosowania w systemach wbudowanych.

Dodatkowo zaletą podejścia trzeciego było nieobciążanie procesora w trakcie obliczeń związanych z siecią LSTM. Różnica czasu obliczeń pomiędzy wykonaniem obliczeń na Raspberry Pi 3 a procesorem ARM będącego częścią układu ZYNQ była bardzo mała i przede wszystkim spowodowana tym, że na procesorze ARM program wykonywał się bezpośrednio - bez pośredniczenia systemu operacyjnego. Z kolei podejście z użyciem HLS pokazuje, że pomimo zdecydowanego ułatwienia w procesie projektowania struktury logicznej, narzędzia HLS mogą wygenerować strukturę, której parametry pracy mogą nie być przekonujące do zastosowania układów programowalnych w danym projekcie. W omawianym przypadku, chcąc maksymalnie wykorzystać możliwości układów FPGA, należy strukturę logiczną zaprojektować od początku ręcznie, rozważając i optymalizując poszczególne stopnie. Wyniki podejścia pierwszego pokazują też, że podczas projektowania tego typu układów należy minimalizować potrzebę transmisji danych do niezbędnego minimum, gdyż wpływa to bezpośrednio i bardzo negatywnie, na czas obliczeń oraz bardziej obciąża procesor odbiorem i przygotowaniem kolejnego pakietu danych do przesłania.

3.6.4. Podsumowanie

W ramach implementacji z wykorzystaniem układu ZYNQ rozważono cztery podejścia implementacyjne, które dają się łatwo zaadaptować do implementacji dowolnej sieci składającej się z warstwy LSTM. Pierwsze podejście polegało na implementacji pojedynczej komórki LSTM w strukturze PL, a parametry sieci były trzymane po stronie PS. Było to podejście zakładające wykorzystanie minimalnej ilości zasobów sprzętowych. Taka implementacja wiązała się z intensywną komunikacją pomiędzy

PS i PL, co przełożyło się na długi czas obliczeń, który wyniósł 100 ms, natomiast osiągnięto cel w postaci minimalnego wykorzystania zasobów, które wyniosło 6962 tablic LUT, 8194 FF, 12 DSP i 6 BRAM dla nastawy bloków arytmetycznych na 0 cykli. Przy okazji eksperymentów badano wpływ nastaw bloków arytmetycznych na czas obliczeń i częstotliwość taktowania. Zauważono, że maksymalna częstotliwość taktowania jaką można było taktować układ, a która była osiągalna dla źródła sygnału zegarowego, rosła wraz z liczbą cykli potrzebnych na wykonanie obliczeń w blokach arytmetycznych. Natomiast nie przekładało się to na czas obliczeń, który był dla każdej nastawy porównywalny. Drugie podejście polegało na wykonaniu obliczeń w całości po stronie PS, co okazało się względnie szybkie - porównując z obliczeniami referencyjnymi. Trzecie podejście polegało na przeniesieniu parametrów do PL i rozważeniu różnej liczby zaimplementowanych komórek oraz różnej liczby par bloków arytmetycznych. Okazało się, że takie podejście jest dużo szybsze od pierwszego, jaki i od drugiego, i wypada bardzo dobrze w stosunku do obliczeń wykonywanych w programach referencyjnych. Pojemność rozważanego układu pozwoliła na implementację 4 komórek i 4 par bloków arytmetycznych, co okazało się 90.1 razy szybsze niż GPU1 (NVIDIA GeForce GTX 1060), 93.6 razy szybsze niż GPU2 (NVIDIA GeForce RTX 3080 Ti), 137 razy szybsze niż obliczenia wykonywane na procesorze z wykorzystaniem Pythona i biblioteki NumPy oraz 32.7 razy szybsze niż obliczenia wykonywane na Raspberry Pi 3. Ostatnie z podejść polegało na wykorzystaniu narzędzi HLS do wygenerowania struktury logicznej. Takie podejście okazało się tylko nieznacznie szybsze od obliczeń wykonywanych na procesorze ARM, będącym częścią układu ZYNQ oraz wolniejsze od podejścia trzeciego (nawet od minimalnego wariantu tego podejścia).

Przedstawione wyniki, szczególnie w przypadku podejścia trzeciego, pokazują, że wykorzystanie platformy ZYNQ w prezentowanym zastosowaniu ma sens i może okazać się bardzo korzystne, szczególnie z uwagi na to, że wykonywane obliczenia w strukturze PL nie angażują procesora i przez to go odciążają. W przypadku dostatecznie dużej ilości zasobów sprzętowych, możliwa byłaby sytuacja, że logika po stronie PL mogłaby nadzorować pracę maszyny i wykonywać zadanie detekcji zużycia narzędzi w pełni niezależnie od procesora. Logika odpowiadałaby wtedy za realizację całego procesu związanego z klasyfikacją, tj. odbieranie danych z czujnika, wykonywanie wstępnej filtracji, wykonywanie klasyfikacji oraz podjęcie decyzji o zatrzymaniu maszyny, a jedynie informując procesor o podjętych działaniach. Warto podkreślić,

że wykorzystany układ nie jest największym układem z serii, a został wybrany ze względu na dostępność. Największy tj. XC7Z100 posiada 277 tys. tablic LUT, 554.8 tys. przerzutników oraz 2020 DSP, co pozwoliłoby na implementację wspomnianego scenariusza, z wykorzystaniem w pełni zalet przeprowadzania obliczeń w sposób równoległy w strukturze FPGA, w sposób zaprezentowany w punkcie 3.5.

Układy z serii ZYNQ są zdecydowanie wygodniejszym rozwiązaniem w przypadku współpracy logiki programowalnej z procesorem niż wykorzystanie osobnych układów - o ile nie ma ograniczeń w stosunku do zasobów sprzętowych. Zalety są łatwo dostrzegalne na etapie prototypowania, gdzie jednym przewodem podłączonym do płytki ewaluacyjnej można ją zasilić, wgrać dane do procesora i do struktury logicznej, prowadzić komunikację oraz debugować. Wykorzystywana płyta ewaluacyjna jest też znacznie tańsza od wykorzystywanej w przypadku pełnej implementacji sieci LSTM na FPGA, co przedstawiono w zestawieniu, w punkcie 4.5.

Po układy z serii ZYNQ chętnie sięgają autorzy cytowanej literatury chcący wykonać część obliczeń w strukturze logicznej. Obecnie nie ma wielu, precyzyjnie udokumentowanych prac naukowych, które poruszałyby tematykę implementacji sieci LSTM na FPGA, jednak pojawiające się prace częściej wykorzystują układy typu ZYNQ niż inne. Przykładem pracy, dotykającej takiej tematyki, jest [102], w której modelowano sieć zawierającą dwie warstwy LSTM po 128 komórek. Podejście implementacyjne składało się z implementacji jednej komórki w PL i logiki koordynującej obliczenia. Jest podobne do przedstawianego w niniejszej pracy, zawiera jednak znacznie uproszczone funkcje aktywacji i stosuje arytmetykę Q8.8. W pracy [102] uzyskano podobny stopień przyspieszenia obliczeń jak w niniejszej pracy, względem obliczeń na procesorze ARM wbudowanym w układ ZYNQ, ale dla znacznie prostszego modelu komórki. Autorzy artykułu twierdzą, że uzyskali błąd na wyjściu sieci na poziomie 7.1%, co w przypadku zadania klasyfikacji może być problematyczne, a w przypadku rozważanej w niniejszej pracy sieci do zadania detekcji wadliwego wykucia w procesie kucia na zimno, taka dokładność byłaby dyskwalifikująca. Innym przykładem jest [119], gdzie autorzy prezentują trzy podejścia implementacyjne i osiągają stosunkowo duże błędy dla parametrów wyjściowych sieci, tj. średnio 3.9% dla Ct oraz 2.8% dla ht. Autorzy nie porównują prezentowanego rozwiązania z podejściem programowym oraz nie prezentują analizy wyników pracy sieci. W literaturze przedmiotu często podaje się, iż prezentowana w danej publikacji struktura logiczna została wygenerowana za pomocą

narzędzi HLS, a nie zakodowana ręcznie. W nawiązaniu do [88], można stwierdzić, że nie jest to podejście optymalne i otrzymane w cytowanej literaturze wyniki mogą być lepsze kosztem poświęconego czasu na rozwój struktury logicznej. Taki pogląd został potwierdzony w niniejszej pracy, co widoczne jest porównując czwarte podejście implementacyjne z drugim i trzecim.

4. Uczenie sieci LSTM z wykorzystaniem układów FPGA

Proces uczenia sieci jest bardzo czasochłonną częścią projektowania modelu, a czas obliczeń silnie zależy od rodzaju sieci. Im jest ona bardziej złożona, tym dłuższy jest proces uczenia. Wydłużanie się czasu obliczeń łatwo zauważyć w przypadku rekurencyjnych sieci neuronowych, w których rośnie on zauważalnie wraz z długością przetwarzanej sekwencji lub ze wzrostem liczby neuronów w warstwie rekurencyjnej. W przypadku sieci rekurencyjnych, jaką jest m.in. LSTM, w celu obliczenia błędu od każdego z parametrów sieci, wykorzystuje się rozbudowany algorytm propagacji wstecznej w czasie (BPTT - Back Propagation Through Time). Pozwala on na uwzględnienie zależności czasowych powstających pomiędzy kolejnymi etapami przetwarzania sekwencji danych. Obliczony błąd uwzględniany jest w aktualizacji parametrów sieci, która przebiega zgodnie z założonym algorytmem optymalizacyjnym. Ze względu na złożoność obliczeniową procesu uczenia rzadko wykorzystuje się w tym celu samo CPU, lecz korzysta się z różnych dodatkowych platform, np. GPU, czy obliczeń w chmurze, umożliwiających wykonywanie obliczeń w sposób znacznie bardziej wydajny.

Ze względu na popularność i bardzo dobre wyniki osiągane z użyciem m.in. GPU, jak i poziom trudności w projektowaniu wydajnych układów sprzętowych, temat modułów sprzętowych wspomagających proces uczenia, opartych o FPGA, nie pojawia się zbyt często w literaturze badawczej. Łatwo natomiast zauważyć zastosowania, w których brak jest dostępu do któregoś z wymienionych rozwiązań lub podejście oparte o moduły przeznaczone do procesu uczenia zrealizowane na FPGA zdają się być lepszym rozwiązaniem. Rozwiązania oparte na FPGA mogą być szczególnie przydatne m.in. w sytuacjach rygorystycznych ograniczeń energii, w środowisku utrudniającym lub uniemożliwiającym proces chłodzenia, lub gdy dysponuje się słabą jednostką obliczeniową i nie ma możliwości podłączenia się do sieci. Moduł uczący mógłby również współpracować z procesorami lub mikrokontrolerami jako układ peryferyjny i znacząco przyspieszyć obliczenia związane z uczeniem sieci, co mogłoby się okazać korzystne np. w przypadku robotów wykorzystywanych w misjach kosmicznych.

W [120] autor podjął próby wykonania modułu uczenia sieci LSTM wykorzystując implementację z funkcją aktywacji wzorowaną na algorytmie CORDIC (ale bezpośrednio z niego nie korzystając), o której mowa w punkcie 3.3.2. Uzyskane wyniki zaowocowały w dalszych badaniach, które zostały znacząco rozszerzone i przedsta-

wione w niniejszej pracy. Dalsze rozważania obejmowały wykorzystanie ulepszonej wersji komórki LSTM, z funkcją aktywacji wzorowanej na CORDIC oraz pozostałych wariantów komórki LSTM, przedstawionych w punktach 3.3.3 i 3.3.4. W eksperymentach wykorzystano architekturę sieci przedstawioną w punkcie 3.5.3. Celem badań było sprawdzenie ilości wykorzystywanych zasobów sprzętowych, potencjalnych korzyści m.in. w postaci czasu obliczeń czy złożoności tego typu projektów. Istotne było również wykonanie analizy wpływu dokładności aproksymacyjnej funkcji aktywacji na proces uczenia.

Głównym przedmiotem rozdziału są rozważania dotyczące algorytmu propagacji wstecznej w czasie dla sieci LSTM. Rozważania dotyczyły zarówno czasu obliczeń jak i osiąganej dokładności po całym cyklu uczenia w stosunku do programów referencyjnych. W obliczeniach, jako optymalizator, wykorzystany został algorytm SGD (Stochastic Gradient Descent), który w bardzo małym stopniu wpływa na czas obliczeń i złożoność kodu, co umożliwia maksymalne wyeliminowanie czynników wpływających na analizę algorytmu BPTT. W celu porównania czasu obliczeń, poza implementacjami na FPGA oraz ZYNQ, wykonano trzy dodatkowe implementacje referencyjne. Dwie implementacje uruchamiane były na komputerze PC. Obie napisane zostały w Pythonie. Pierwsza opierała się o ręcznie napisany kod, bez wykorzystania żadnych bibliotek do uczenia głębokiego oraz bez przekierowania obliczeń na GPU. Druga wykorzystywała bibliotekę Keras i przekierowywała obliczenia na GPU, co odpowiada popularnej obecnie metodzie obliczeń związanych z sieciami neuronowymi. Implementacja była uruchamiana na dwóch różnych komputerach z różnymi GPU. Trzecia implementacja została napisana w C++11 i uruchamiana była na Raspberry Pi 3, które wyposażone jest w popularny czip Broadcom BCM2837 z rdzeniem, z rodziny ARM-8 Cortex-A53.

Poza rozważaniami dotyczącymi algorytmu BPTT, w rozdziale przedstawiono analizę wpływu liczby komórek na czas obliczeń z wykorzystaniem optymalizatora Adam. Przedstawiono również rozważania dotyczące wpływu dokładności aproksymacji na proces uczenia oraz wykorzystanie układu typu ZYNQ do wykonania procesu uczenia z wykorzystaniem optymalizatora Adam.

4.1. Sprzętowe uczenie sieci LSTM w literaturze badawczej

W ciągu ostatnich lat opublikowano zaledwie kilka prac dotyczących implementacji algorytmu propagacji wstecznej na układach programowalnych. W [121] autorzy skupili się na maksymalizacji ilości obliczeń wykonywanych równolegle w standardowym algorytmie propagacji wstecznej. Wykorzystali w tym celu różne techniki optymalizacji np. rozwijanie pętli i wykonywanie ich w sposób kombinacyjny oraz optymalizację przepływu danych w obrębie sieci. Wyniki optymalizacji poprawiły wydajność obliczeniową sieci na FPGA w porównaniu z implementacjami na komputerze.

Praca [122] przedstawia implementację algorytmu propagacji wstecznej dla perceptronu wielowarstwowego, który osiąga dokładność na poziomie 98% w stosunku do analogicznej implementacji w MATLAB-ie.

Autorzy w [77] podjęli się próby implementacji algorytmu propagacji wstecznej na FPGA oraz na mikrokontrolerach, skupiając się na wydajnej sprzętowo implementacji i minimalizując przy tym zużycie zasobów. W implementacji użyto bloków DSP w celu optymalizacji obliczeń. Zastosowano też implementację stałoprzecinkową. Autorzy twierdzą, że implementacja na FPGA osiągnęła większą szybkość obliczeń niźli na porównywanym mikrokontrolerze.

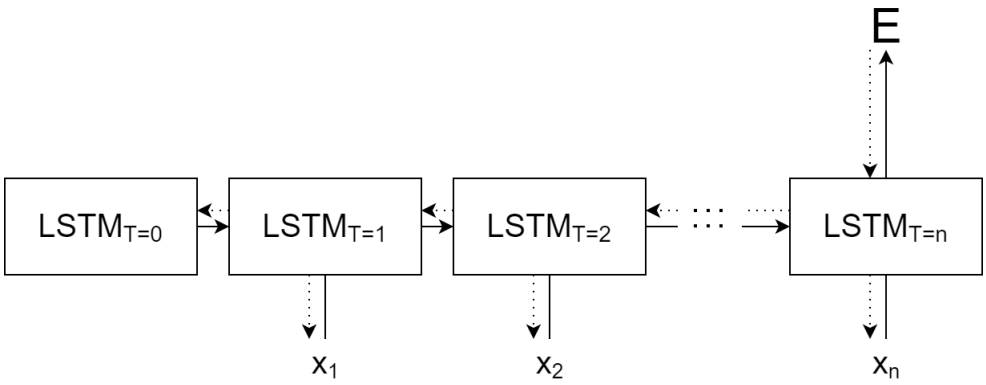
W [123] przedstawiono implementację propagacji wstecznej dla wielowarstwowej sieci neuronowej na FPGA, gdzie zauważono, że naturalne dla FPGA obliczenia przeprowadzane równolegle stwarzają duże możliwości w optymalizacji prędkości uczenia.

Temat implementacji sprzętowej algorytmu propagacji wstecznej w czasie lub akceleratorów sprzętowych procesu uczenia sieci rekurencyjnych nie został jeszcze dokładnie przebadany. Istnieje duże zapotrzebowanie na publikacje prezentujące szczegóły implementacyjne, porównanie wyników z innymi platformami, analizę jakościową działania przyuczonej sieci oraz wpływ założeń projektowych na jej pracę.

4.2. Algorytm BPTT

Jak wspomniano we wprowadzeniu do punktu 4, w celu uczenia nadzorowanego sieci rekurencyjnych wykorzystuje się rozbudowaną wersję algorytmu propagacji wstecznej, która uwzględnia zależności czasowe zachodzące w konkretnych warstwach sieci. Wersja ta nosi nazwę propagacji wstecznej w czasie lub skrótowo BPTT. Zależności czasowe w sieciach rekurencyjnych wynikają z uwzględniania w obliczeniach wartości

ze stanu poprzedniego. Jak pokazano na rysunku 4.44, przedstawiającym diagram przepływu sygnałów w poszczególnych stanach dla pojedynczej warstwy LSTM, w stanie $T = 1$, gdzie analizowana jest pierwsza wartość z sekwencji, uwzględnia się również wartości ze stanu początkowego $T = 0$, a wynik z $T = 1$ uwzględniany jest również w stanie $T = 2$, gdy na wejściu jest już druga wartość z procedowanej sekwencji. Proces zachodzi aż do ostatniej wartości z sekwencji [124].



Rysunek 4.44: Diagram przepływu sygnałów w warstwie LSTM

W zależności od architektury, wartość z wyjścia przekazywana jest dalej, do następnej warstwy lub obliczany jest błąd względem danych uczących. Propagacja wsteczna dla warstwy LSTM polega na podaniu obliczonej lub przekazanej z następnej warstwy wartości błędu, a następnie obliczenie błędu dla każdego z parametrów sieci. Obliczenia zaczynają się od ostatniego osiągniętego stanu $T = n$ oraz ostatniej wartości z analizowanej sekwencji x_n. Wartości błędów są następnie przekazywane do poprzedniego stanu, gdzie na wejściu danych znajduje się wartość x_{n-1} z analizowanej sekwencji.

Obliczenia związane z propagacją wsteczną w czasie, dla sieci LSTM są oparte o regułę łańcuchową opisaną w (4.26) oraz o pochodne funkcji aktywacji przedstawione w (4.27) oraz (4.28).

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y} \cdot \frac{\delta y}{\delta x} \tag{4.26}$$

$$\frac{\delta \sigma(x)}{\delta x} = \sigma(x) \cdot (1 - \sigma(x)) \tag{4.27}$$

$$\frac{\delta \tanh(x)}{\delta x} = 1 - \tanh^2(x) \tag{4.28}$$

W obrębie komórki LSTM, opisanej w punkcie 3, pochodne dla poszczególnych

sygnałów oblicza się według (4.29) - (4.36) [124].

$$\delta h_t = \Delta_t + \Delta h_t \tag{4.29}$$

$$\delta C_t = \delta h_t \odot o_t \odot (1 - \tanh^2(C_t)) + \delta C_{t+1} \odot f_{t+1} \tag{4.30}$$

$$\delta \hat{C}_t = \delta C_t \odot i_t \odot (1 - \hat{C}_t^2) \tag{4.31}$$

$$\delta i_t = \delta C_t \odot \hat{C}_t \odot i_t \odot (1 - i_t) \tag{4.32}$$

$$\delta f_t = \delta C_t \odot C_{t-1} \odot f_t \odot (1 - f_t) \tag{4.33}$$

$$\delta o_t = \delta h_t \odot \tanh(C_t) \odot o_t \odot (1 - o_t) \tag{4.34}$$

$$\delta x_t = W_x^T \odot \delta G_t \tag{4.35}$$

$$\Delta h_{t-1} = W_h^T \odot \delta G_t \tag{4.36}$$

Wartości błędów wykorzystywane później do aktualizacji wag i przesunięć przedstawione są w (4.37) - (4.39) [124].

$$\delta W_x = \sum_{t=0}^{T} \delta G_t \otimes x_t \tag{4.37}$$

$$\delta W_h = \sum_{t=0}^{T-1} \delta G_{t+1} \otimes h_t \tag{4.38}$$

$$\delta b = \sum_{t=0}^{T} \delta G_t \tag{4.39}$$

gdzie: $G_t = \begin{bmatrix} \hat{C}_t \\ i_t \\ f_t \\ o_t \end{bmatrix}; W_x = \begin{bmatrix} W_{x\hat{C}} \\ W_{xi} \\ W_{xf} \\ W_{xo} \end{bmatrix}; W_h = \begin{bmatrix} W_{h\hat{C}} \\ W_{hi} \\ W_{hf} \\ W_{ho} \end{bmatrix}; b = \begin{bmatrix} b_{\hat{C}} \\ b_i \\ b_f \\ b_o \end{bmatrix}$; W_{xY} - macierz wag dla sygnału wejściowego Y; W_{hY} - macierz wag dla sygnału rekurencyjnego Y; b_Y - macierz przesunięć dla sygnału Y.

4.3. Implementacja na FPGA

4.3.1. Sposób implementacji

Implementacja modułu uczenia przedstawiona została częściowo przez autora w [120] i skupia się na propagacji w przód oraz propagacji wstecznej w czasie. Mniej uwagi poświęcone zostało zagadnieniom takim jak wybór optymalizatora, ponieważ jest to w dużej mierze wybór projektowy, a niektóre optymalizatory mają bardzo mały

wpływ na czas obliczeń w porównaniu z BPTT, przykładem może być tu SGD, który wybrany został w przedstawianej implementacji. SGD aktualizuje wagi według (4.40).

$$W = W \ - \lambda \cdot \delta W \tag{4.40}$$

gdzie: λ - współczynnik uczenia, W - waga, δW - błąd od wagi W.

W celu uniezależnienia się od niuansów obliczeń stałoprzecinkowych (m.in. występowanie przepełnień) i potrzeby uwydatnienia różnicy, jaką wprowadza sama zmiana platformy obliczeniowej, wykorzystano arytmetykę zmiennoprzecinkową, zgodną z IEEE-754. Takie podejście ułatwia późniejszy proces transmisji danych do komputerów klasy PC lub mikrokontrolerów, ponieważ nie wymaga dalszej konwersji. Przedstawiana w tym punkcie implementacja uzupełnia implementację sieci LSTM dla procesu kucia na zimno, przedstawioną w punkcie 3.5, wykorzystuje dane tam opisane i używane. Sieć zbudowana została zatem z dwóch warstw, na które składa się warstwa LSTM z 44 neuronami oraz warstwa wyjściowa z jednym neuronem i sigmoidalną funkcją aktywacji. W trakcie propagacji w przód, na wejście podawano kolejno dane z sekwencji uczących oraz obliczano parametry wewnętrzne komórek, które następnie zostały zapisane w celu późniejszego wykorzystania przy propagacji wstecznej. Po przeprocedowaniu wszystkich danych z sekwencji, sygnały wyjściowe z warstwy LSTM przekierowane zostały do warstwy wyjściowej, gdzie przemnażano je przez wagi i dodawano wraz z przesunięciem, a następnie podawano do funkcji aktywacji. Wartość funkcji aktywacji jest wykorzystywana w celu obliczenia wartości błędu względem danych uczących, zgodnie z formułą średnio-kwadratowej funkcji straty przedstawionej w (4.41).

$$E = \frac{1}{2}(y - h_o)^2 \tag{4.41}$$

$$\frac{\delta E}{\delta h} = (h_o - y) \tag{4.42}$$

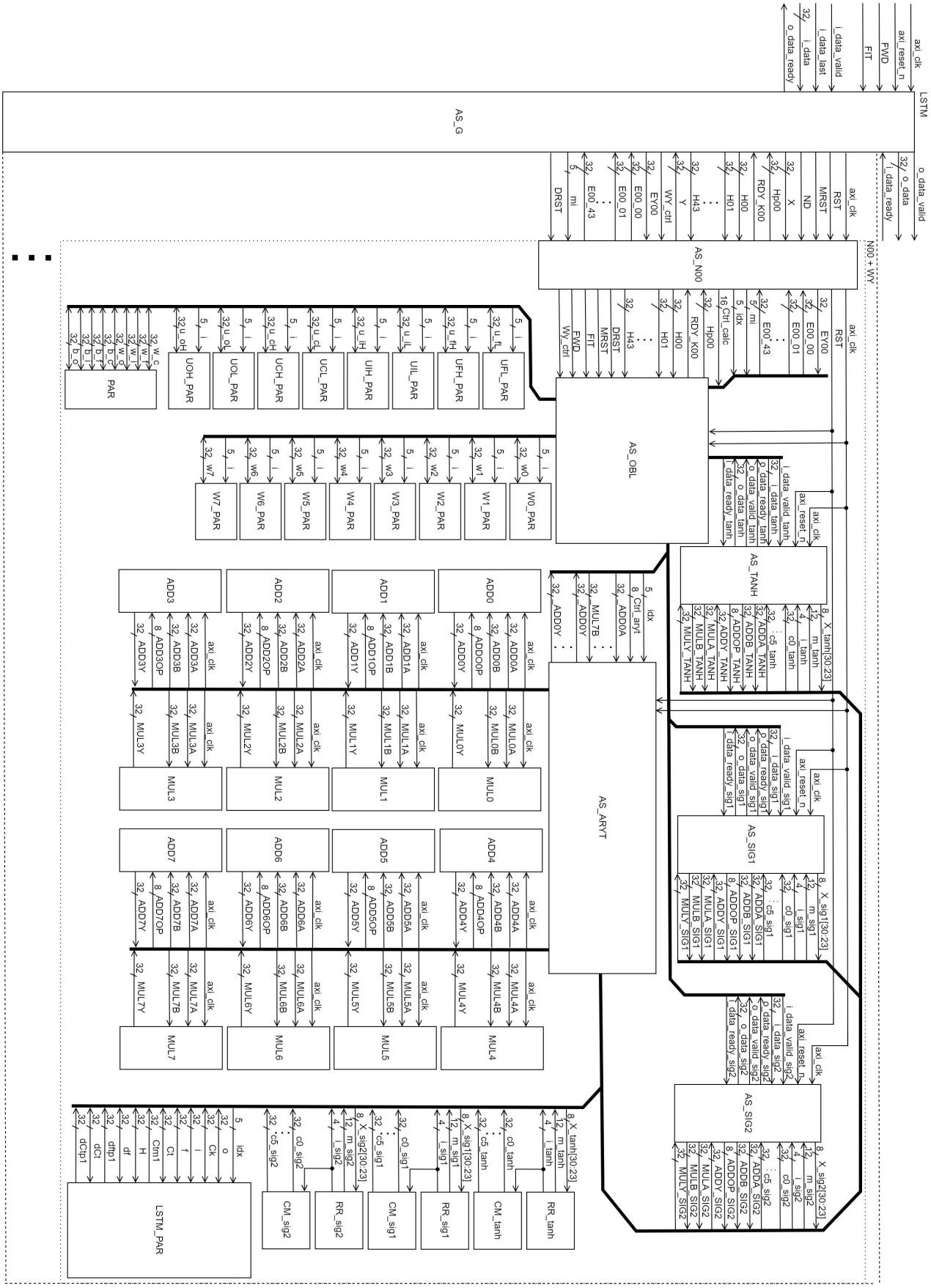
gdzie: h_o - wyjście sieci; y - wartość z danych uczących.

Pochodna wybranej funkcji straty przedstawiona jest w (4.42) i składa się jedynie z odejmowania, co dzięki zastosowaniu wirtualnych komponentów dostępnych w narzędziach firmy Xilinx, pozwala na otrzymanie wyniku tej operacji nawet w następnym cyklu zegara, w zależności od nastaw bloków wirtualnych komponentów. Warstwa wyjściowa składa się z jednego neuronu o sigmoidalnej funkcji aktywacji, którego pochodna

przedstawiona jest w (4.27). Korzystając z równań (4.42), (4.27) oraz z reguły łańcucha (4.26) obliczyć można błąd przed (patrząc od strony wejścia danych do sieci) sumatorem neuronu wyjściowego. Po przemnożeniu rezultatu przez wagi połączeń pomiędzy warstwą wyjściową a warstwą LSTM, błąd jest dalej podawany do warstwy LSTM, gdzie korzystając z równań (4.29) - (4.36) oblicza się parametry wewnętrzne komórki. W tym czasie wykorzystywane są wielkości parametrów obliczone w czasie propagacji w przód. Dla każdego kroku sygnały rekurencyjne muszą być rozprowadzone po całej warstwie. Najbardziej czasochłonną operacją wewnątrz komórki jest tu obliczenie funkcji tangensa hiperbolicznego, która normalnie musi być obliczana dwukrotnie w cyklu obliczeń dla jednej danej z sekwencji, natomiast tworząc konfigurację na FPGA można zadecydować o zapisie raz obliczonej wartości i ponownym wykorzystaniu. Na zewnątrz komórki bardzo czasochłonne są obliczenia opisane równaniami (4.35) - (4.36), czyli związane z przemnażaniem wartości błędu z wagami połączeń i dystrybucja ich w obrębie warstwy. Warto tutaj zauważyć, że każda komórka LSTM może przeprowadzać obliczenia równolegle, przez co obliczenie błędu w całej warstwie trwa tyle, co dla pojedynczej, najdłużej liczącej komórki. Wymagana tutaj jest synchronizacja obliczeń ze względu na połączenia pomiędzy komórkami w warstwie rekurencyjnej. W prezentowanej implementacji warstwa LSTM była od razu warstwą wejściową, co pozwalało na rozpoczęcie obliczeń z kolejną sekwencją na wejściu sieci lub przystąpienie do aktualizacji wag po zakończeniu obliczeń w obrębie warstwy LSTM.

Scenariusz treningowy, w przypadku pomiaru czasu obliczeń, składał się z 176 sekwencji uczących stanowiących jeden pakiet danych przetwarzanych po kolei. Dopiero po przetworzeniu całego pakietu danych wykonywana była aktualizacja wag. Ze względu na skupienie się, w trakcie eksperymentów związanych z czasem obliczeń, na realizacji algorytmu BPTT, z czego wynika minimalizacja wpływu optymalizatora w scenariuszu, aktualizacja wag przeprowadzona została tylko raz po przeprocedowaniu wszystkich sekwencji testowych. Synteza sieci została wykonana dla układu XCU250-FIGD2104-2L-E, będącego częścią Alveo U250 Data Center Accelerator i była porównana z trzema innymi implementacjami referencyjnymi: ręcznie napisanym kodem w Pythonie, który korzystał z biblioteki NumPy i pracował na CPU, z modelem zbudowanym w oparciu o Keras, korzystającym z GPU, uruchamianym na dwóch różnych komputerach, z dwoma różnymi GPU oraz implementacji napisanej w C++11 i uruchamianej na Raspberry Pi 3.

Na rysunku 4.45 przedstawiono schemat struktury logicznej zaimplementowanej na FPGA. Struktura logiczna bardzo przypomina tą z rys. 3.37. Tutaj również poszczególne komórki wyposażone są w trzy bloki wyliczające wartość funkcji aktywacji (2 bloki do obliczania wartości funkcji sigmoidalnej oraz jeden blok do obliczania wartości funkcji tangensa hiperbolicznego), w 8 par bloków arytmetycznych oraz w bloki przechowujące wartości wykorzystywane w trakcie aproksymacji funkcji (RR_* oraz CM_*). Różnica w strukturze logicznej komórki LSTM jest przede wszystkim widoczna w blokach związanych z parametrami, które w obecnym przypadku mogą zapisywać nową wartość parametru oraz w liczbie połączeń pomiędzy głównym automatem sekwencyjnym oraz jednostką obliczeniową. Jest to związane z dużo większą liczbą sygnałów jakie muszą być wykorzystane. Pierwsza komórka (o indeksie 0) została dodatkowo wyposażona w funkcjonalność warstwy wyjściowej, przez co błąd obliczony dla warstwy wyjściowej musi być dalej dystrybuowany do każdej komórki poprzedniej warstwy. Analogicznie jest w przypadku wykonywania przejścia w tył dla warstwy rekurencyjnej. Obliczony błąd na wejściu bramki musi zostać rozesłany w obrębie całej warstwy. Ponadto dochodzą sygnały sterujące, przez które przekazywana jest informacja o tym jakie obliczenia są w danym momencie realizowane w komórce, tj. czy jest to przejście w przód (sygnał FWD w stanie wysokim), czy w tył (sygnał FWD w stanie niskim). Dodatkowo występuje sygnał Wy_ctrl, który w przypadku komórki o indeksie 0 przełącza komórkę do wykonywania obliczeń związanych z warstwą wyjściową, a w przypadku pozostałych komórek wstrzymuje ich pracę. Z kolei sygnał FIT określa, czy komórka ma wykonać aktualizację wag. Dotyczy to wszystkich wag - również tych związanych z warstwą wyjściową. Bardzo ważnym blokiem, który został dodany na rysunku 4.45, w stosunku do rys. 3.37, jest blok odpowiedzialny za przechowywanie wartości sygnałów z komórki, obliczonych w trakcie przejścia w przód. Tym blokiem jest $LSTM\_PAR$, który poza przechowywaniem tych wartości, przechowuje też kilka wartości sygnałów obliczanych w trakcie przejścia w tył, a które są wykorzystywane w kolejnych krokach obliczeniowych. Największa z różnic w logice nie jest wprost widoczna na schemacie, natomiast zaznaczono ją w postaci powiększenia bloków AS_OBL i AS_ARYT. Znaczącemu powiększeniu uległy automaty sekwencyjne związane z wykonywanymi obliczeniami. W ramach algorytmu BPTT magazynuje się bowiem dane, wykonuje dodatkowe obliczenia i przygotowuje wartości dla znacznie większej liczby sygnałów wychodzących z komórki.



Rysunek 4.45: Struktura logiczna pełnej sieci LSTM z możliwością uczenia

Weryfikacja działania projektowanej logiki jest analogiczna do tej, przedstawionej w punkcie 3.5.4, lecz rozszerzona o funkcjonalność związaną z przejściem w tył i aktualizacją wag. W tym celu najpierw wykonano pełne przejście w przód całej sieci, a następnie podano wartość docelową dla badanej sekwencji i wykonano obliczenia związane z przejściem w tył przez neuron wyjściowy. Następnie sprawdzono wartości błędów podawane na komórki warstwy LSTM od strony neuronu wyjściowego. Kolejnym krokiem było sprawdzenie wartości sygnałów wewnątrz wszystkich komórek warstwy LSTM, które były zapisywane w trakcie przejścia w przód, a następnie wartości sygnałów rekurencyjnych po każdym kroku obliczeniowym. Po pełnym przejściu w tył sprawdzono wartości obliczonych błędów dla każdego z parametrów, a następnie uruchomiono proces aktualizacji wag i sprawdzono nowe wartości wag oraz przesunięć. Wyniki na każdym etapie porównywano z referencyjnym programem napisanym w Pythonie. Czynności weryfikacyjne powtórzono dla 5 różnych sekwencji testowych.

4.3.2. Wyniki implementacji

Analogicznie jak w punkcie 3.5, na LSTM wykonano cztery warianty implementacji modułu wspomagającego uczenie sieci LSTM, różniące się metodą obliczania wartości funkcji aktywacji, które przedstawiono w punkcie 3.3.5. Określenie czasu obliczeń na FPGA polegało na odczytaniu liczby cykli z symulacji FPGA wykonywanej w Vivado oraz przemnożenie tej liczby przez długość okresu zegara, wyznaczoną w trakcie syntezy, i kroku implementacyjnego przez oprogramowanie Vivado firmy Xilinx. Długość symulacji dla każdego z wariantów liczona była dla 176 danych uczących, zgodnie ze scenariuszem uczenia opisanym w punkcie 4.3. Po przetworzeniu całego pakietu danych uczących następowała jednokrotna aktualizacja wag w sieci. Wyniki pomiarów czasu zostały przedstawione w tabeli 4.1 wraz z estymowaną wartością wykorzystywanej mocy. Spośród implementowanych wariantów najszybsze okazały się warianty 2 i 3, które wykonywały obliczenia w takim samym czasie tj. 15.8 ms. Rezultat ten jest nieco szybszy w przypadku wariantu 4, który obliczenia wykonywał w 16.7 ms oraz dużo szybszy niż w przypadku wariantu 1, który wykonywał obliczenia w 34.1 ms i był taktowany najniższą częstotliwością zegara. Wariant 4 był taktowany częstotliwością zauważalnie mniejszą niż warianty 2 i 3, co wydłużyło czas obliczeń pomimo zajmowania mniejszej liczby cykli zegara. Warto dodać, że wprowadzone w niniejszej pracy ulepszenia dla wariantu 1 pozwoliły znacząco poprawić osiągi w porównaniu z pierwotną wersją z pracy [120].

Tabela 4.1. Porównanie czasów obliczeń

Wariant	Czas obliczeń [ms]	Liczba taktów zegara	Częstotliwość zegara [MHz]	Estymowana moc [W]
Wariant 1	34.1	1294515	38	36.8
Wariant 2	15.8	1080011	68	26.7
Wariant 3	15.8	1080011	68	22.4
Wariant 4	16.7	982859	59	26.4
Z artykułu [120]	74.6	2238622	30	-

W celu odniesienia wyników do powszechnie wykorzystywanych platform, wykonano trzy programy referencyjne. Pierwszy program wykorzystywał Pythona z biblioteką NumPy (która u podstaw ma język C) oraz przeprowadzał obliczenia na CPU. Drugi również wykorzystywał Pythona, ale obliczenia były wykonywane za pomocą biblioteki Keras, na dwóch różnych komputerach z różnymi GPU. Trzeci program referencyjny był napisany przy użyciu języka C++ i uruchamiany był na Raspberry Pi 3. W każdym z trzech przypadków wykonano 15 uruchomień w celu uniezależnienia wyników od chwilowej zajętości procesora lub karty graficznej. Spośród 15 uruchomień obliczano wartość średnią. Wyniki umieszczono w tabeli 4.2. Każdy z programów referencyjnych wykonywał pełny cykl uczenia opisany w punkcie 4.3. Sprzęt na którym wykonywano obliczenia referencyjne został opisany w punkcie 2.

Porównanie czasów obliczeń wygląda bardzo korzystnie dla podejścia z użyciem FPGA. Najszybsze implementacje (wariant 2 i 3) były dużo szybsze od obliczeń wykonywanych programowo. W porównaniu z GPU1, warianty 2 i 3 były 60.8 razy szybsze, w porównaniu z GPU2 48.4 razy, a w porównaniu z obliczeniami z wykorzystaniem Pythona na CPU nawet 269 razy szybsze. Nawet najwolniejszy z wariantów, tj. wariant 1, okazywał się szybszy od obliczeń na GPU1 - 28 razy i na GPU2 - 22.4. Bazując na wynikach, można stwierdzić, że wykorzystanie FPGA do tego typu obliczeń jest obiecujące i przy złożonych obliczeniach można zaoszczędzić sporo czasu obliczeniowego szczególnie, jeśli alternatywą są obliczenia wykonywane na samym procesorze.

Tabela 4.2. Zestawienie czasów obliczeń

Wariant	Czas obliczeń	$\frac{T_{GPU1}}{T_x}$	$\frac{T_{GPU2}}{T_x}$	$\frac{T_{PC}}{T_x}$	$\frac{T_{RPi3}}{T_x}$
Wind+Keras+GPU1	960 ms	1.00	0.80	4.43	1.91
Ubun+Keras+GPU2	765 ms	1.25	1.00	5.55	2.39
Python+CPU	4250 ms	0.23	0.18	1.00	0.43
Raspberry Pi 3	1830 ms	0.52	0.42	2.32	1.00
Wariant 1	34.1 ms	28.2	22.4	125	53.7
Wariant 2	15.8 ms	60.8	48.4	269	116
Wariant 3	15.8 ms	60.8	48.4	269	116
Wariant 4	16.7 ms	57.5	45.8	254	110

Do wyznaczenia dokładności obliczeń wykorzystano program inicjujący, wykorzystujący bibliotekę Keras, który był zatrzymywany po wygenerowaniu wstępnych wartości wag i przesunięć. Wartości te zapisywano i eksportowano na FPGA. Następnie uruchamiano symulację na układzie FPGA. Po zakończeniu symulacji uruchamiano program referencyjny w Pythonie (wersję nie korzystającą z Keras) z pominięciem etapu inicjacji wag, lecz z wykorzystaniem zapisanych wcześniej wartości początkowych wag i przesunięć. Po zakończeniu działania porównywano wartości aktualizowanych parametrów (wag i przesunięć). Ze względu na złożoność procesu w przypadku badania dokładności, skrócono liczbę sekwencji do 20, a przedstawiona w tabeli 4.3 dokładność jest maksymalną wartością różnicy wartości parametrów pomiędzy programem referencyjnym a poszczególnymi wariantami. Maksymalne wartości błędu były na poziomie 10^{-2} dla każdego z rozważanych wariantów. Są to wartości nieco większe niż maksymalne błędy zaobserwowane dla klasyfikatora. Powstaje zatem pytanie, czy taka wartość błędu ma wpływ na proces uczenia oraz czy zmniejszenie lub zwiększenie dokładności aproksymacji funkcji aktywacji znacząco wpłynie na jakość procesu uczenia. To zagadnienie zostało poruszone w punkcie 4.3.4.

Tabela 4.3. Zestawienie dokładności obliczeń

Wariant	Błąd maksymalny
Wariant 1	$3.42 \cdot 10^{-2}$
Wariant 2	$2.75 \cdot 10^{-2}$
Wariant 3	$3.21 \cdot 10^{-2}$
Wariant 4	$2.93 \cdot 10^{-2}$

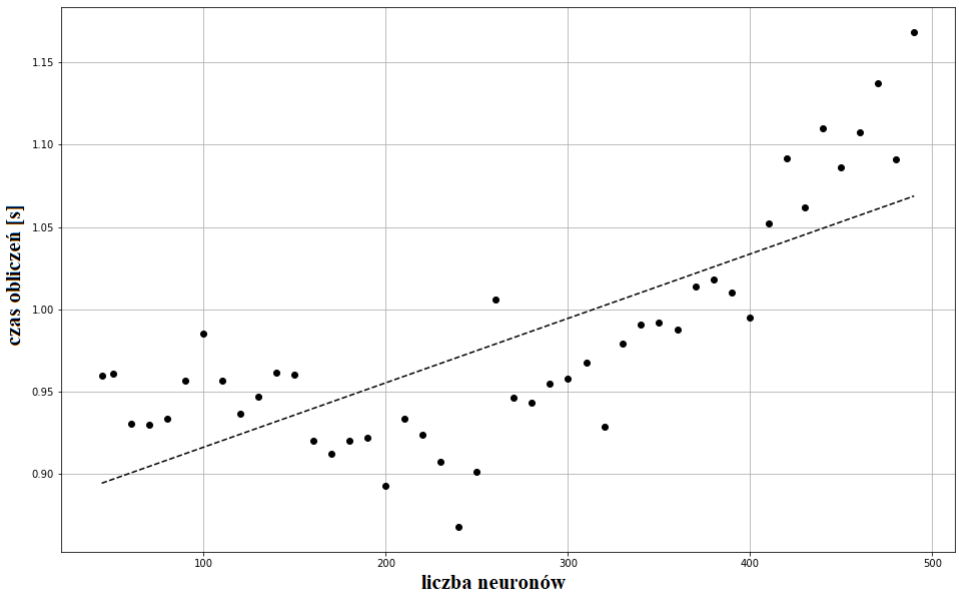
Implementacja struktury logicznej realizującej proces uczenia wymaga znacznego rozbudowania opisu sprzętu przygotowywanego za pomocą języka Verilog, a co za tym idzie zużycia zasobów sprzętowych. Dokładne wartości wykorzystania zasobów są określane w trakcie przetwarzania opisu sprzętu w programie Vivado. Tam m.in. dopasowuje się obliczane wartości do wprowadzonych ograniczeń np. dotyczących minimalnego czasów narastania sygnału zegara, które wraz z innymi ograniczeniami można zdefiniować w pliku (ang. constraints file). Program Vivado wprowadza też własne optymalizacje. Nasuwa to oczywistą trudność w oszacowaniu wykorzystania zasobów jedynie bazując na opisie sprzętu w Verilogu. Powiększenie kodu o elementy zapisujące wartości sygnałów w komórce dla każdego kroku w przód, znaczące rozbudowanie automatu sekwencyjnego, dodanie elementów zapisujących wartości gradientów oraz rozbudowanie interfejsu komórki LSTM o wyprowadzenia umożliwiające przepływ gradientów w obrębie warstwy poskutkowało kilkukrotnym zwiększeniem zużycia zasobów sprzętowych. Jak widać w tabeli 4.4, dla wariantu 3 liczba wykorzystanych bramek LUT wyniosła 1030323, przerzutników 781374 oraz 1408 DSP. Porównując z wykorzystaniem zasobów klasyfikatora z punktu 3.5, gdzie dla wariantu 3 zużycie bramek LUT wyniosło 195431, przerzutników 126647, zaś DSP 1408, można zauważyć, że zużycie tablic LUT wzrosło 5.27 razy, zaś przerzutników 6.17 razy. Zużycie bloków DPS pozostało bez zmian ze względu na współdzielenie bloków arytmetycznych. Duża część obecnie dostępnych układów FPGA radzi sobie z pomieszczeniem tak dużych struktur logicznych, pozostawiając jeszcze miejsce na rozbudowę. Dostępne zasoby dla rozważanego układu tj. XCU250-FIGD2104-2L-E, umieszczone zostały jako ostatni wiersz w tabeli 4.4. Widać zatem, że w przypadku wariantu 3 wykorzystano 59% dostępnych bramek LUT, 23% przerzutników i 11% układów DSP. Duże układy FPGA będące w stanie pomieścić prezentowane struktury logiczne są stosunkowo drogie, co bezpośrednio rzutuje na koszt takiej implementacji, co zostało szerzej przedstawione w punkcie 4.5. Najwięcej tablic LUT zużywa wariant 1, jednocześnie zużywając najmniej przerzutników FF. Niestety tablic LUT w rozważanym układzie jest znacząco mniej, przez co stanowią bardziej istotny czynnik przy analizie zużycia zasobów.

Tabela 4.4. Porównanie wykorzystania zasobów sprzętowych (XCU250-FIGD2104-2L-E)

Wariant	LUT	FF	DSP
Wariant 1	1121285	738393	1408
Wariant 2	1044745	791006	1408
Wariant 3	1030323	781374	1408
Wariant 4	1077780	782102	1408
Dostępne	1728000	3456000	12288

4.3.3. Czas obliczeń dla różnej liczby komórek LSTM

Na rys. 4.46 przedstawiono jak zmienia się czas obliczeń wykonywanych z wykorzystaniem GPU, dla różnej liczby komórek w warstwie LSTM, przy zachowaniu struktury sieci. W trakcie eksperymentu dane zostały zgrupowane w jeden pakiet, przez co aktualizacja wag miała miejsce jeden raz, na końcu, po uwzględnieniu wpływu każdego z 176 przebiegów, cykl uczenia następnie kończono. Proces uczenia odpowiada więc temu, wykorzystywanemu w trakcie testów, w punkcie 4.3.2. Każdy punkt na rys. 4.46 odpowiada jednemu cyklowi uczenia.



Rysunek 4.46: Czas obliczeń z użyciem GPU dla różnej liczby neuronów w sieci

Zaobserwować można, że czas obliczeń charakteryzuje się dużą losowością i nie zmienia się w jednostajny sposób. Widoczna jest natomiast tendencja wzrostowa i czasy obliczeń dla liczby neuronów zbliżającej się do 500 są już około 1.3 razy większe niż na początku obliczeń. W celu lepszego zobrazowania tendencji wyznaczono linię trendu wykorzystując regresję liniową. Nachylenie linii trendu to 39.112 ms na 100 neuronów.

W przypadku FPGA wzrost liczby neuronów w sieci wpływa jedynie na rozbudowanie sumatorów (blok oznaczony jako Σ na rysunku 3.36). W sumatorze, w trakcie przejścia w tył odbywają się również obliczenia związane z równaniami (4.35) - (4.39). Wzrost czasu obliczeń w przypadku FPGA nie charakteryzuje się losowością. Dodanie dwóch komórek LSTM w warstwie sprawia, że w czasie propagacji w przód liczba stanów w automacie sekwencyjnym, zwiększa się o 1. Analogicznie w przypadku fazy aktualizacji wag. W przypadku przejścia w tył liczba stanów zwiększa się o 3. Wzrost liczby komórek ma też swój efekt w warstwie wyjściowej. Nieco inna logika jest realizowana w przypadku komórki wyjściowej, przy przejściu w przód dołożenie 8 komórek skutkuje wydłużeniem obliczeń o 1 stan, analogicznie w tył oraz przy aktualizacji wag.

W przypadku wariantów 2-4, bloki arytmetyczne wykonują operacje w 2 cykle zegara, daje to łącznie 41850 cykli przy dodaniu 100 komórek w założonym scenariuszu. Przeliczenie liczby cykli na czas, z użyciem częstotliwości zegara z tabeli 4.1, nie byłoby uzasadnione, ponieważ zmiana liczby komórek LSTM wiąże się z innym wykorzystaniem zasobów i innym rozplanowaniem w obrębie układu, co przekłada się na maksymalną częstotliwość taktowania. Natomiast wykonując takie obliczenie w celu poglądowym, wykorzystując częstotliwość dla wariantu 3, otrzymuje się przyrost 1.231 ms na 100 neuronów. Żeby zrównać się z przyrostem dla GPU maksymalna częstotliwość taktowania musiałaby zmaleć 31.8 raza. Wydaje się to niemożliwe, biorąc pod uwagę wyniki osiągnięte w punkcie 3 i 4, w których rozmiar sieci urósł kilkukrotnie, a maksymalna częstotliwość taktowania zmalała około 2 razy dla wariantu 3. Nawet gdyby przyrosty się zrównały, to wciąż łatwo zauważalne jest, że w wielu przypadkach wykonanie obliczeń za pośrednictwem FPGA ma sens i może okazać się bardzo wydajne.

4.3.4. Wpływ dokładności aproksymacji na proces uczenia

Wpływ dokładności na proces uczenia przeanalizowano z wykorzystaniem funkcji aktywacji opartej na CORDIC poprzez zmianę parametru metody w zakresie od 15 do 31 (dokładny opis funkcji aktywacji znajduje się w punkcie 2.2) dla zakresu dziedziny <-6,6> oraz dodatkowo dla <-10,10>. Ze względu na to, że w trakcie pełnego

procesu uczenia dobre wyniki osiąga się przy zastosowaniu dodatkowych technik, które wykraczają poza zakres implementacji na FPGA, analizę wykonano na CPU, w Pythonie wykorzystując kod napisany od podstaw - bez wykorzystania biblioteki Keras, ani innych przeznaczonych do uczenia maszynowego. Analiza ma na celu zbadanie potencjalnego wpływu dokładności obliczeń w module FPGA na proces uczenia, przy założeniu, że układ FPGA współpracowałby z PC lub mikrokontrolerem. W eksperymencie uwzględniono elementy odpowiedzialne za inicjalizację, przetasowywanie danych oraz algorytm optymalizacyjny Adam (analogicznie do punktu 3.5.3 gdzie uczono sieć do zadania klasyfikacji). Kod odpowiedzialny za inicjalizację oraz przetasowywanie danych uczących wykorzystują rachunek prawdopodobieństwa oraz algorytmy, których dopracowane implementacje są dostępne w bibliotekach standardowych języków C++ lub właśnie Pythona. Inicjalizacja wag w przypadku implementacji użytej w eksperymencie opierała się o rozkład normalny ze średnią równą 0 oraz odchyleniem standardowym równym 0.1. Inicjalizacja przesunięć w pakiecie Keras odbywała się poprzez przypisanie wartości 0 do większości przesunięć poza przesunięciami bramek zapominających - te inicjowane były wartością 1. Analogicznie postąpiono w implementacji przeznaczonej do eksperymentu. Kolejnym elementem obliczeń, który został wykorzystany w niniejszym eksperymencie, ale nie był rozważany na FPGA, był algorytm optymalizacyjny Adam. Główne równanie opisujące optymalizator Adam jest przedstawione jako (4.43) [125].

$$\theta_{n+1} = \theta_n - \frac{\alpha}{\sqrt{\hat{v_n} + \epsilon}} \cdot \hat{m_n} \tag{4.43}$$

gdzie: θ - optymalizowany parametr; α - współczynnik uczenia; ϵ - bardzo mała wartość stała, zabezpieczająca przed dzieleniem przez 0; m - pierwszy moment; v - drugi moment.

Momenty m_n oraz v_n oblicza się według (4.44) oraz (4.45). Parametry $\hat{m_n}$ oraz $\hat{v_n}$ oznaczają wartości momentów po korekcji. Korekcję wykonuje się odpowiednio według wzorów (4.46) oraz (4.47), zaś wartości początkowe m_0 oraz v_0 przyjmuje się jako równe 0. W omawianych wzorach g oznacza gradient, a β_1 i β_2 to parametry algorytmu.

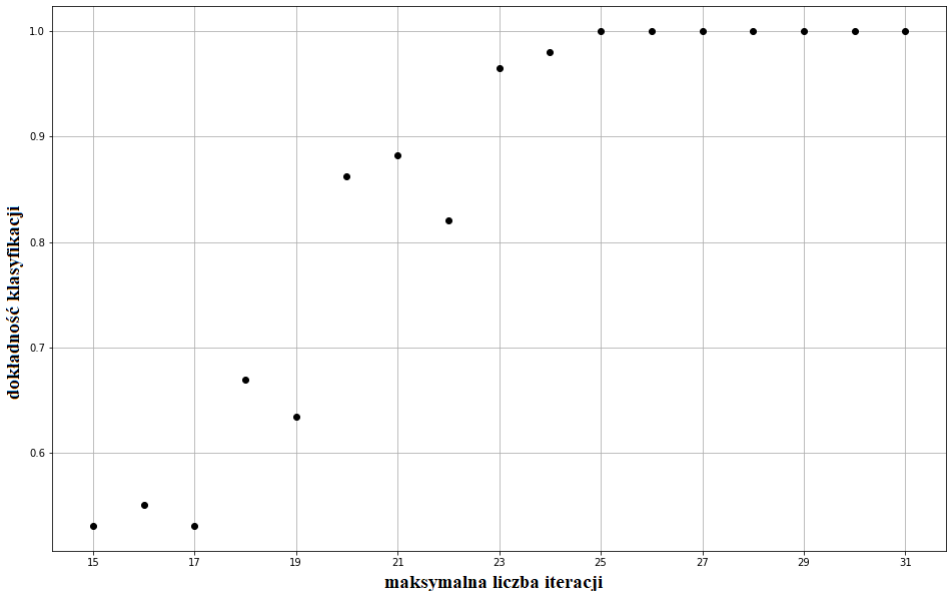
$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{4.44}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{4.45}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4.46}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4.47}$$

Algorytm Adam wymaga zatem przetrzymywania w pamięci wartości momentów dla każdej trenowanej zmiennej, zawiera dzielenie, pierwiastkowanie oraz potęgowanie. Może zatem być kosztowny sprzętowo w przypadku implementacji na FPGA.



Rysunek 4.47: Dokładność klasyfikacji po uczeniu dla różnych dokładności aproksymacji funkcji aktywacji

W trakcie epoki obliczeniowej, dane najpierw przetasowywano, a następnie grupowano w pakiety po 32 szeregi. Następnie wykonywane były obliczenia związane z przejściem w przód i w tył. Gradienty w obrębie pakietów były sumowane. Po wykonaniu obliczeń w obrębie pakietu wykonywano aktualizację wag z użyciem algorytmu Adam. Wykonanie obliczeń dla wszystkich danych wejściowych kończyło epokę obliczeniową. W omawianym eksperymencie liczbę epok ustalono na 15, a na jeden punkt na rys. 4.47

składało się 20 cykli uczących spośród których wybierano najwyższą wartość dokładności klasyfikacji. Samą klasyfikację wykonywano na danych testowych składających się z 88 szeregów. Dane wykorzystywane w eksperymencie to te same, które opisywano w punkcie 3.5.2. Jak widać na rys. 4.47 dokładność aproksymacji wpływa na osiąganą dokładność uczenia. Ze względu na ograniczenie obliczeń, w sensie liczby epok oraz prób dla konkretnej wartości dokładności, wykres prezentuje tendencję do łatwiejszego osiągnięcia zadowalających efektów uczenia w miarę wzrostu dokładności aproksymacji. Jest prawdopodobne osiągnięcie lepszej dokładności klasyfikacji dla poszczególnych punktów, natomiast widoczny trend obrazuje, że przy małej dokładności aproksymacji trudniej uzyskać dobry efekt uczenia sieci. W opisywanym eksperymencie, dla zakresu $< -6,6 >$, zadowalające wyniki osiąga się od $i = 23$, a 100% dokładność klasyfikacji na danych testowych można osiągnąć już od $i = 25$. Odnosząc liczbę iteracji do osiąganej dokładności według rys. 3.20 oraz 3.21, odpowiada to dokładności aproksymacji na poziomie 10^{-5}(dla $i = 23$) -10^{-6}(dla $i = 25$). Bardzo podobne wyniki, różniące się nieznacznie ze względu na losową inicjację parametrów i pozwalające na dojście do tych samych wniosków, osiągnięto dla zakresu $< -10,10 >$. Odnosząc te wyniki do podjętych w punkcie 3.2 założeń, można zauważyć, że założenie dokładności na poziomie 10^{-7} dla zakresu $< -6,6 >$ było zauważalne z punktu widzenia procesu uczenia, natomiast było bardzo dobrym punktem wyjścia do eksperymentów i rozważań.

4.4. Implementacja na platformie ZYNQ

W niniejszym puncie przedstawiono inne podejście do wykorzystania logiki programowalnej do zadania uczenia sieci z warstwą LSTM. Podobnie jak w w punkcie 3.6 wykorzystano w tym celu układ z serii ZYNQ zamontowany na płytce ZYBO Z7-20 tj. Xilinx XC7Z020-1CLG400C.

4.4.1. Sposób implementacji

W celu wykorzystania logiki programowalnej dostępnej na platformie ZYNQ do procesu uczenia sieci LSTM, wykonano trzy warianty implementacyjne, które opierały się na algorytmie propagacji wstecznej w czasie dla sieci LSTM i różniły się sposobem aktualizacji wag i przesunięć oraz miejscem wykonywania obliczeń. Pierwszy wariant podobnie jak w punkcie 4.3, aktualizował parametry sieci według algorytmu

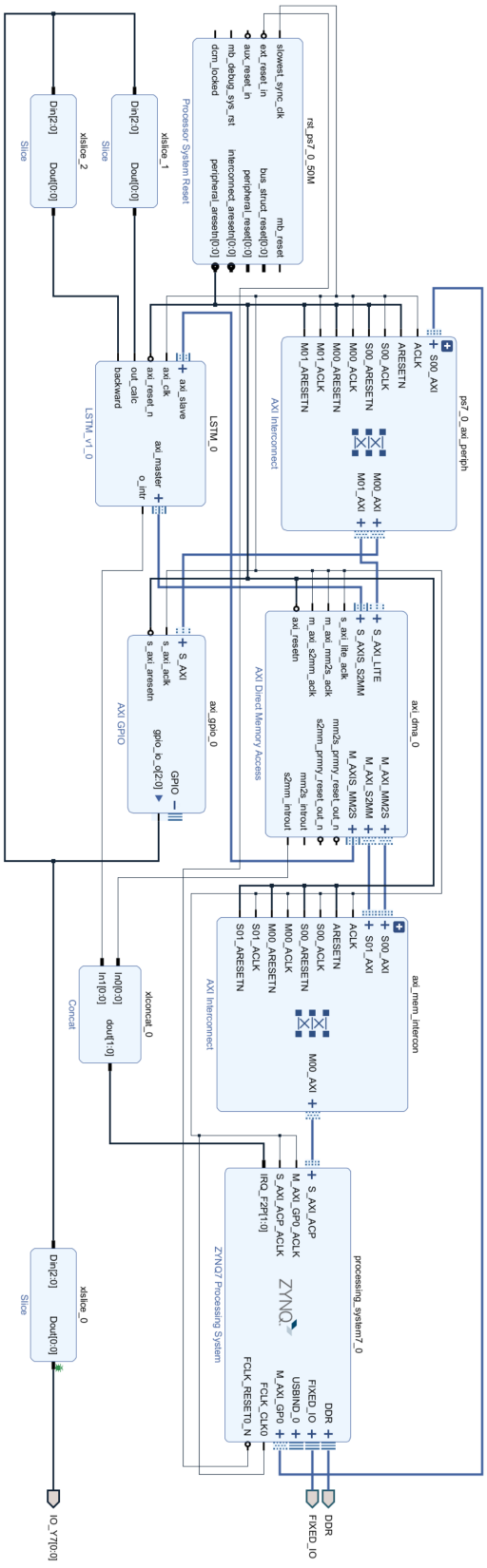
SGD (zgodnie z (4.40)) i został przygotowany na potrzeby pomiaru czasu obliczeń.

Drugi wariant wykonywał obliczenia w pełni po stronie procesora i został zaimplementowany w celach referencyjnych. Wariant ten wykorzystywał PL jedynie do wystawiania sygnału wykorzystywanego do pomiaru czasu obliczeń, a schemat połączeń po stronie PL wyglądał tak samo, jak na w punkcie 3.6.2 (rysunek 3.41).

Trzeci wariant wykorzystywał w swoich obliczeniach PL (w takiej samej wersji jak w wariancie pierwszym), ale ze znacznie rozszerzoną funkcjonalnością po stronie PS, która zawierała algorytm Adam (zgodnie z 4.43), inny sposób inicjowania wag, przetasowywanie danych w trakcie uczenia oraz grupowanie w pakiety. Zaimplementowany program był odzwierciedleniem zaimplementowanego w Pythonie, w punkcie 4.3.4. Ten wariant został wykorzystany został do sprawdzenia wydajności procesu uczenia przy wykorzystaniu układu ZYNQ i prezentowanej logiki. Inicjowanie wag odbywało się na początku cyklu uczenia, zaraz po konfiguracji i zainicjowaniu interfejsów. Wagi były obliczane zgodnie z (4.48), gdzie wykorzystana została funkcja $rand()$ z biblioteki *stdlib.h*, która zwraca pseudo-losowe liczby z zakresu od 0 do $RAND\_MAX$. Zwracana liczba najpierw była rzutowana na typ $float$, a następnie przeskalowana do zakresu -0.1 do 0.1. Przesunięcia były w większości inicjowane wartością 0. Jedynie przesunięcia dla bramki zapominającej były inicjowane wartością 1. Taka inicjalizacja jest podobna do mającej miejsce w bibliotece Keras.

$$W_{xy} = -0.1 + \frac{0.2 * ((float)rand())}{RAND\_MAX} \tag{4.48}$$

W przypadku normalnej pracy klasyfikatora, w każdym z omawianych w niniejszym punkcie podejść, program na PS odpowiadał opisowi podejścia pierwszego z punktu 3.6. Pominięty został etap inicjowania wag. Logika zaimplementowana na PL działała również analogicznie do przedstawionej w punkcie 3.6. Jedyną różnicą po stronie PS, w porównaniu z punktem 3.6, była konieczność ustawienia jednego, dodatkowego bitu w porcie GPIO, który odpowiadał za wykonywane przez komórkę obliczenia związane z przejściem w przód oraz z przejściem w tył. Dodatkowe wyprowadzenie (o nazwie *backward*) i dodatkowy bit wiązały się z dołożeniem kolejnego bloku Slice, widocznym na rysunku 4.48. W porównaniu z konfiguracją z rysunku 3.40 można zauważyć, że jest to jedyna zmiana poza samym blokiem realizującym obliczenia.



Rysunek 4.48: Schemat wykorzystanej konfiguracji połączeń pomiędzy PL i PS w układzie typu ZYNQ dla BPTT

Blok realizujący obliczenia został znacząco rozbudowany w stosunku do wersji z punktu 3.6. Funkcjonalność klasyfikatora została rozszerzona, przy zachowaniu kompatybilności z poprzednim rozwiązaniem, przez co ramka danych opisana w punkcie 3.6 jest wciąż obsługiwana, przy ustawieniu portu *backward* w stan niski. Ze względu na to, że w algorytmie BPTT dla LSTM przedstawionym w równaniach (4.29)-(4.36) występują wartości sygnałów z wnętrza komórki, z konkretnych chwil czasowych w trakcie przejścia w przód, wymagane jest przesyłanie tych wartości do PS i zapisywanie ich. Zwracany z PL strumień danych ma wówczas postać:

$$[ht, ct], \quad [f, i], \quad [c, o],$$

gdzie: ht wartość wyjściowa komórki; ct wartość stanu wewnętrznego komórki; f, i, c, o to wartości na wyjściu poszczególnych bramek.

Jak widać, drugie i trzecie pole danych podczas klasyfikacji można bezpiecznie zignorować, a podczas przejścia w przód, w trakcie procesu uczenia, zapisywać wartości sygnałów po stronie PS. W ramach rozbudowania funkcjonalności modułu dodano obsługę innego zestawu danych wejściowych, które wykorzystywane są podczas przejścia w tył (port *backward* w stanie wysokim).

Poza pochodnymi sygnałów przekazywanymi rekurencyjnie, w obliczeniach wykorzystywane są wspomniane wartości sygnałów z konkretnych chwil czasowych oraz wartość pochodnej stanu z poprzedniego kroku obliczeniowego dla danej komórki. Bufor danych przesyłanych w trakcie przejścia w tył wygląda zatem następująco:

$$\begin{array}{llll}
[ftp1, it], & [Ct, Ctm1], & [ot, dCtp1], & [Ckt, ft], \\
[df00, w00f], & [df01, w01f], & ..., & [df43, w43f], \\
[di00, w00i], & [di01, w01i], & ..., & [di43, w43i], \\
[dC00, w00C], & [dC01, w01C], & ..., & [dC43, w43C], \\
[do00, w00o], & [do01, w01o], & ..., & [do43, w43o], \quad [dD, \text{"1"}].
\end{array}$$

gdzie: $ftp1$ wartość sygnału zapominającego z kroku $t+1$; it, Ckt, ot, ft wartości sygnałów na wyjściu bramek z kroku t; $Ctm1$ wartość stanu wewnętrznego komórki z kroku $t-1$; $dCtp1$ wartość pochodnej stanu wewnętrznego obliczonej dla kroku $t+1$; Ct wartość stanu wewnętrznego komórki z kroku t; $df00...df43$, $di00...di43$, $dC00...dC43$, $do00...do43$ wartości pochodnych sygnałów od każdej z komórek w warstwie; $w00f...w43f$, $w00i...w43i$, $w00C...w43C$, $w00o...w43o$ wartości wag dla przesyłanych pochodnych; dD błąd od warstwy następnej.

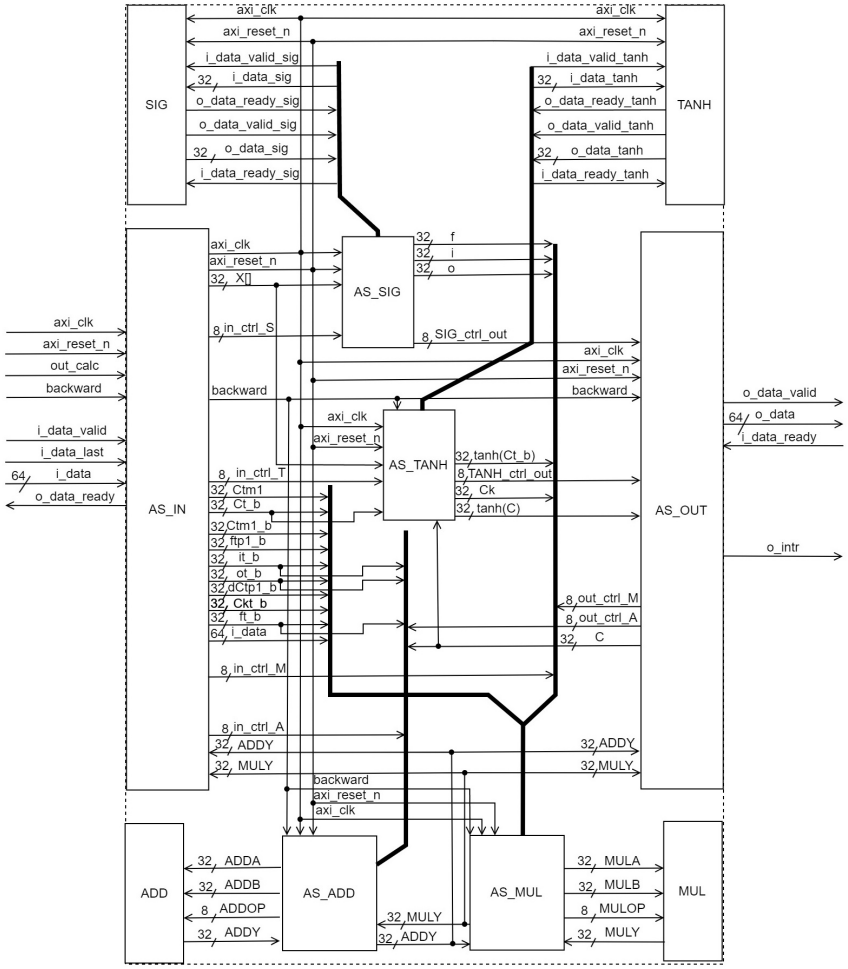
Daje to łącznie 181 pól danych, a więc nieco mniej niż podczas przejścia w przód. Parametry zwracane do PS również mają inną formę, chociaż długość strumienia danych jest taka sama i wynosi 64 bity dla pary parametrów. W przypadku przejścia w tył przesyłane są do PS 3 pola danych zawierające 5 parametrów będących pochodnymi sygnałów występujących w komórce. Bufor danych otrzymywanych przez PS wygląda zatem następująco:

$$[df, di], \quad [dCk, do], \quad [dC, "1"]$$

Wartość "1" na ostatnim miejscu ma rozmiar 32 bitów i jest jedynie dopełnieniem dla niewykorzystywanych bitów. Ze względu na postać równań wykorzystywanych w trakcie przejścia w tył, tj. równań (4.29)-(4.36), w których od samego początku wykorzystuje się wartość Δh_t, konieczne było wstrzymanie obliczeń do momentu zgromadzenia wszystkich danych wejściowych składających się na nią zgodnie z (4.36) oraz na wartość dD, która zawierała błąd od wyjścia warstwy dla konkretnej komórki, tj. Δ_t. Podobnie jak w punkcie 4.3, obliczenia wykonywane były po kolei w sposób, aby podobne działania wykonywane były tylko raz, a wartość wykorzystana w kilku miejscach, tj. obliczenie tangensa hiperbolicznego oraz iloczynu $\delta C_t \odot i_t$. Dla wariantu pierwszego sprawdzono również różne nastawy bloków arytmetycznych.

Podejścia z przeniesieniem parametrów sieci na stronę PL (analogicznie do podejścia trzeciego i czwartego z punktu 3.6) były również rozważane, natomiast zapotrzebowanie na zasoby okazało się zbyt duże i nie mieściłyby się na wykorzystywanym układzie ZYNQ, co program Vivado sygnalizował błędem syntezy. Ze względu na brak dostępu do większego układu te podejścia zostały pominięte w pomiarach i w dalszym opisie.

Na rysunku 4.49 przedstawiono opisywany schemat struktury logicznej sieci LSTM z możliwością uczenia i z parametrami po stronie PS, wykorzystywanej zarówno w przypadku wariantu pierwszego jak i trzeciego przedstawionych w tym punkcie. Schemat jest rozwinięciem z rysunku 3.39. Zauważalną różnicą jest występowanie większej liczby sygnałów związanych z obliczaniem poszczególnych pochodnych. Liczba bloków jest taka sama jak poprzednio. Główna rozbudowa dotyczyła automatów sekwencyjnych AS_IN oraz AS_OUT, w których to miały miejsce dodatkowe operacje. Ze względu na przetwarzanie strumieniowe danych wchodzących na wejście modułu, które na etapie zliczania fragmentów strumienia danych jest takie samo jak w przy-



Rysunek 4.49: Struktura logiczna sieci LSTM zaimplementowanej na układzie ZYNQ z możliwością uczenia i z parametrami po stronie PS

padku klasyfikatora, można było wykorzystać niemal w całości kod modułu, dodając tylko elementy związane z zapisaniem i udostępnieniem wewnątrz komórki parametrów niezbędnych w obliczeniach.

Walidacja przedstawianej logiki odbywała się w sposób analogiczny do przedstawionego w punkcie 3.6 dla wariantu pierwszego. Poza wykonaniem kroków dla przejścia w przód, dodatkowym elementem było sprawdzenie działania logiki podczas przejścia w tył. Obliczanie błędu dla neuronu wyjściowego oraz aktualizacja wag znajdowała się po stronie PS, walidacja odbyła się tamże poprzez porównanie wyników z programem referencyjnym napisanym w Pythonie. Moduł logiki odpowiadał głównie za obliczenia wewnątrz komórki, gdzie wykorzystywany był jednokrotnie blok wykonujący obliczenia funkcji aktywacji, w związku z tym przystąpiono do weryfikacji działania logiki w całości poprzez początkowe zasymulowanie przejścia w tył dla 100

zestawów danych testowych i porównanie wyników z programem referencyjnym. Następnie prześledzono i przeanalizowano zarejestrowane transakcje pomiędzy PS i PL, i również porównano ich porównania.

4.4.2. Pomiary i zestawienie wyników

W celu wykonania pomiarów czasu obliczeń wykorzystano ten sam obwód pomiarowy, jak w punkcie 3.6.3, tj. oscyloskop RIGOL DS1022C, który został podłączony bezpośrednio do wyprowadzeń JB płytki ewaluacyjnej, do portu $Y7$. Pomiar czasu obliczeń dla algorytmu BPTT, na podstawie wariantu pierwszego implementacji, odbywał się według scenariusza, jak w punkcie 4.3 tj. pakiet danych uczących składał się z 176 sekwencji. Następnie sekwencja po sekwencji, wykonywano przejście w przód, w trakcie którego zbierane i zapisywane były wartości sygnałów z wnętrza komórki w każdej chwili czasowej.

Tabela 4.5. Implementacja BPTT na platformie ZYNQ (XC7Z020-1CLG400C)

Liczba taktów zegara	Miejsce wykonywania obliczeń	LUT	FF	DSP	Częstotliwość taktowania [MHz]	Czas obliczeń [s]
0	PS+PL	7706	8620	12	46.4	46.8
1	PS+PL	7546	8975	12	49.9	48.6
2	PS+PL	7378	9187	12	81.4	48.0
3	PS+PL	7355	9234	12	102.7	47.6
4	PS+PL	7416	9367	12	107.0	48.4
-	PS	426	543	0	132.3	4.26

Dla każdego przejścia w przód wykonywane było przejście w tył i obliczany był błąd dla każdego parametru sieci. Po przeprocedowaniu wszystkich sekwencji jednokrotnie wykonywano aktualizację wag zgodnie z SGD. Sygnał pomiarowy na wyprowadzeniu $Y7$ ustawiany był w stan wysoki przed rozpoczęciem procedowania pierwszej sekwencji, a następnie ustawiany w stan niski po wykonaniu aktualizacji wag. W trakcie pomiarów zmieniano nastawy bloków arytmetycznych w strukturze PL od 0 cykli do 4. Wykonano również pomiar czasu obliczeń dla przypadku wykonywania obliczeń w całości programowo. Czas obliczeń był mierzony w taki sam sposób, niezależnie od wariantu.

Tabela 4.6. Zestawienie czasów obliczeń BPTT

Wariant	Czas obliczeń	$\frac{T_{GPU1}}{T_x}$	$\frac{T_{GPU2}}{T_x}$	$\frac{T_{PC}}{T_x}$	$\frac{T_{RPi3}}{T_x}$
Wind+Keras+GPU1	960 ms	1.00	0.80	4.43	1.91
Ubun+Keras+GPU2	765 ms	1.25	1.00	5.55	2.39
Python+CPU	4250 ms	0.23	0.18	1.00	0.43
Raspberry Pi 3	1830 ms	0.52	0.42	2.32	1.00
Wariant 1	34.1 ms	28.2	22.4	125	53.7
Wariant 2	15.8 ms	60.8	48.4	269	116
Wariant 3	15.8 ms	60.8	48.4	269	116
Wariant 4	16.7 ms	57.5	45.8	254	110
ZYNQ PL+PS	46.8 s	0.021	0.016	0.091	0.039
ZYNQ PS (ARM)	4.26 s	0.225	0.179	0.998	0.429

Wyniki dla implementacji sieci LSTM z możliwością uczenia przedstawione są w tabeli 4.5. Można zauważyć, że implementacje z wykorzystaniem platformy ZYNQ wykorzystują bardzo małą liczbę zasobów sprzętowych. Są też zauważalnie wolniejsze niż te same obliczenia wykonywane z wykorzystaniem samego procesora (samej części PS), na którym program napisany był w języku C, bez użycia żadnego systemu operacyjnego oraz wykorzystywał jeden rdzeń procesora. Porównując wyniki z otrzymanymi podczas pomiaru klasyfikatora, można zauważyć, że częstotliwości taktowania w przypadku klasyfikatora są mniejsze, pomimo że struktura logiczna jest mniejsza. Może być to wynik strategii wykorzystywania zasobów i sposobu optymalizacji struktury logicznej w trakcie syntezy, w programie Vivado lub efekt uzyskania np. dłuższej ścieżki krytycznej. Podobnie jak w przypadku klasyfikatora, w przypadku implementacji z możliwością uczenia, zmiana nastaw bloków arytmetycznych nie wpłynęła znacząco na czas obliczeń, natomiast istotnie na częstotliwość taktowania. Różnica w zużyciu zasobów sprzętowych pomiędzy nastawą bloków arytmetycznych na 0 oraz nastawą na 4 jest obserwowalna, natomiast stosunkowo nieduża: spadek zużycia tablic LUT o 3.9%, a w przypadku przerzutników FF wzrost o 8.7%.

W tabeli 4.6. zestawiono wyniki z pozostałych podejść opisywanych w pracy oraz z obliczeń wykonywanych za pomocą programów referencyjnych. Do porównania czasów obliczeń wykorzystano najszybszą implementację wariantu pierwszego. W porównaniu z pozostałymi implementacjami, realizacje w oparciu o układ ZYNQ z małymi zasobami w strukturze logicznej wypadają najwolniej. Wykonanie obliczeń w sa-

mym procesorze ARM, będącym częścią układu ZYNQ jest porównywalne z wykonaniem obliczeń napisanych w Pythonie na procesorze komputera. Jest też zauważalnie wolniejsze niż analogiczne rozwiązanie wykonywane na Raspberry Pi 3, co jest spowodowane tym, że w programie pozostawiono ten sam sposób zarządzania parametrami i pamięcią, jaki był konieczny podczas eksperymentów z implementacją korzystającą z PL, a który nie jest zoptymalizowany w przypadku obliczeń programowych na procesorze.

Kolejnym etapem eksperymentów było wykonanie uczenia z wykorzystaniem losowej inicjalizacji wag, przetasowywania danych i aktualizacji wag zgodnie z algorytmem Adam. Do tego celu wykorzystano trzeci wariant implementacyjny, który również korzystał z obu części tj. z PL i PS. W trakcie epoki obliczeniowej, dane najpierw były przetasowywane, a następnie grupowane po 32 szeregi na pakiet, po czym wykonywane były obliczenia związane z przejściem w przód i propagacją wsteczną. Gradienty w obrębie pakietów były sumowane, a po wykonaniu obliczeń dla każdego przebiegu w pakiecie aktualizowano wagi. Podczas eksperymentów ustalono liczbę epok na 20. Dokładność klasyfikacji dla danych testowych (88 przebiegów) była sprawdzana co epokę. Wykonano 50 prób w trakcie których osiągnięto maksymalną dokładność klasyfikacji 95.4%.

4.5. Koszty proponowanych implementacji dla różnych układów FPGA

W ramach pracy jako fizyczny układ wykorzystano płytkę ewaluacyjną ZYBO Z7-20 z układem XC7Z020-1CLG400C (opisanym w punkcie 2.5). Ponadto, podczas prac nad funkcjami aktywacji oraz komórką LSTM (punkty 3.3 i 3.4), synteza i implementacja w narzędziach projektowych wykonywana była dla układu ARTIX-7 XC7A100T-CSG384. Synteza i implementacja całej sieci LSTM zarówno dla zadania klasyfikacji, jak i dla procesu uczenia (punkty 3.5 i 4.3) wykonywana była dla układu XCVU250-FIGD2104-2L-E, będącego częścią karty ALVEO U250 Data Center Accelerator Card. Karta ALVEO U250 jest kartą akceleratora z przeznaczeniem do wspomagania obliczeń związanych z uczeniem maszynowym w centrach analiz danych, popularna w przypadku wizji komputerowej. Poza wykorzystywanymi w pracy układami, w celu zaprezentowania szerszego kontekstu, przedstawiono w niniejszym punkcie kilka dodatkowych układów FPGA dostępnych na rynku.

Najważniejsze z parametrów, z punktu widzenia niniejszej pracy, zostały przedstawione w tabeli 4.7. Jak widać układ ARTIX-7 XC7A100T-CSG384 jest niewielkim układem, który był wystarczający przy rozważaniach funkcji aktywacji, w przypadku implementacji całej sieci lub modułu wspomagającego uczenie sieci liczba zasobów sprzętowych była stanowczo niewystarczająca. Natomiast dysponuje on większą liczbą zasobów niż jest dostępna w XC7Z020-1CLG400C (układ zamontowany na płytce Zybo Z7-20), co pozwala sądzić, że byłby również wystarczający w przypadku implementacji prezentowanych w pracy dla platformy ZYNQ.

Tabela 4.7. Parametry płytek lub układów FPGA

Układ/Moduł FPGA	LUT	FF	DSP	Orientacyjna cena
ARTIX-7 XC7A100T-CSG384	63400	126800	240	1 tys. zł
ALVEO U250	1728000	3456000	12288	35 tys. zł
ZYBO Z7-20	53200	106400	220	2 tys. zł
XC7VX690T-1FFG1158CES9919	433200	866400	3600	1.7 tys. zł
XCKU035-1SFVA784	203128	406256	1700	6.3 tys. zł
XCKU5P-1SFVB784E	216960	433920	1824	8.8 tys. zł
ALVEO U200	1182240	2364480	6840	25 tys. zł
AMD ZYNQ Z-7100 Module	277400	554800	2020	10 tys. zł

Widać przy tym, że ważnym aspektem przy projektowaniu systemu opartego o FPGA, jest wybór odpowiedniego układu, który wymaga dokładnego przeanalizowania. Moduł ALVEO U250 Data Center Accelerator Card zawiera relatywnie bardzo dużą liczbę zasobów sprzętowych, które były wystarczające na każdym z etapów badań, natomiast ze względu na cenę, jego zastosowanie raczej ogranicza się do dużych centr obliczeniowych. Przy małych, doraźnych projektach byłby zdecydowanie za drogi. W celu realizacji klasyfikatora przedstawionego w punkcie 3.5, zasoby XC7VX690T-1FFG1158CES9919 powinny być wystarczające, a dla niektórych wariantów również zasoby układów XCKU035-1SFVA784 oraz XCKU5P-1SFVB784E. Przedstawiony w punkcie 4.3 moduł wspomagający uczenie sieci rekurencyjnych mógłby zostać zrealizowany za pomocą ALVEO U200, który również jest kartą przeznaczoną do centr danych, natomiast jest mniejszą i tańszą wersją. Wykorzystana w punktach 3.6 i 4.4, płyta ewaluacyjna ZYBO Z7-20, zawiera stosunkowo niedużo zasobów sprzętowych FPGA, natomiast nie jest największym układem w serii. Mała liczba zasobów

wymusiła zmianę podejścia, co poskutkowało wydłużeniem czasu obliczeń i zmniejszeniem korzyści związanych z równoległością obliczeń. Największy układ z serii ZYNQ tj. XC7Z100-2FFG900I powinien pozwolić na implementację całej sieci, jest jednak droższy. Przedstawione w tabeli ceny detaliczne są jedynie orientacyjne i mogą ulec dużej zmianie ze względu na sytuację rynku półprzewodników w ostatnich latach. Na cenę w niedalekiej przyszłości może również wpłynąć fakt dużego zainteresowania układami FPGA. Koszt zależny jest również od zamówień hurtowych, co ma miejsce w przypadku produkcji urządzeń wykorzystujące konkretne układy.

4.6. Podsumowanie

Wykorzystanie układów FPGA do procesu uczenia rekurencyjnych sieci neuronowych jest tematem rzadko poruszanym i trudno spotykanym w literaturze przedmiotu, co sprawia, że nie jest jednoznaczne z czym wiąże się takie przedsięwzięcie i czy takie podejście do zagadnienia jest efektywne lub opłacalne ekonomiczne. Osiągnięte w niniejszej pracy wyniki czasów obliczeń oraz ich zestawienie z popularnymi obecnie podejściami sugerują, że zastosowanie układów FPGA w procesie uczenia może znacząco przyspieszyć obliczenia i tym samym proces uczenia, a samo podejście daje się łatwo zaadaptować do implementacji dowolnej sieci składającej się z warstwy LSTM. Obliczenia z wykorzystaniem FPGA okazały się szybsze nawet od obliczeń z wykorzystaniem GPU, który obecnie jest szeroko stosowany w uczeniu maszynowym. W stosunku do GPU1 (NVIDIA GeForce GTX 1060) 60.8 razy, a w stosunku do GPU2 (NVIDIA GeForce RTX 3080 Ti) 48.4 razy. W odniesieniu do obliczeń z użyciem procesora i kodu napisanego w Pythonie rezultat jest nawet 269 razy szybszy. Niemniej jednak, warto zwrócić uwagę na zużycie zasobów sprzętowych, które jest wielokrotnie większe niż w przypadku implementacji klasyfikatora w punkcie 3.5. Nawet najmniej wykorzystujący zasoby wariant zajmuje około 10^6 tablic LUT, co mieści się jedynie na dużych układach FPGA, spotykanych m.in. w centrach obliczeniowych, takich jak Alveo U200 lub Alveo U250. W ostatnich latach zwiększa się zainteresowanie układami FPGA, co może sprawić, że duże układy staną się względnie tanie i łatwo dostępne, wtedy korzystanie z przedstawianego podejścia zdecydowanie stanie się ekonomicznie atrakcyjne. Kolejnym kosztem osiągnięcia tak dobrych czasów obliczeń jest złożoność i czasochłonność procesu projektowania struktury logicznej. Pomimo bardzo rozwiniętych narzędzi z ogromnymi możliwościami m.in. symulacji zachowania się

układu, czas wykonywania obliczeń w pakiecie Vivado związanych z syntezą, analizami czasowymi, optymalizacją struktury, dla tak dużej struktury logicznej jest znaczny. Wprowadzanie zmian, to często kilkudziesięciominutowy proces przebudowy, dlatego też zmiany należy wprowadzać z rozwagą.

Istotnym zagadnieniem w trakcie realizowania tego typu projektów jest dokładność obliczeń. Jest to istotne, gdyż bezpośrednio przekłada się na wykorzystywane rozwiązania, w tym na arytmetykę oraz implementowane algorytmy aproksymacyjne. Dokładność aproksymacji bezpośrednio przekłada się z kolei na długość obliczeń funkcji aktywacji, a w rezultacie na czas obliczeń całej sieci. W rozważaniach przedstawiono wpływ dokładności aproksymacji na proces uczenia. Zaprezentowane wyniki pozwalają zbudować pogląd na temat tego, jak ważna jest dokładność odwzorowania funkcji aktywacji na FPGA. Według eksperymentu wykonanego na CPU z użyciem Pythona, w przypadku chęci pełnego odwzorowania działania sieci oraz procesu uczenia, należy zapewnić dokładność na poziomie 10^{-6}. W oparciu o zaprezentowane wyniki można podjąć wstępne decyzje projektowe dotyczące m.in. zmiany reprezentacji liczbowej lub skrócenie długości słowa.

Porównując rozważane warianty implementacyjne na FPGA, nieco zaskakujące jest, że wariant 4, który wypadał lepiej przy okazji implementacji funkcji aktywacji, pojedynczej komórki lub nawet całej sieci realizującej zadanie klasyfikacji, w przypadku wykorzystania go w logice realizującej proces uczenia, wypada gorzej pod względem czasu obliczeń. Zauważalnie większe jest zużycie zasobów dla wariantu 4 w porównaniu z wariantem 2 i 3, co prawdopodobnie jest przyczyną tak znaczącego zmniejszenia maksymalnej możliwej częstotliwości taktowania. Pomimo mniejszej liczby cykli, poskutkowało to dłuższym czasem wykonywania obliczeń.

Wykorzystanie płytki ZYBO Z7-20 w zadaniu uczenia sieci okazało się mało atrakcyjne ze względu na bardzo długi czas obliczeń. Warto mieć na uwadze, że układ jaki został użyty w eksperymentach nie był największym układem z serii. Implementacja pojedynczej komórki z parametrami po stronie PS była nawet wolniejsza od obliczeń wykonywanych na samym procesorze ARM, wbudowanym w układ ZYNQ. Mała ilość zasobów sprzętowych układu nie pozwoliła na implementację pozostałych podejść, jakie rozważano w przypadku klasyfikacji, tj. podejścia z przeniesieniem parametrów na stronę PL oraz wykorzystanie narzędzi HLS, które pomimo poprawności opisu sprzętu sprawiały, że proces syntezy kończył się błędem.

Odnosząc otrzymane wyniki do zaprezentowanego zestawienia cen i ilości dostępnych zasobów, w dostępnych na rynku układach, przedstawionego w punkcie 4.5, można zauważyć, że koszt finansowy układów FPGA, które mogą zostać wykorzystane do implementacji pełnej warstwy LSTM do procesu uczenia sieci, jest zauważalnie większy, niż w przypadku zadania klasyfikacji, a wykorzystanie małych i względnie tanich układów skutkuje znacznym wzrostem czasu obliczeń.

5. Podsumowanie i wnioski końcowe

Wykorzystanie układów FPGA do obliczeń związanych z sieciami neuronowymi otwiera wiele nowych możliwości projektowych. Zdolność do równoległego wykonywania obliczeń, dopasowanie struktury logicznej do rozwiązywanego problemu oraz możliwość przechowywania części danych w strukturze logicznej sprawia, że w niektórych przypadkach układy FPGA wydają się być najbardziej odpowiednim wyborem. Szczególnie widoczne jest to w przypadkach, gdy obliczenia związane z sieciami neuronowymi muszą być wykonane szybko, bez dostępu do GPU lub chmury, w trudnych warunkach, w przypadku ograniczenia wykorzystywanej energii lub uniemożliwiających wydajne chłodzenie, wymagane w przypadku pracy z użyciem GPU [126, 127]. Układami pokrewnymi do FPGA są układy typu ZYNQ, które posiadają w swojej strukturze część FPGA, ale poza nią są wyposażone w procesor (jedno lub dwurdzeniowy) oraz dodatkowe moduły m.in. interfejsy komunikacyjne lub ADC. Obie części oznaczane jako PS i PL są wewnętrznie wielorako połączone i mogą wspólnie korzystać z interfejsów oraz wyprowadzeń. Do transmisji pomiędzy PS i PL można wykorzystać kilka rodzajów portów lub wykorzystać w tym celu pamięć i moduł DMA. Elementem nadrzędnym w stosunku do logiki programowalnej jest procesor i to on decyduje o inicjalizacji po załączeniu napięcia. Do synchronizacji prac pomiędzy PS i PL można wykorzystać mechanizm przerwań, co sprawia, że procesor nie musi czekać, cyklicznie sprawdzając stan obliczeń, tylko może wykonywać inne procesy w czasie działania PL. Ostatnio przedstawiono również serię układów Versal będącą rozwinięciem koncepcji towarzyszącej projektowaniu układów ZYNQ. Układy Versal są dodatkowo wyposażone w programowalne silniki obliczeniowe oraz dodatkowy procesor pełniący funkcję RPU, co ma na celu jeszcze bardziej poszerzyć możliwości oraz dopasować do obliczeń wykonywanych przy uczeniu maszynowym.

Dostępne obecnie narzędzia do projektowania układów logicznych, takie jak Vivado, pozwalają na projektowanie bardzo dużych układów logicznych, ich symulowanie oraz analizę pracy pod kątem czasu i wartości sygnałów wewnątrz struktury w trakcie działania. W przypadku układów ZYNQ i potrzeby napisania programu na procesor znajdujący się wewnątrz układu, należy skorzystać z innych narzędzi. Producenci w tym zakresie sugerują wykorzystanie programu Vitis, który poza edycją kodu, pozwala na kompilację programu i wgrywanie go, wraz ze strukturą logiczną, bezpo-

średnio do układu. Pozwala też na komunikację poprzez wbudowany terminal oraz debugowanie. Program Vitis posiada również wersję HLS, która jest w istocie nowszą wersją programu Vivado HLS. Pozwala on na projektowanie struktury logicznej poprzez opis funkcjonalności w języku C lub C++, a następnie wykorzystanie narzędzi generujących opis struktury logicznej, który można użyć dalej w programie Vivado. Konfiguracja procesu generowania odbywa się poprzez dyrektywy *#pragma*, które pozwalają na doprecyzowanie m.in. jakie interfejsy mają zostać wykorzystane, albo jaka strategia powinna zostać obrana przy procesowaniu występujących w kodzie pętli.

W pracy przedstawiono wykorzystanie układów FPGA i ZYNQ do celów realizacji sieci LSTM przeznaczonej do detekcji zużycia narzędzia w procesie kucia na zimno. Prezentowane podejścia są łatwe do zaadaptowania do implementacji dowolnej sieci składającej się z warstwy LSTM. Zagadnienie wykorzystywania logiki programowalnej do obliczeń związanych z sieciami neuronowymi jest obecnie bardzo popularne, natomiast w dużo mniejszym zakresie dotyczy sieci rekurencyjnych, co zostało pokazane w punktach 3.5.1 i 3.6.1. Tamże zaprezentowano dostępną literaturę w zakresie wykorzystania układów FPGA oraz układów ZYNQ do implementacji sieci LSTM. W jeszcze mniejszym zakresie obecna literatura przedmiotu porusza zagadnienie procesu uczenia sieci rekurencyjnych z wykorzystaniem układów ZYNQ i FPGA, co zostało szerzej przedstawione w punkcie 4.1. Sprawia to, że temat jest wciąż mało zbadany, zwłaszcza w zakresie odniesienia wyników do innych platform. Szczególnie jest to widoczne w przypadku układów ZYNQ, gdzie kwestia porównania z innymi rozwiązaniami angażującymi sprzętowe rozwiązania wciąż pozostaje nierozpoznana.

Podczas wstępnych rozważań, pierwszym zidentyfikowanym problemem okazała się być implementacja funkcji aktywacji. Poza analizą literatury w tym temacie, w punkcie 3.3.1, w niniejszej pracy zaprezentowano trzy różne podejścia: wzorowane na CORDIC, z użyciem wielomianów Czebyszewa oraz z użyciem klasycznych wielomianów. Ostatnie podejście zaprezentowano w dwóch różnych wariantach, tj. z małą liczbą przedziałów (15) oraz wyższym stopniem wielomianu (4), oraz drugi z większą liczbą przedziałów (186) i niższym stopniem wielomianu (2). Wyniki implementacji przedstawiono w tabeli 3.13, a wyniki prezentowane w cytowanej literaturze zgrupowano w tabeli 3.14. Na podstawie eksperymentów można wnioskować, że implementacja funkcji aktywacji jako bezpośredniej aproksymacji funkcji jest szybsza i zajmuje mniej zasobów sprzętowych, niżeli np. implementacja wyliczająca najpierw wartość

eksponenty i korzystająca z formuł matematycznych. Przykładem tego jest wariant pierwszy, wzorowany na CORDIC, który w przypadku funkcji tangensa hiperbolicznego wykorzystuje 1305 LUT, 262 FF oraz 4 DSP, obliczenia wykonuje w 67-80 taktów zegara przy maksymalnej częstotliwości zegara wynoszącej $33MHz$, podczas gdy wykorzystanie klasycznych wielomianów pozwala na obliczenie wyniku w 28 taktów zegara, przy $121.6MHz$ oraz wykorzystuje 672 LUT, 339 FF oraz 4 DSP.

Wykorzystując przygotowane warianty funkcji aktywacji podczas implementacji klasyfikatora, można było początkowo zaobserwować, że zastosowanie większej liczby przedziałów i niższego stopnia wielomianu wypada lepiej niż drugi wariant tej samej metody, co widać w tabeli 3.21. W przypadku bardzo rozbudowanej struktury logicznej tj. podczas implementacji algorytmu BPTT, zauważalnie lepsze wyniki otrzymano natomiast przy większym stopniu wielomianu i mniejszej liczbie przedziałów, co można wywnioskować po wynikach z tabeli 4.2. Struktura okazała się być mniejsza, co pozwalało na taktowanie układu wyższą częstotliwością zegara, a obliczenia były wykonywane szybciej.

Pomimo, że implementacja wzorowana na CORDIC wypadała gorzej, to dysponowała dużą zaletą w postaci możliwości prostego wpływania na osiąganą dokładność, tj. poprzez zmianę liczby iteracji algorytmu, przez co umożliwiła w dalszych etapach analizę wpływu dokładności aproksymacji funkcji podczas klasyfikacji oraz podczas procesu uczenia sieci rekurencyjnej. W rozpatrywanym zadaniu klasyfikacji okazało się, że w przypadku prezentowanej architektury sieci i wykorzystania arytmetyki zmiennoprzecinkowej pojedynczej precyzji, użycie aproksymacji funkcji aktywacji o dokładności poniżej 10^{-4} (odnośnie do dokładności aproksymacji funkcji tangensa hiperbolicznego) zaczyna wpływać na wynik klasyfikacji, co szczegółowo przedstawiono w punkcie 3.5.6. W przypadku algorytmu BPTT i uczenia sieci wpływ zaczął być obserwowalny już dla dokładności poniżej 10^{-6} (również w odniesieniu do dokładności aproksymacji funkcji tangensa hiperbolicznego), co dokładnie omówiono w punkcie 4.3.4.

Analizując wyniki przedstawionych w pracy czterech wariantów klasyfikatora zaimplementowanych na FPGA zauważalne jest, że przy wykorzystaniu układów FPGA do ciągłej analizy monitorowanych sygnałów, można uzyskać dużo większą przepustowość. W tabeli 3.21. przedstawiono czasy obliczeń klasyfikatorów zaimplementowanych na FPGA oraz programów referencyjnych. Wykorzystanie FPGA sprawia, że czas obliczeń jest wielokrotnie mniejszy dla każdego z wariantów od porównywanych podejść

referencyjnych, nawet w stosunku do obu rozważanych GPU, co jest spowodowane prawdopodobnie tym, że obliczenia są zdecydowanie zbyt krótkie, żeby można było skorzystać z zalet wykorzystania GPU, a z czym świetnie radzi sobie implementacja na FPGA. Najszybsza implementacja na FPGA jest 1760 razy szybsza od GPU1 oraz 1829 razy szybsza od GPU2. Przy okazji prac nad klasyfikatorem zaobserwowano, że używany na etapie implementacji funkcji aktywacji program ISE Design Suite nie radzi sobie z tak dużą logiką i konieczne okazało się przejście na nowsze oprogramowanie firmy Xilinx tj. na Vivado.

W przypadku zadania uczenia sieci rekurencyjnej zalety FPGA są wciąż widoczne, chociaż różnica w czasie obliczeń była mniejsza, co widać w tabeli 4.2. W porównaniu z GPU1, najszybszy wariant osiągnął 60.8 razy krótsze obliczenia, a w porównaniu z GPU2 48.4 razy krótsze. Bardzo korzystny stosunek czasu obliczeń jest wciąż obserwowalny w porównaniu z obliczeniami wykonywanymi na procesorze: 269 razy szybciej niż obliczenia na CPU z wykorzystaniem Pythona oraz 116 razy szybciej niż obliczenia w C++ na platformie Raspberry Pi 3. Podczas analizy wyników można zauważyć, że problemem przy tego typu rozwiązaniach jest rozmiar sieci w strukturze FPGA wraz z dobudowanymi elementami potrzebnymi do obliczeń w ramach algorytmu BPTT, gdzie poza przejściem w przód występuje konieczność wykonywania wielu dodatkowych operacji oraz przechowywania parametrów z kroków obliczeniowych wykonywanych w trakcie przejścia w przód. Liczba wykorzystanych elementów pozwala zmieścić taką strukturę logiczną tylko na dużych układach FPGA, co zostało przedstawione w tabeli 4.1 i co można porównać z tabelą 4.7, gdzie przedstawiono parametry dostępnych na rynku FPGA. Moduł wspomagający obliczenia, wykorzystujący algorytm BPTT, mógłby zostać zaimplementowany, m.in. na ALVEO U200 lub ALVEO U250, natomiast są to stosunkowo drogie karty, stosowane głównie w centrach obliczeniowych.

W ramach eksperymentów na fizycznej płycie ewaluacyjnej wykorzystano układ z serii ZYNQ do zadania klasyfikacji oraz do zadania uczenia sieci LSTM. Ze względu na dostępność wykorzystano płytkę ewaluacyjną z układem dysponującym małą liczbą zasobów sprzętowych, co wymusiło zmianę koncepcji implementacji, a zatem i kompletną przebudowę struktury logicznej. W ramach rozwoju nowej koncepcji, zaprezentowano cztery podejścia implementacyjne dla zadania klasyfikacji, wykorzystujące płytkę ZYBO Z7-20. Rozważono podejście z implementacją jednej komórki LSTM

oraz parametrami po stronie PS. Kolejnym podejściem było wykorzystanie samego procesora, będącego częścią układu ZYNQ. Następnym podejściem było przeniesienie parametrów sieci do PL oraz rozważenie różnej liczby komórek oraz par bloków arytmetycznych. Ostatnim podejściem było wykorzystanie narzędzi do generowania struktury logicznej - w tym przypadku Vitis HLS.

W pierwszym podejściu w części logicznej zaimplementowano tylko jedną komórkę, co wymusiło bardzo intensywną komunikację z PS oraz wpłynęło negatywnie na czas obliczeń. W ramach tego podejścia rozważono różne nastawy bloków arytmetycznych, co pomimo znaczącego wpływu na częstotliwość taktowania nie wpłynęło istotnie na czas obliczeń. Pozwoliło to na minimalizację zużycia zasobów sprzętowych do bardzo małej liczby, natomiast osiągane czasy obliczeń były najwolniejsze spośród wszystkich rozważanych implementacji.

Drugie podejście miało głównie znaczenie referencyjne dla pozostałych. Wykorzystana bezpośrednio na procesorze implementacja, napisana w języku C, pozwoliła na osiągnięcie lepszych czasów obliczeń 1.65 razy (tabela 3.23), niż w przypadku obliczeń na Raspberry Pi 3, co mogło być spowodowane działaniem systemu operacyjnego na Raspberry Pi 3.

W podejściu trzecim, przeniesienie parametrów sieci na stronę PL pozwoliło na znaczące polepszenie wyników czasu obliczeń. Już nawet dla jednej komórki w strukturze PL oraz jednej pary bloków arytmetycznych, obliczenia były 2.36 razy szybsze niż podczas wykonywania ich bezpośrednio na procesorze (podejście drugie), co zostało przedstawione w tabeli 3.22. Dalsze zwiększanie liczby komórek i liczby par bloków arytmetycznych pozwoliło na przyspieszenie obliczeń 19.9 razy w stosunku do podejścia drugiego, a w porównaniu z programami referencyjnymi: 90.1 razy szybciej niż na GPU1, 93.6 razy szybciej niż na GPU2, 137 razy szybciej niż na CPU z wykorzystaniem biblioteki Pythona NumPy oraz 32.7 razy szybciej niż na Raspberry Pi 3, co zostało przedstawione w tabeli 3.23.

Ostatnie z rozważanych w tym temacie podejść pozwoliło porównać parametry generowanej struktury logicznej przez programy takie jak Vitis HLS, z implementacją pisaną ręcznie w języku Verilog. W tym przypadku parametry sieci również były przechowywane po stronie PL. Osiągnięte rezultaty pozwalają stwierdzić, że wykonując opis struktury logicznej ręcznie można lepiej zoptymalizować powstającą strukturę logiczną do rozwiązywanego problemu i przez to uzyskać dużo lepsze rezultaty, przy za-

uważalnie mniejszym zużyciu zasobów sprzętowych, co przedstawiono porównując wyniki z podejściem trzecim. Zestawiając podejście czwarte z podejściem trzecim, w oparciu o tabelę 3.22 widać, że implementacja napisana ręcznie wykorzystująca podobną liczbę LUT i FF, jest 4 razy szybsza i zużywa dużo mniej DSP (mniej o 47) i BRAM (mniej o 19). Analogiczne wnioski znajdują się w artykule [88], który między innymi porusza kwestię porównania wyników otrzymanych poprzez samodzielne zaprojektowanie struktury logicznej ze strukturą logiczną wygenerowaną przez narzędzia syntezy. To drugie podejście, tzn. wykorzystanie programów generujących strukturę logiczną, jest często spotykane w literaturze, co można zaobserwować przy okazji analizy literatury związanej z układami ZYNQ (punkt 3.6.1). Porównanie w [88] pokazuje, że w przypadku implementacji bloku funkcji aktywacji według przedstawionych tam metod, implementacje w których wykorzystano narzędzia syntezy wypadają zauważalnie gorzej, niż w przypadku ręcznego zaprojektowania takiego bloku. Warto mieć przy tym na uwadze, że zamontowany na płytce ewaluacyjnej ZYBO Z7-20 układ nie jest największym z serii. W serii ZYNQ są też większe układy np. XC7Z100-2FFG900, o którym mowa w punkcie 4.5, które pomieściłyby całą strukturę logiczną klasyfikatora, co pozwoliłoby na wykorzystanie w pełni zalet połączenia układu FPGA z procesorem w jednej obudowie. Układ ten zawiera w swojej obudowie 277400 LUT, 554800 FF oraz 2020 DSP.

Wykorzystanie płytki ZYBO Z7-20 rozważono również w kontekście zadania uczenia sieci LSTM. Próbowano wykonać analogiczne podejścia implementacyjne jak w przypadku zadania klasyfikacji. Przeniesienie parametrów sieci na stronę PL wiązało się ze znaczącym wzrostem zużycia zasobów. Spowodowane było to koniecznością implementacji dodatkowej logiki. Związana ona była nie tylko z przejściem w tył, ale również z obsługą dużej liczby parametrów, które były wykorzystywane do odczytu, zapisu nowej wartości oraz nosiły informacje o obliczanych błędach. W związku z tym, nie udało się zaimplementować podejść analogicznych do podejścia trzeciego i czwartego, rozważanych w trakcie zadania klasyfikacji. W przypadku podejścia pierwszego, po stronie PS zaimplementowano logikę wraz z parametrami oraz funkcjonalność potrzebną do ich obsługi i aktualizacji. Po stronie PL znajdowała się jedynie logika odpowiedzialna za obliczenia związane z przejściem w przód i w tył, działająca w sposób strumieniowy. W tej konfiguracji przyrost liczby wykorzystywanych zasobów był znacznie mniejszy i eksperymenty były możliwe. Rezultat eksperymentów pokazał, że

jest możliwa taka implementacja oraz liczba zużytych w takim przypadku zasobów logicznych jest bardzo niska. Osiągane w ten sposób czasy obliczeń natomiast nie były atrakcyjne. Potwierdza to wniosek, który pojawił się przy okazji rozważań dotyczących implementacji na układach FPGA, że w przypadku implementacji algorytmów uczenia sieci neuronowych wymagana jest duża liczba zasobów sprzętowych.

Rozwój technologii i rosnące zainteresowanie układami FPGA pozwala sądzić, że powszechność układów FPGA będzie rosła wraz z liczbą dostępnych w ich strukturze zasobów. Już teraz dostępne układy pozwalają na implementacje złożonych konfiguracji sprzętowych i zastosowanie układów FPGA do różnorodnych zadań, często nawet wymagających złożonych obliczeń. Pojawienie się niedawno układów Versal pokazuje, że zainteresowanie dalszym rozwojem takich technologii, w zastosowaniu uczenia maszynowego, jest obecne wśród producentów układów konfigurowalnych, a wyniki przedstawione w niniejszej pracy pozwalają sądzić, że wykorzystanie układów FPGA w przypadku sieci LSTM może być bardzo atrakcyjne, zwłaszcza w kontekście problemów wymagających bardzo krótkich czasów obliczeń.

Syntetycznie podsumowując, w ramach niniejszej rozprawy zrealizowano następujące prace:

1) Przedstawiono problematykę rekurencyjnych sztucznych sieci neuronowych ze szczególnym uwzględnieniem sieci LSTM.

2) Przedstawiono tematykę układów programowalnych FPGA, ich strukturę, metodykę wykorzystania ich w projekcie oraz przedstawiono układy logiczne łączące w sobie układ FPGA oraz procesor, z serii ZYNQ oraz Versal.

3) Przeanalizowano spotykane w literaturze podejścia do implementacji funkcji aktywacji oraz zaprezentowano implementacje funkcji wykorzystujące różne podejścia, tj. nowe podejście wzorowane na algorytmie CORDIC, podejście z użyciem wielomianów Czebyszewa oraz podejście z użyciem zwykłych wielomianów. Zaprezentowane implementacje zostały wykonane w wielu wariantach i szeroko przebadane, w wyniku czego wyselekcjonowano 4 warianty wykorzystywane dalej w pracy.

4) W celu sprawdzenia użyteczności implementowanych modułów, również w szerszym kontekście sieci, na podstawie wyselekcjonowanych wariantów utworzono

komórkę LSTM. Głównym kryterium projektowym komórki była szybkość obliczeń przy możliwie małym wykorzystaniu zasobów sprzętowych.

5) Utworzona komórka została przetestowana, a na dalszym etapie wykorzystana do budowy sieci oraz porównana z innymi podejściami spotykanymi w literaturze.

6) Zaimplementowano sieć LSTM wykorzystującą układ FPGA do rozwiązania rzeczywistego procesu przemysłowego, tj. procesu kucia na zimno, oraz porównano implementację z programami referencyjnymi, w tym z korzystającymi z GPU oraz z literaturą. Prezentowana implementacja znacząco różniła się od spotykanych do tej pory i osiągała bardzo dobre rezultaty.

7) Implementację sieci LSTM wykonano również z wykorzystaniem układów Xilinx ZYNQ, które łączą w sobie układ FPGA i procesor. Prezentowane w tym zakresie podejścia różniły się znacząco od prezentowanej wcześniej implementacji na FPGA oraz od implementacji spotykanych w literaturze. Przeanalizowano podejścia różnie dzielące przeznaczenie FPGA i procesora oraz porównano implementację bezpośrednią w Verilogu z wygenerowaną przez narzędzia HLS.

8) W oparciu o implementację sieci LSTM na FPGA przeanalizowano wpływ dokładności aproksymacji funkcji aktywacji na klasyfikację.

9) Rozszerzono funkcjonalność przedstawionych klasyfikatorów o możliwość przeprowadzenia procesu uczenia ze szczególnym naciskiem na algorytm BPTT w sieciach LSTM. Rozszerzenie funkcjonalności wykonano zarówno dla implementacji korzystającej z samego układu FPGA oraz dla implementacji korzystającej z układu Xilinx ZYNQ.

10) Przeanalizowano wpływ liczby komórek na czas uczenia sieci LSTM oraz wpływ dokładności aproksymacji funkcji aktywacji na proces uczenia.

11) Przeanalizowano koszt prezentowanych podejść.

12) Przedstawiono wnioski końcowe.

Postawione cele pracy udało się osiągnąć. Przedstawione w ramach niniejszej pracy implementacje sieci LSTM różniły się od spotykanych w literaturze sposobem wy-

konywania obliczeń związanych z funkcjami aktywacji oraz organizacją obliczeń w obrębie komórki i sieci. Omawiane podejście do implementacji sieci LSTM wykorzystującej układy FPGA zostało zweryfikowane poprzez wykonanie klasyfikatora do procesu kucia na zimno. Prezentowane implementacje rozszerzono o możliwość przeprowadzania procesu uczenia tworząc w ten sposób układ cyfrowy w strukturach FPGA wspomagający proces uczenia sieci LSTM.

Przeprowadzone w ramach niniejszej pracy badania potwierdzają postawione w punkcie 1 tezy pracy. Tezę 1 potwierdzają wyniki przedstawione w ramach punktu 3. Przedstawione tam implementacje z wykorzystaniem FPGA osiągają dużo większe szybkości obliczeń w porównaniu z przygotowanymi programami referencyjnymi, będąc nawet do 1760 razy szybszymi od GPU1 (GeForce GTX 1060 6GB, Intel Core i7-6700K CPU 4.00GHz, 16GB RAM, Windows 10) i 1829 razy szybszymi od GPU2 (NVIDIA GeForce RTX 3080 Ti, Intel Core i9-12900H 5.00GHz, 64GBRAM, Ubuntu 20.04). Nawet w przypadku wykorzystania układów ZYNQ udało się znacząco przyspieszyć obliczenia, tj. do 90.1 razy szybciej niż GPU1 oraz 93.6 razy szybciej niż GPU2. Tezę 2 również potwierdzają przedstawione w ramach pracy wyniki, w punkcie 4. W tym przypadku potwierdzenie tezy dotyczy jedynie wyników otrzymanych dla FPGA, gdzie osiągnięto do 60.8 razy szybsze obliczenia względem GPU1 oraz 48.4 razy szybsze od GPU2. Zużycie zasobów uniemożliwiło odwzorowanie wszystkich planowanych implementacji, wykorzystujących układ ZYNQ, w związku z czym niemożliwe okazało się zweryfikowanie w tym konkretnym zakresie tezy 2. Tezę tę udało się przetestować dla podejścia mieszczącego się na ZYNQ, co poskutkowało uzyskaniem wyników, której jej nie potwierdziły.

Możliwe dalsze kierunki rozwoju tematyki pracy:

1) Wykorzystanie układów Xilinx Versal do budowy klasyfikatora oraz budowy układu cyfrowego wspomagającego uczenie sieci LSTM.

2) Wykonanie zaprezentowanych implementacji z uwzględnieniem różnych reprezentacji liczbowych, w tym różnej liczby bitów, różnego podziału liczby na część dziesiętną i całkowitą oraz sprawdzenie ich wpływu na zapotrzebowanie sprzętowe, zadanie klasyfikacji i uczenie sieci.

3) Wykonanie implementacji sieci bi-LSTM oraz GRU, będącymi innymi rodzajami rekurencyjnych sieci neuronowych.

Literatura

[1] F. Chollet, Deep Learning with Python (Manning, November 2017).

[2] T. Żabiński, T. Mączka, J. Kluska, M. Kusy, Z. Hajduk and S. Prucnal, Failures Prediction in the Cold Forging Process Using Machine Learning Methods, in Artificial Intelligence and Soft Computing eds. L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh and J. M. Zurada (Springer International Publishing, Cham, 2014) 622–633.

[3] R. Miotto, F. Wang, S. Wang, X. Jiang and J. T. Dudley, Deep learning for healthcare: review, opportunities and challenges, Briefings in Bioinformatics **19** (05 2017) 1236–1246.

[4] A. Graves, N. Jaitly and A.-r. Mohamed, Hybrid speech recognition with Deep Bidirectional LSTM, in 2013 IEEE Workshop on Automatic Speech Recognition and Understanding 2013 273–278.

[5] B. Ren, The use of machine translation algorithm based on residual and LSTM neural network in translation teaching, PLOS ONE **15** (11 2020) p. e0240663.

[6] M. Sharp, R. Ak and T. Hedberg, A survey of the advancing use and development of machine learning in smart manufacturing, Journal of Manufacturing Systems **48** (2018) 170–179, Special Issue on Smart Manufacturing.

[7] R. Sahal, J. G. Breslin and M. I. Ali, Big data and stream processing platforms for Industry 4.0 requirements mapping for a predictive maintenance use case, Journal of Manufacturing Systems **54** (2020) 138–151.

[8] K. Chen, L. Huang, M. Li, X. Zeng and Y. Fan, A Compact and Configurable Long Short-Term Memory Neural Network Hardware Architecture, in 2018 25th IEEE International Conference on Image Processing (ICIP) 2018 4168–4172.

[9] S. Bouguezzi, H. Faiedh and C. Souani, Hardware Implementation of Tanh Exponential Activation Function using FPGA, in 2021 18th International Multi-Conference on Systems, Signals Devices (SSD) 2021 1020–1025.

[10] A. M. Soares, L. C. Leite, J. O. P. Pinto, L. E. B. da Silva, B. K. Bose and M. E. Romero, Field Programmable Gate Array (FPGA) Based Neural Network

Implementation of Stator Flux Oriented Vector Control of Induction Motor Drive, in 2006 IEEE International Conference on Industrial Technology 2006 31–34.

[11] T. De Ryck, S. Lanthaler and S. Mishra, On the approximation of functions by tanh neural networks, Neural Networks **143** (2021) 732–750.

[12] S. Mujawar, D. Kiran and H. Ramasangu, An Efficient CNN Architecture for Image Classification on FPGA Accelerator, in 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC) 2018 1–4.

[13] S. Zhai, C. Qiu, Y. Yang, J. Li and Y. Cui, Design of Convolutional Neural Network Based on FPGA, Journal of Physics: Conference Series **1168** (feb 2019) p. 062016.

[14] S. Hochreiter and J. Schmidhuber, Long Short-Term Memory, Neural Comput. **9** (November 1997) p. 1735–1780.

[15] A. Howard and S. Gupta, Introducing the Next Generation of On-Device Vision Models: MobileNetV3 and MobileNetEdgeTPU, Google Research (2019).

[16] Cloud TPU pricing https://cloud.google.com/tpu/pricing, Accessed: 2023-04-15.

[17] P. Ferreira, P. Ribeiro, A. Antunes and F. M. Dias, A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function, Neurocomputing **71**(1) (2007) 71–77, Dedicated Hardware Architectures for Intelligent Systems Advances on Neural Networks for Speech and Audio Processing.

[18] V. Rybalkin and N. Wehn, When Massive GPU Parallelism Ain't Enough: A Novel Hardware Architecture of 2D-LSTM Neural Network, in Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays FPGA '20, (Association for Computing Machinery, New York, NY, USA, 2020) p. 111–121.

[19] H. Hashimoto, R. Naka and Y. Wada, An FPGA-Based Image Recognition with Remote Update Functions for Autonomous Driving on "ad-refkit", in 2021 International Conference on Field-Programmable Technology (ICFPT) 2021 1–3.

[20] D. Baptista and F. Morgado-Dias, Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks, Neural Computing and Applications **23** (09 2013) 601–607.

[21] Zynq 7000 Overview https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview, Accessed: 2023-04-15.

[22] Versal Architecture and Product Data Sheet: Overview (DS950) https://docs.xilinx.com/v/u/en-US/ds950-versal-overview, Accessed: 2023-04-15.

[23] Most Popular Machine Learning Libraries – 2014/2021 `https://statisticsanddata.org/data/most-popular-machine-learning-libraries`, Accessed: 2023-04-15.

[24] TensorFlow Overview https://www.tensorflow.org/api$_d ocs/python/tf, Accessed$: $2023 - 04 - 15.

[25] Keras API reference https://keras.io/api/, Accessed: 2023-04-15.

[26] NumPy documentation https://numpy.org/doc/stable, Accessed: 2023-04-15.

[27] SDAccel Environment Profiling and Optimization Guide `https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/gac1504034293050.html`, Accessed: 2023-04-15.

[28] Karta charakterystyki UltraScale+ PCIe z VU13P `https://www.mouser.pl/datasheet/2/276/2/ds-xup-vv4-1858129.pdf`, Accessed: 2023-04-15.

[29] FPGA Parallelism Versus Processor Architectures `https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/pld1504034294759.html`, Accessed: 2023-04-15.

[30] C. Maxfield, The Design Warrior's Guide to FPGAs 2004.

[31] E. Haskell, Richard and M. Hanna, Darrin, Introduction to Digital Design Using Digilent FPGA Boards Block Diagram / VHDL Exa 2009.

[32] I. Grout, Digital Systems Design with FPGAs and CPLDs (Elsevier, 2008).

[33] J. Kalisz, Podstawy elektroniki cyfrowej 1998.

[34] S. Jaffry, Development of wave propagation imaging technology using field programmable gate array (fpga), PhD thesis02 2014

[35] J. Zhang, Q. Wu, D. Yipeng, Y.-Q. Lv, Q. Zhou, Z.-H. Xia, X.-M. Sun and X.-W. Wang, Techniques for design and implementation of an fpga-specific physical unclonable function, Journal of Computer Science and Technology **31** (01 2016) 124–136.

[36] Introduction to Versal ACAP `https://docs.xilinx.com/r/en-US/am005-versal-clb`, Accessed: 2023-04-15.

[37] Understanding FPGA Architecture `https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/odz1504034293215.html`, Accessed: 2023-04-15.

[38] 7-Series DSP Resources `https://www.xilinx.com/video/fpga/7-series-dsp-resources.html`, Accessed: 2023-04-15.

[39] UltraScale Architecture DSP Slice `https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp`, Accessed: 2023-04-15.

[40] DSP Macro LogiCORE IP Product Guide `https://docs.xilinx.com/r/en-US/pg323-dsp-macro`, Accessed: 2023-04-15.

[41] Model-Based DSP Design Using System Generator `https://www.xilinx.com/support/documents/sw_manuals/xilinx2021_2/ug958-vivado-sysgen-ref.pdf`, Accessed: 2023-04-15.

[42] Vivado Design Suite User Guide : Designing with IP (UG896) `https://docs.xilinx.com/r/2022.1-English/ug896-vivado-ip/IP-Centric-Design-Flow`, Accessed: 2023-04-15.

[43] Block RAM Introduction `https://docs.xilinx.com/r/en-US/am007-versal-memory/Block-RAM?tocId=4P4QDFML7TiXdlJaQlcRog`, Accessed: 2023-04-15.

[44] 7 Series FPGAs Data Sheet: Overview `https://datasheet.lcsc.com/lcsc/2108141630_XILINX-XC7K160T-2FFG676I_C966964.pdf`, Accessed: 2023-04-15.

[45] S. Kilts, Advanced FPGA Design Architecture, Implementation, and Optimization 2007.

[46] IEEE Spectrum's Top Programming Languages 2022 https://spectrum.ieee.org/top-programming-languages-2022, Accessed: 2023-04-15.

[47] J. Ashenden, Peter, VHDL Tutorial (Elsevier Science, 2004).

[48] L. Perry, Douglas, VHDL: Programming by Example (The McGraw-Hill Companies, Inc., 2002).

[49] Ieee standard verilog hardware description language, IEEE Std 1364-2001 (2001) 1–792.

[50] Z. Hajduk, Wprowadzenie do języka Verilog 2009.

[51] Ieee standard for systemverilog: Unified hardware design, specification and verification language, IEEE Std 1800-2005 (2005) 1–648.

[52] Ieee standard for systemverilog–unified hardware design, specification, and verification language, IEEE STD 1800-2009 (2009) 1–1285.

[53] Ieee standard for systemverilog–unified hardware design, specification, and verification language, IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009) (2013) 1–1315.

[54] Ieee standard for systemverilog–unified hardware design, specification, and verification language, IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) (2018) 1–1315.

[55] Vivado Design Suite User Guide: Synthesis (UG901) `https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/Verilog-2001-Support`, Accessed: 2023-04-15.

[56] V. Salauyou, A. Klimowicz, T. Grześ and I. Bułatowa, JĘZYK VERILOG W PROJEKTOWANIU SYSTEMÓW WBUDOWANYCH NA UKŁAI 2022.

[57] P. Chu, P., FPGA PROTOTYPING BY VERILOG EXAMPLES 2008.

[58] K. Patel, Meher, FPGA designs with Verilog and SystemVerilog 2017.

[59] S. Ramagond, S. Yellampalli and C. Kanagasabapathi, A review and analysis of communication logic between pl and ps in zynq ap soc, in 2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon) 2017 946–951.

[60] AXI Reference Guide https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide, Accessed: 2023-04-15.

[61] AXI4-Stream Documentation `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/How-AXI4-Stream-Works`, Accessed: 2023-04-15.

[62] Zynq 7000 SoC https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, Accessed: 2023-04-15.

[63] Zybo Z7 Board Reference Manual `https://digilent.com/reference/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf`, Accessed: 2023-04-15.

[64] Y. Wang, S. Chen, K. Zhang, H. Chen and X. Chen, Architecture design trade-offs among vliw simd and multi-core schemes, in 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum 2012 1649–1658.

[65] M. Wijtvliet, A. Kumar and H. Corporaal, Blocks: Challenging simds and vliws with a reconfigurable architecture, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **41**(9) (2022) 2915–2928.

[66] Versal: The First Adaptive Compute Acceleration Platform (ACAP) https://docs.xilinx.com/v/u/en-US/wp505-versal-acap, Accessed: 2023-04-15.

[67] AI Engines and Their Applications https://docs.xilinx.com/v/u/en-US/wp506-ai-engine, Accessed: 2023-04-15.

[68] B. Gaide, D. Gaitonde, C. Ravishankar and T. Bauer, Xilinx adaptive compute acceleration platform: Versaltm architecture, in Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays

FPGA '19, (Association for Computing Machinery, New York, NY, USA, 2019) p. 84–93.

[69] ISE In-Depth Tutorial https://www.xilinx.com/htmldocs/xilinx13_3/ise_tutorial_ug695.pdf, Accessed: 2023-04-15.

[70] Vivado Design Suite User Guide Using the Vivado IDE https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_2/ug893-vivado-ide.pdf}, Accessed: 2023-04-15.

[71] Vivado Design Suite User Guide Getting Started https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug910-vivado-getting-started.pdf, Accessed: 2023-04-15.

[72] Vitis Documentation https://docs.xilinx.com/v/u/en-US/ug1416-vitis-documentation, Accessed: 2023-04-15.

[73] M. Woźniak, M. Wieczorek and J. Siłka, Bilstm deep neural network model for imbalanced medical data of iot systems, Future Generation Computer Systems **141** (2023) 489–499.

[74] R. Leszek, Metody i techniki sztucznej inteligencji (Wydawnictwo Naukowe PWN, 2023).

[75] J. C. Ferreira and J. Fonseca, An FPGA implementation of a long short-term memory neural network, in 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig) 2016 1–8.

[76] V. Tiwari and N. Khare, Hardware implementation of neural network with sigmoidal activation functions using cordic, Microprocessors and Microsystems **39** (2015) 373–381.

[77] F. Ortega-Zamorano, J. M. Jerez, D. Urda Muñoz, R. M. Luque-Baena and L. Franco, Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers, IEEE Transactions on Neural Networks and Learning Systems **27**(9) (2016) 1840–1850.

[78] A. M. Abdelsalam, J. M. P. Langlois and F. Cheriet, A Configurable FPGA Implementation of the Tanh Function Using DCT Interpolation, in 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2017 168–171.

[79] S. Baraha and P. K. Biswal, Implementation of activation functions for ELM based classifiers, in 2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET) 2017 1038–1042.

[80] I. del Campo, R. Finker, J. Echanobe and B. K., Controlled accuracy approximation of sigmoid function for efficient fpga-based implementation of artificial neurons, Electronics Letters **49**(25) (2013) 1598–1600.

[81] I. Koyuncu, Implementation of high speed tangent sigmoid transfer function approximations for artificial neural network applications on fpga, Advances in Electrical and Computer Engineering **18**(3) (2018) 79–86.

[82] S. Gomar, M. Mirhassani and M. Ahmadi, Precise digital implementations of hyperbolic tanh and sigmoid function, in 2016 50th Asilomar Conference on Signals, Systems and Computers 2016 1586–1589.

[83] A. Armato, L. Fanucci, P. G. and D. R. D., Low-error approximation of artificial neuron sigmoid function and its derivative, Electronics Letters **45**(21) (2009) 1–2.

[84] A. Armato, L. Fanucci, S. E.P. and D. R. D., Low-error digital hardware implementation of artificial neuron activation functions and their derivative, Microprocessors and Microsystems **35** (2011) 557–567.

[85] T. Orlowska-Kowalska and M. Kaminski, Fpga implementation of the multilayer neural network for the speed estimation of the two-mass drive system, IEEE Trns. on Industrial Informatics **7**(3) (2011) 557–567.

[86] A. Gomperts, A. Ukil and F. Zurfluh, Development and implementation of parameterized fpga-based general purpose neural networks for online applications, IEEE Transactions on Industrial Informatics **7**(1) (2011) 78–89.

[87] Z. M. Shakiba, F.M., Novel analog implementation of a hyperbolic tangent neuron in artificial neural networks, IEEE Transactions on Industrial Electronics **68**(11) (2020) 10856–10867.

[88] Z. Hajduk, Hardware implementation of hyperbolic tangent and sigmoid activation functions, Bulletin of the Polish Academy of Sciences Technical Sciences **66**(5) (2018) 563–577.

[89] P. Malík, High throughput floating point exponential function implemented in FPGA, 2015 IEEE Computer Society Annual Symposium on VLSI (2015) 97–100.

[90] H. M. Al-Rikabi, M. A. Al-Ja'afari, A. H. Ali and S. H. Abdulwahed, Generic model implementation of deep neural network activation functions using GWO-optimized SCPWL model on FPGA, Microprocessors and Microsystems **77** (2020) p. 103141.

[91] B. Pasca and M. Langhammer, in 2018 28th International Conference on Field Programmable Logic and Applications (FPL)

[92] S. Wang, P. Lin, R. Hu, H. Wang, J. He, Q. Huang and S. Chang, Acceleration of LSTM With Structured Pruning Method on FPGA, IEEE Access **7** (2019) 62930–62937.

[93] J. Sujitha and V. R. Reddy, Implementation of Log and Exponential Function in FPGA, **tom 03**(11) (2014).

[94] M. Garrido, P. Källström, M. Kumm and O. Gustafsson, CORDIC II: A New Improved CORDIC Algorithm, IEEE Transactions on Circuits and Systems II: Express Briefs **63**(2) (2016) 186–190.

[95] G. R. Dec, LSTM cell implementation on FPGAs, Parallel Processing Letters **31**(02) (2021).

[96] Z. Hajduk and G. R. Dec, Very high accuracy hyperbolic tangent function implementation in fpgas, IEEE Access **11** (2023) 23701–23713.

[97] S. Bouguezzi, H. Faiedh and C. Souani, Hardware implementation of tanh exponential activation function using fpga, International Multi-Conference on Systems Signals Devices (2021) p. 1020–1025.

[98] I. Tsmots, O. Skorokhoda and V. Rabyk, Hardware Implementation of Sigmoid Activation Functions using FPGA, in 2019 IEEE 15th International Conference

on the Experience of Designing and Application of CAD Systems (CADSM) 2019 34–38.

[99] H. Nakahara, Z. Que and W. Luk, High-throughput convolutional neural network on an fpga by customized jpeg compression, in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2020 1–9.

[100] Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, X. Li and X. Zhou, A Power-Efficient Accelerator Based on FPGAs for LSTM Network, in 2017 IEEE International Conference on Cluster Computing (CLUSTER) 2017 629–630.

[101] Y. Guan, Z. Yuan, G. Sun and J. Cong, FPGA-based accelerator for long short-term memory recurrent neural networks, in 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC) 2017 629–634.

[102] A. X. M. Chang, B. Martini and E. Culurciello, Recurrent Neural Networks Hardware Implementation on FPGA (2016).

[103] D. He, J. He, J. Liu, J. Yang, Q. Yan and Y. Yang, An fpga-based lstm acceleration engine for deep learning frameworks, Electronics **10**(6) (2021).

[104] Z. Sun, Y. Zhu, Y. Zheng, H. Wu, Z. Cao, P. Xiong, J. Hou, T. Huang and Z. Que, Fpga acceleration of lstm based on data for test flight, in 2018 IEEE International Conference on Smart Cloud (SmartCloud) 2018 1–6.

[105] W. Zhang, F. Ge, C. Cui, Y. Yang, F. Zhou and N. Wu, Design and implementation of lstm accelerator based on fpga, in 2020 IEEE 20th International Conference on Communication Technology (ICCT) 2020 1675–1679.

[106] J. Jiang, T. Xiao, J. Xu, D. Wen, L. Gao and Y. Dou, A low-latency lstm accelerator using balanced sparsity based on fpga, Microprocessors and Microsystems **89** (2022) p. 104417.

[107] G. R. Dec, FPGA-based Neural Net for Failures Prediction in the Cold Forging Process, Parallel Processing Letters **32**(01n02) (2021).

[108] S. Yu, W. Zeng, J. Guo and Y. Liu, Image processing and return system based on zynq, in 2020 2nd International Conference on Information Technology and Computer Application (ITCA) 2020 307–311.

[109] M. Koushik, S. Shivanagi, G. Gupta, J. Qumar and D. Saravanan, Implementation of g.723.1decoder on zynq fpga using hls, in 2017 International Conference on Inventive Computing and Informatics (ICICI) 2017 263–266.

[110] R. Mamarde and S. Khoje, Viterbi decoder using zynq-7000 ap-soc, in 2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS) 2018 941–944.

[111] B. Banerjee, Z. Khan, J. J. Lehtomäki and M. Juntti, Deep learning based over-the-air channel estimation using a zynq sdr platform, IEEE Access **10** (2022) 60610–60621.

[112] R. Rajesh, S. J. Darak, A. Jain, S. Chandhok and A. Sharma, Hardware–software co-design of statistical and deep-learning frameworks for wideband sensing on zynq system on chip, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **31**(1) (2023) 79–89.

[113] Z. Cao and X. Li, A design of zynq-soc-based miniaturized navigation controller, in CSAA/IET International Conference on Aircraft Utility Systems (AUS 2018) 2018 1–5.

[114] J. Soh and X. Wu, An fpga-based unscented kalman filter for system-on-chip applications, IEEE Transactions on Circuits and Systems II: Express Briefs **64**(4) (2017) 447–451.

[115] S. Kasap, S. Redif and E. Wachter, Acceleration of polynomial matrix multiplication on zynq-7000 system-on-chip, in 2019 32nd IEEE International System-on-Chip Conference (SOCC) 2019 300–305.

[116] O. Bartik, The implementation of the real-time model of the asynchronous motor and the two mass mechanical load at zynq-7000, in 2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE) 2017 150–155.

[117] O. Kerdjidj, K. Amara, F. Harizi and H. Boumridja, Implementing hand gesture recognition using emg on the zynq circuit, IEEE Sensors Journal (2023) 1–1.

[118] X. Zhai, A. A. S. Ali, A. Amira and F. Bensaali, Mlp neural network based gas classification system on zynq soc, IEEE Access **4** (2016) 8138–8146.

[119] A. X. M. Chang and E. Culurciello, Hardware accelerators for recurrent neural networks on fpga, in 2017 IEEE International Symposium on Circuits and Systems (ISCAS) 2017 1–4.

[120] G. R. Dec, FPGA-based Learning Acceleration for LSTM Neural Network, Parallel Processing Letters **33**(01n02) (2021).

[121] A. A. Bataineh, D. Kaur and A. Jarrah, Enhancing the Parallelization of Backpropagation Neural Network Algorithm for Implementation on FPGA Platform, in NAECON 2018 - IEEE National Aerospace and Electronics Conference 2018 192–196.

[122] H. M. Vo, Implementing the on-chip backpropagation learning algorithm on FPGA architecture, in 2017 International Conference on System Science and Engineering (ICSSE) 2017 538–541.

[123] S. L. Pinjare and A. Kumar, Implementation of Neural Network Back Propagation Training Algorithm on FPGA, International Journal of Computer Applications **52** (08 2012) 975–8887.

[124] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink and J. Schmidhuber, LSTM: A Search Space Odyssey, IEEE Transactions on Neural Networks and Learning Systems **28**(10) (2017) 2222–2232.

[125] D. P. Kingma and J. Ba, Adam: A Method for Stochastic Optimization (2014).

[126] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno and P. H. Jones, Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels, in 2019 IEEE International Conference on Embedded Software and Systems (ICESS) 2019 1–8.

[127] A. Siricharoenpanich, S. Wiriyasart and P. Naphon, Study on the thermal dissipation performance of gpu cooling system with nanofluid as coolant, Case Studies in Thermal Engineering **25** (2021) p. 100904.

Streszczenie

Autor: mgr inż. Grzegorz Rafał Dec, numer albumu: d491
Promotor: dr hab. inż. Zbigniew Hajduk, prof. PRz
Słowa kluczowe: FPGA, ZYNQ, LSTM, BPTT

Celem pracy doktorskiej było przeprowadzanie analizy możliwości implementacyjnych rekurencyjnych sieci neuronowych typu LSTM na układach FPGA z intencją uzyskania porównywalnych lub krótszych czasów obliczeń w stosunku do GPU lub uzyskania wysokiej wydajności obliczeniowej w sytuacjach, gdy GPU lub TPU nie może zostać wykorzystane. W ramach pracy opracowano nową metodę implementacyjną wykorzystującą zarówno układy FPGA, jak i układy hybrydowe z wbudowanym procesorem i matrycą programowalną, zarówno dla zadania klasyfikacji w wybranym procesie przemysłowym, jak i wspierającego proces uczenia sieci LSTM dla wybranego procesu przemysłowego. Prezentowane w ramach pracy podejścia porównano z dotychczas opracowanymi implementacjami.

Zakres pracy obejmuje analizę różnych metod implementacji funkcji aktywacji, ze szczególnym uwzględnieniem dokładności, szybkości działania i wykorzystanych zasobów, zaprezentowanie nowego podejścia implementacyjnego oraz wyselekcjonowanie na drodze eksperymentów różnych wariantów, w tym spotykanych w literaturze, wykorzystywanych w dalszych badaniach. Ponadto obejmuje implementację modułu komórki LSTM z nowym sposobem organizacji obliczeń i prezentację wykorzystania omawianego modułu do celów budowy dowolnej sieci z warstwą LSTM. Dodatkowo, zaprojektowano i zaimplementowano sieć LSTM w strukturach FPGA realizującą zadanie detekcji wadliwego uderzenia w procesie kucia na zimno w oparciu o klasyfikację binarną. Wykonano różne warianty implementacyjne z wykorzystaniem układu FPGA oraz układu z podziałem na sprzęt i oprogramowanie tj. Xilinx ZYNQ. Oprócz tego zakres pracy obejmuje projekt i implementację układu cyfrowego wspierającego proces uczenia sieci LSTM, będących rozszerzeniem wariantów przygotowanych w ramach projektu klasyfikatora do zadania kucia na zimno. Przedstawione w pracy podejścia zostały przeanalizowane pod kątem dostępnych na rynku układów FPGA, uwzględniając analizę kosztów oraz ilość dostępnych zasobów sprzętowych. W ramach pracy wykonano analizę wpływu dokładności aproksymacji funkcji aktywacji na działanie sieci, zarówno w przypadku zadania klasyfikacji, jak i na proces uczenia sieci LSTM.

Uzyskane wyniki pozwalają sądzić, że wykorzystanie układów FPGA oraz układów hybrydowych w celu implementacji sieci LSTM jest wykonalne i może poskutkować znaczącym przyspieszeniem obliczeń, nawet w porównaniu z GPU. Możliwe jest również wykonanie układu cyfrowego w strukturach FPGA wspomagającego uczenie sieci LSTM.

Summary

Author: mgr inż. Grzegorz Rafał Dec, numer albumu: d491
Supervisor: dr hab. inż. Zbigniew Hajduk, prof. PRz
Key words: FPGA, ZYNQ, LSTM, BPTT

The aim of this work is to investigate a possible implementation of RNNs, specifically LSTMs, in FPGAs, in order to achieve comparable or better calculation time to GPU or greater performance in cases where GPU or TPU cannot be utilized. A novel method was identified in this paper, presented and compared to those already in use. The approach utilized an FPGA as well as a hybrid hardware architecture, which contained a built-in processor and a programmable hardware, in a classification task of a chosen industrial process and in support of the created LSTM network's training process.

An analysis of various implementations of activation function was performed, with a focus on accuracy, calculation time and resources utilization. Subsequently, approaches already present in literature were selected for further experiments alongside the one proposed by the author. In addition to that, an LSTM cell module was implemented with multiple approaches, including novel organization of calculations, in order to examine its possible use in custom-made neural networks of any kind with an LSTM layer. In order to inspect the exhibited approach, the LSTM cell was implemented in FPGA and then tested in a real-life problem, in that case in a binary classification during failure prediction in cold forging process. Different implementation options on an FPGA were considered as well as on a hybrid hardware (software and hardware separately), specifically on Xilinx ZYNQ. During a classification task for cold forging process, in order to design and implement a digital circuit for training purposes of the LSTM network, several concepts considered in the research were further developed. All of the presented approaches were analyzed in terms of hardware availability, costs and resources required for them to operate properly. Eventually, the research was also focused on exploring the significance of accuracy of the activation functions, especially in terms of its possible effect on a classification task as well as on a training process of an LSTM network.

The results obtained in this work provide a solid basis for consideration of FPGAs and hybrid systems in LSTM network implementations, which may result in a

significant acceleration of calculations, compared even to GPUs. Also, the results have proven that it is possible to create a digital circuit in FPGA structures supporting the training of the LSTM network.