Układy FPGA w systemach sterowania i regulacji

Praca zbiorowa pod redakcją
Grzegorza Karpiela

Katedra Robotyki i Mechatroniki
Akademia Górniczo-Hutnicza w Krakowie
Kraków 2023

Układy FPGA w systemach sterowania i regulacji

Praca zbiorowa pod redakcją
Grzegorza Karpiela
AGH Akademia Górniczo-Hutnicza
im. Stanisława Staszica

Akademia Górniczo-Hutnicza w Krakowie
Katedra Robotyki i Mechatroniki
Kraków
2023

ISBN 978-83-965598-5-2

SPIS TREŚCI

Od redaktora ... 6
Rozdział 1 – Platformy sprzętowe ... 7
1.1 Platformy sprzętowe w układach sterowania i regulacji ... 7
1.2 Mikrokontrolery z rdzeniem ARM Cortex ... 8
1.3 Układy reprogramowalne ... 12
1.4 Układy hybrydowe ... 17
1.5. Bibliografia ... 20
Rozdział 2 – Implementacja w układzie FPGA ... 22
2.1 Języki opisu sprzętu ... 22
2.1.1 Przegląd języków programowania ... 22
2.1.2 Projektowanie modułów sprzętowych przy użyciu języków HDL ... 24
2.2 Quartus Prime ... 27
2.3 Narzędzia do projektowania systemu metodą zintegrowaną ... 28
2.4 Procesor Nios II ... 29
2.5 Bibliografia ... 33
Rozdział 3 – Implementacja obliczeń arytmetycznych na liczbach zmiennoprzecinkowych ... 35
3.1 Reprezentacja liczby zmiennoprzecinkowej ... 35
3.2 Jednostka zmiennoprzecinkowa FPU ... 37
3.2.1 Dodawanie i odejmowanie liczb zmiennoprzecinkowych ... 37
3.2.2 Mnożenie liczb zmiennoprzecinkowych ... 38
3.2.3 Dzielenie liczb zmiennoprzecinkowych ... 39
3.2.4 Testy jednostki zmiennoprzecinkowej FPU ... 40
3.3 Funkcje trygonometryczne sinus i kosinus ... 41
3.4 Pierwiastek kwadratowy ... 43
3.5 Funkcja nasycenia (saturacji) ... 46
3.6 Funkcja signum ... 47
3.7 Funkcja rzutowania float ⇔ int ... 48

3.8 Testy wydajności obliczeniowej pod kątem wykorzystania w układzie sterowania frezarką 50
3.9 Bibliografia 60
Rozdział 4 – Implementacja z wykorzystaniem języków opisu sprzętu HDL 63
4.1. VHDL 63
4.1.1 Język VHDL 63
4.1.2 Biblioteki i pakiety 65
4.1.3 Jednostka projektowa 66
4.1.9 Identyfikatory i komentarze 66
4.1.4 Porty wejścia/wyjścia 67
4.1.5 Sygnały, stałe, zmienne, aliasy 68
4.1.6 Typy danych 72
4.1.7 Tablice 74
4.1.8 Operatory 75
4.1.9 Funkcje 76
4.1.10 Komponenty 76
4.2 Implementacja układów cyfrowych 80
4.2.1 Układy kombinacyjne 80
4.2.2 Układy sekwencyjne 87
4.3 Implementacja obliczeń stałoprzecinkowych 93
4.3.1 Dodawanie i odejmowanie 93
4.3.2 Mnożenie 93
4.3.3 Dzielenie 94
4.3.4 Funkcje trygonometryczne 95
4.5. Bibliografia 98

AUTORZY:

Redaktor – dr inż. Grzegorz Karpiel – AGH Akademia Górniczo-Hutnicza im. St. Staszica w Krakowie, Katedra Robotyki i Mechatroniki, gkarpiel@agh.edu.pl

Rozdział 1 – dr inż. Grzegorz Góra – AGH Akademia Górniczo-Hutnicza im. St. Staszica w Krakowie, Katedra Robotyki i Mechatroniki, ggora@agh.edu.pl

Rozdział 2 – dr inż. Konrad Gac – AGH Akademia Górniczo-Hutnicza im. St. Staszica w Krakowie, Katedra Robotyki i Mechatroniki, kgac@agh.edu.pl

Rozdział 3 – dr inż. Konrad Gac – AGH Akademia Górniczo-Hutnicza im. St. Staszica w Krakowie, Katedra Robotyki i Mechatroniki, kgac@agh.edu.pl

Rozdział4 – dr inż. Grzegorz Góra – AGH Akademia Górniczo-Hutnicza im. St. Staszica w Krakowie, Katedra Robotyki i Mechatroniki, ggora@agh.edu.pl

Od redaktora

Układy FPGA (*ang. Field-Programmable Gate Array*) to programowalne układy cyfrowe pozwalające na realizację zadań obliczeniowych poprzez odwzorowanie struktury matematycznej w macierzy składającej się z elementów logicznych. Powszechnie ich zastosowanie znajduje się w prototypowaniu zaawansowanych układów cyfrowych takich jak procesory i koprocesory arytmetyczne, sprzętowe układy wejścia/wyjścia, czy też procesory sygnałowe do obróbki dźwięku i obrazu. Strukturę połączeń logicznych dla takich układów opisuje się za pomocą języków opisu sprzętu takich jak Verilog czy też VHDL (*ang. Very High Speed Integrated Circuit Hardware Description Language*). Zwykle przy takim podejściu układ FPGA stanowi prototyp do budowy ostatecznego specjalizowanego układu ASIC (*ang. Application-Specific Integrated Circuit*).

Okazuje się, że obecnie narzędzia dla układów FPGA są na tyle elastyczne, iż możliwe jest zastosowanie układów FPGA w innych dyscyplinach naukowych, szczególnie tam gdzie zapis matematyczny nie jest możliwy do bezpośredniego prostego transferu do języków opisu sprzętu. Przykładem mogą być zagadnienia związane z mechaniką, automatyką, robotyką, czy też szeroko pojętą mechatroniką. Bardzo często w tych dyscyplinach modele matematyczne opisane są za pomocą skomplikowanych równań, często macierzowych, różniczkowych i modele te biorą bezpośredni udział w procesie sterowania. Dla takiego obiektu dynamicznego, wymaga się więc ciągłych wydajnych obliczeń w czasie rzeczywistym przy zagwarantowaniu obsługi wszelkich pobocznych zadań takich jak obsługa interfejsów wyjściowych, czy też systemów nadzoru.

W książce przedstawiono techniki opracowania, budowy i implementacji algorytmów do układu FPGA. Szczególnie opisano miejsca, gdzie istnieje potrzeba przetworzenia wydajnych operacji matematycznych opartych na arytmetyce zmiennoprzecinkowej. Pokazano zarówno drogę gdzie wykorzystano wbudowany procesor do układu FPGA, jak i przypadek gdzie projektant buduje własny dedykowany blok sprzętowy pozwalający na realizację zadań zmiennoprzecinkowych. Opisane rozwiązania można zastosować szczególnie w systemach sterowania ale też i innych wymagających obliczeń w czasie rzeczywistym.

Redaktor

Rozdział 1 – Platformy sprzętowe

1.1 Platformy sprzętowe w układach sterowania i regulacji

Platformy sprzętowe w zaawansowanych systemach sterowania i regulacji powinny zapewniać wysoką wydajność obliczeniową oraz umożliwiać współpracę z zewnętrznymi układami i urządzeniami peryferyjnymi, takimi jak: czujniki i przetworniki, sterowniki napędów oraz układy komunikacji i transmisji danych. Integracja obu cech w jednym układzie scalonym pozwala na uproszczenie budowy układu sterowania oraz wyeliminowanie zewnętrznych magistral wymiany danych, które wprowadzają dodatkowe opóźnienia oraz konieczność sprawdzania poprawności transmitowanych danych. Pożądanymi cechami platform sprzętowych w systemach sterowania i regulacji są:

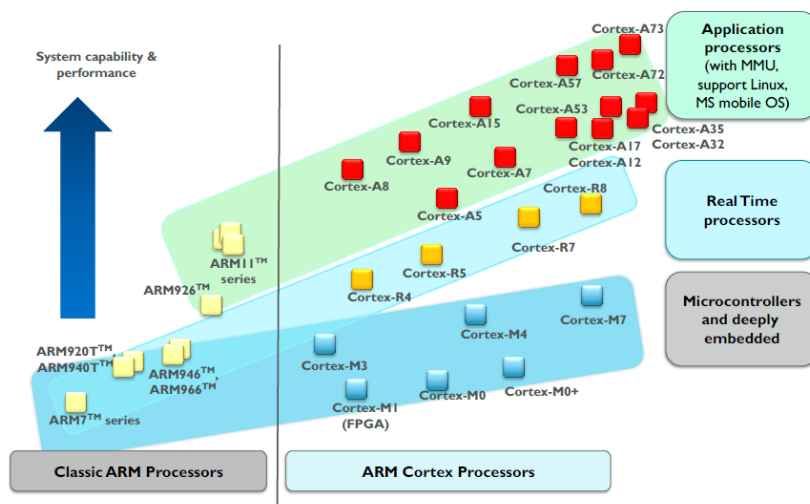
- możliwość implementacji algorytmów o wysokiej złożoności obliczeniowej;
- możliwość implementacji obliczeń na liczbach stałoprzecinkowych oraz zmiennoprzecinkowych;
- możliwość jednoczesnej realizacji wielu zadań z wysoką częstotliwością;
- duża liczba standardowych niskopoziomowych interfejsów do komunikacji z czujnikami i przetwornikami m.in.: SPI (*ang. Serial Peripheral Interface*), I2C (*ang. Inter Integrated Circuit*), UART (*ang. Universal Asynchronous Receiver Transmitter*);
- posiadanie pamięci o dużej pojemności do przechowywania danych;
- możliwość komunikacji z czujnikami i przetwornikami o różnych (również niestandardowych) interfejsach;
- możliwość wykorzystania wbudowanych i/lub zewnętrznych przetworników analogowo-cyfrowych ADC (*ang. Analog to Digital Converter*) i cyfrowo-analogowych DAC (*ang. Digital to Analog Converter*);
- możliwość komunikacji poprzez interfejs szeregowy, w celu odczytywania i zapisywania parametrów pracy układu sterowania;
- możliwość implementacji dowolnego rodzaju modulacji PWM (*ang. Pulse Width Modulation*);
- możliwość transmisji danych z dużą szybkością przez interfejs szeregowy;
- możliwość zachowania stałej, wysokiej częstotliwości pracy układu sterowania;
- duża liczba portów wejściowych i wyjściowych ogólnego przeznaczenia.

Posiadanie wyżej wymienionych cech przez platformę sprzętową pozwana na uzyskanie wydajnego układu sterowania oraz skrócenie czasu projektowania systemu.

1.2 Mikrokontrolery z rdzeniem ARM Cortex

Procesory z architekturą ARM Cortex są obecnie podstawową platformą sprzętową w aplikacjach automatyki, sterowania oraz mobilnych urządzeń elektronicznych. Występują w trzech wariantach przystosowanych do wykorzystania w aplikacjach (rys. 1.1) [1]:

- wymagających dużej mocy obliczeniowej (Cortex-Ax),
- czasu rzeczywistego (Cortex-Rx),
- ogólnego przeznaczenia oraz systemów wbudowanych (Cortex-Mx).



Rys. 1.1: Przegląd procesorów rodziny ARM Cortex [2]

Pierwszą grupą są 32 bitowe i 64 bitowe procesory przeznaczone dla aplikacji wymagających dużych mocy obliczeniowych (Cortex-Ax). Najczęściej pracują pod kontrolą systemów operacyjnych, takich jak Android, Linux lub Windows CE. Posiadają wysoką częstotliwość taktowania (powyżej 1 GHz) oraz rozszerzenia umożliwiające wirtualizację (np. implementację wirtualnej maszyny Javy). Wykorzystywane są najczęściej w komputerach przenośnych, telefonach komórkowych oraz tabletach [3][4].

Kolejną grupą są procesory dedykowane do aplikacji czasu rzeczywistego (Cortex-Rx). 32-biowe procesory przeznaczone do zadań, w których krytyczny jest czas wykonywania operacji. Oferują szereg funkcji wspomagających implementację aplikacji bezpieczeństwa i czasu rzeczywistego. Większość z tych procesorów nie posiada MMU tylko podstawowy moduł ochrony pamięci MPU (*ang. Memory Protection Unit*). Taktowane są zegarem o częstotliwości około kilkuset MHz (od 200 MHz do nieco powyżej 1 GHz). Umożliwiają wykorzystanie okrojonych wersji systemów operacyjnych (np. Linux) lub tzw. systemów operacyjnych czasu rzeczywistego RTOS (*ang. Real Time Operating System*). Wykorzystywane są

najczęściej w aplikacjach samochodowych (np. ABS, układ napędowy), kontrolerach dysku twardego oraz komunikacji bezprzewodowej [2][5].

Ostatnią grupą są procesory wykorzystywane jako mikrokontrolery (Cortex-Mx). Procesory 32-bitowe posiadające zintegrowaną pamięć oraz dużą liczbę układów peryferyjnych. Seria zoptymalizowana pod kątem niskiego zużycia energii oraz niskiej ceny [2]. Taktowane są zegarem o częstotliwości od kilkunastu do kilkuset megaherców. Stosowane są głównie jako mikrokontrolery ogólnego przeznaczenia i wykorzystywane w systemach wbudowanych, sterowania oraz automatyki [2].

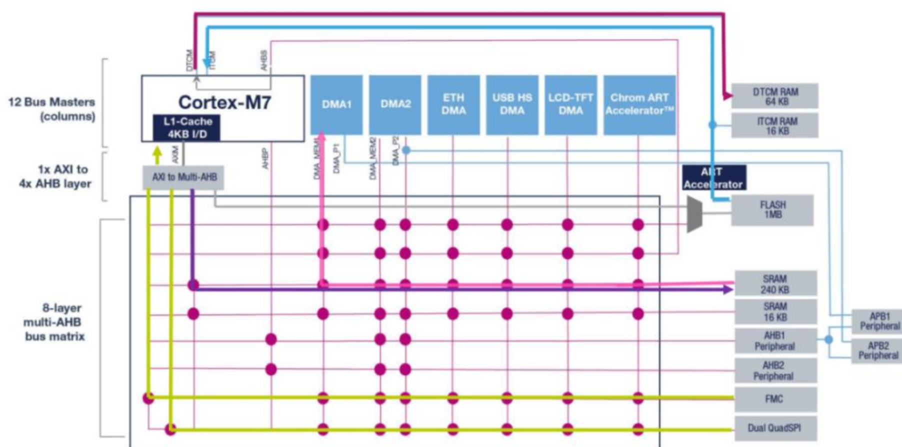
Podstawowymi mikrokontrolerami z rdzeniem Cortex są wersje o uproszczonej architekturze i małej wydajności obliczeniowej, oznaczone jako M0, M0+, M1. Mikrokontrolery bazujące na tych rdzeniach stanowią alternatywę dla układów 8-bitowych i mają je zastąpić w aplikacjach, które nie wymagają wysokiej wydajności obliczeniowej. Lista rozkazów tego rdzenia zawiera tylko podstawowe operacje arytmetyczne i logiczne, umożliwiające proste przetwarzanie danych. Wersja M1 jest natomiast zaprojektowana specjalnie z myślą o implementacji w układach typu FPGA.

Najczęściej wykorzystywaną wersją rdzenia Mx, która zyskała swoją popularność dzięki korzystnemu stosunkowi mocy obliczeniowej do ceny są mikrokontrolery z rdzeniem Cortex M3 [6]. Charakteryzują się one architekturą harwardzką (oddzielone magistrale pamięci programu i pamięci danych) oraz wydajnością na poziomie 1.25 DMIPS/MHz. Rdzeń M3 jest dedykowany dla mikrokontrolerów ogólnego przeznaczania i stosowany w aplikacjach wymagających wysokiej wydajności obliczeniowej na liczbach stałoprzecinkowych. Jedną z największych zalet tej architektury jest zdolność do obsługi instrukcji wykonujących operacje na zmiennych zarówno 16-bitowych jak i 32-bitowych. Lista rozkazów tego rdzenia jest rozszerzona o instrukcje pozwalające na przetwarzanie danych z wysoką wydajnością (np. *SDIV - Signed Divide*, *UDIV - Unsigned Divide*, *CPSIE - Enable Interrupts*) oraz atomowy dostęp do wybranych obszarów pamięci (*ang. bit-banding*). Umożliwia to zmianę pojedynczych bitów w pamięci danych lub rejestrach peryferyjnych poprzez wywołanie jednej instrukcji bez konieczności stosowania konwencjonalnej sekwencji: odczyt – modyfikacja - zapis. Rdzeń Cortex-M3 wyposażony jest w jednostkę ochrony pamięci MPU, która umożliwia definiowanie uprzywilejowanych uprawnień i atrybutów dostępu do różnych obszarów pamięci. Każdy dostęp do pamięci jest monitorowany przez MPU, która może zgłosić naruszenie (błąd) w przypadku próby nieautoryzowanego dostępu do danego obszaru pamięci. Jednostka ochrony pamięci pozwala m.in.: zapobiegać przepełnieniu stosu każdego z zadań, zabezpieczać regiony pamięci RAM/SRAM przed przypadkowym nadpisaniem definiując te regiony jako "tylko do odczytu", definiować obszary pamięci jako "współdzielone" dla wielu zadań, definiować obszary pamięci z których nie jest możliwe pobieranie instrukcji programowych

(rozkazów procesora), zapobiegając tym samym potencjalnym złośliwym oprogramowaniom. Obsługa przerwań w mikrokontrolerach z rdzeniem M3 realizowana jest poprzez wbudowany sprzętowy kontroler przerwań NVIC (*ang. Nested Vectored Interrupt Controller*). Celem zastosowania NVIC jest uproszczenie obsługi przerwań oraz skrócenie opóźnień wynikających z ich realizacji. Kontroler NVIC umożliwia: obsługę do 15 przerwań systemowych i 240 przerwań zewnętrznych, obsługę przerwań zagnieżdżonych, dynamiczne zmiany priorytetów, zmniejszenie opóźnień związanych z przerwaniami oraz ich maskowanie.

Dla aplikacji wymagających większej wydajności obliczeniowej powstał rdzeń Cortex M4 [7][8], który jest rozbudowany o instrukcje DSP oraz opcjonalnie jednostkę zmiennoprzecinkową pojedynczej precyzji FPU (wtedy oznaczany jako M4F). Podobnie jak wersja Cortex M3, charakteryzuje się prędkością wykonywania standardowych programów 1.25 MIPS/MHz, ale ze względu na wbudowaną 32-bitowa jednostkę obliczeniową MAC (*ang. Multiply and ACcumulate*) rdzeń Cortex-M4 zapewnia lepszą wydajność aplikacji realizujących cyfrowe przetwarzanie sygnałów. Wynika to z udostępnianych przez MAC wielu instrukcji mnożenia i sumowania sprzętowego wykonywanych w jednym cyklu pracy na liczbach: 16-bitowych, 32-bitowych oraz 64-bitowych. Jednostka MAC w jednym cyklu może wykonać m.in.: jedną operację mnożenia dwóch liczb 32-bitowych i sumowania wyniku z liczbą 64-bitową, dając w rezultacie liczbę 64-bitową lub dwie operacje mnożenia dwóch liczb 16-bitowych: 16×16. Rdzeń Cortex-M4 obsługuje również grupę instrukcji SIMD (*ang. Single Instruction, Multiple Data*), które pozwalają skrócić czas wykonywania operacji na wielu danych jednocześnie. W skład zestawu instrukcji SIMD wchodzą niektóre instrukcje DSP, do wykonywania takich operacji jak dodawanie, odejmowanie, mnożenie, mnożenie i sumowanie, które są wykorzystywane do implementacji powszechnie stosowanych operacji cyfrowego przetwarzania sygnałów, w tym filtrowania sygnałów cyfrowych za pomocą filtrów o skończonej odpowiedzi impulsowej FIR (*ang. Finite Impulse Response*) lub o nieskończonej odpowiedzi impulsowej IIR (*ang. Infinite Impulse Response*), obliczania szybkiej transformaty Fouriera FFT (*ang. Fast Fourier Transform*) na liczbach zespolonych oraz sumowania, odejmowania i mnożenia macierzy. Instrukcje SIMD umożliwiają (w jednym takcie zegara): wykonanie czterech operacji równoległych dodawania lub odejmowania liczb 8 bitowych oraz wykonanie dwóch operacji równoległych dodawania lub odejmowania liczb 16 bitowych. Kolejną jednostką wspomagającą obliczenia arytmetyczne jest koprocesor FPU, który służy do wykonywania operacji na liczbach zmiennoprzecinkowych pojedynczej precyzji i jest elementem stosowanym opcjonalnie w rdzeniach Cortex-M4. Zapewnia on obliczenia na liczbach zmiennoprzecinkowych, spełniając normy ANSI/IEEE STD 754-2008 oraz normy IEEE (*ang. Standard for Binary Arithmetic Floating-Point, IEEE 754*) [9]. FPU umożliwia realizację następujących operacji na liczbach zmiennoprzecinkowych pojedynczej precyzji: dodawanie, odejmowanie, mnożenie,

dzielenie, mnożenie i sumowanie oraz pierwiastkowanie. Zapewnia również konwersję formatów pomiędzy liczbami stałoprzecinkowymi a zmiennoprzecinkowymi (tzw. rzutowanie) oraz dostarcza instrukcje z wykorzystaniem stałych zmiennoprzecinkowych.



Rys. 1.2: Wewnętrzny interfejs komunikacyjny w rdzeniu ARM Cortex M7 [10]

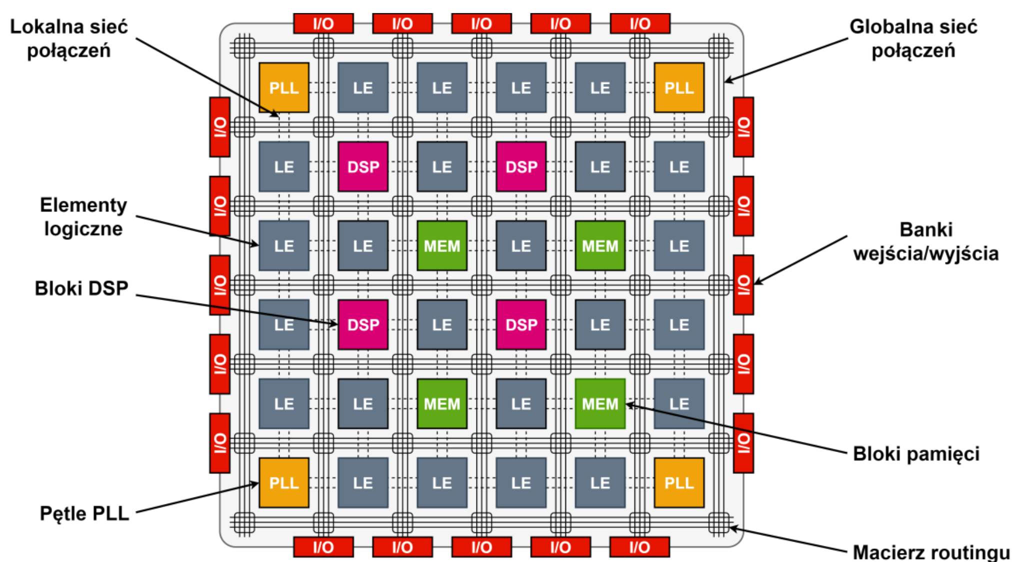
Najnowszą i najbardziej rozbudowaną wersją rdzeni Mx jest Cortex M7 [11] przeznaczony do implementacji złożonych algorytmów DSP. Dodatkowo jest wspomagany sprzętową jednostką zmiennoprzecinkową pojedynczej lub podwójnej precyzji. Wydajność nowego rdzenia wynosi 3.23 DMIPS/MHz, a lista rozkazów dodatkowo została rozszerzona o instrukcje do zaawansowanego przetwarzania sygnałów. Przystosowany jest do współpracy z wewnętrzną, szybką pamięcią SRAM TCM, wyposażoną w pamięci podręczne (*ang. cache)* dla danych i rozkazów, a także zaawansowany 6-poziomowy mechanizm przetwarzania potokowego z predykcją oraz sprzętowym wsparciem superskalarnego wykonywania programu.

Udoskonaleniem wprowadzonym w rdzeniu Cortex-M7 jest dedykowany interfejs komunikacyjny, który fizycznie jest konfigurowalną magistralą z lokalnymi kontrolerami (rys. 1.2). Interfejs ten wpływa na poprawę wypadkowej prędkość pracy mikrokontrolera poprzez działanie polegające na "splataniu" kilku kanałów magistrali w jeden, bardzo szybki kanał dwukierunkowej komunikacji rdzenia z wewnętrznymi układami peryferyjnymi (w poprzednich wersjach rdzenie komunikują się z otoczeniem za pomocą standardowych magistral). Rozwiązania zastosowane przez firmę ARM w rdzeniu Cortex-M7 pozwalają na szybszy niż w przypadku poprzednich wersji, dostęp rdzenia do zawartości pamięci SRAM i Flash, oczywiście przy założeniu, że pamięć umożliwia bezpośredni odczyt danych z wymaganą, wysoką częstotliwością (dla typowych pamięci około 70-90 MHz). Żeby uniknąć efektu tzw. "wąskiego gardła" w dostępie do zasobów pamięci Flash, producenci stosują różne mechanizmy, mające na celu zmniejszenie czasu odczytu i zapisu

danych. Jednym z nich jest pobieranie danych z pamięci i buforowanie za pomocą sprzętowego akceleratora ART (*ang. Adaptive Real-Time*), którego działanie polega m.in. na dekompozycji 128-bitowych słów przechowywanych w pamięci Flash na słowa 16-bitowe lub 32-bitowe. Są one następnie kolejkowane w lokalnej pamięci cache. Według informacji publikowanych przez producenta, mechanizmy usprawniające dostęp do zasobów pamięci Flash spowodowały wyeliminowanie konieczności używania podczas odczytu opóźnień, które istotnie zmniejszały rzeczywistą prędkość transferu danych [10].

1.3 Układy reprogramowalne

Układy FPGA (*ang. Field Programmable Gate Array*) to jeden z dwóch, obok CPLD (*ang. Complex Programmable Logic Devices*), obecnie produkowanych układów reprogramowalnych typu PLD (*ang. Programmable Logic Devices*) [12]. Układy te wywodzą się z prostych programowalnych układów elektronicznych pierwszej generacji typu PAL (*ang. Programmable Array Logic*), które były zbudowane z macierzy bramek typu *and* i *or* oraz programowalnych przełączników [13].



Rys. 1.3: Architektura układów FPGA

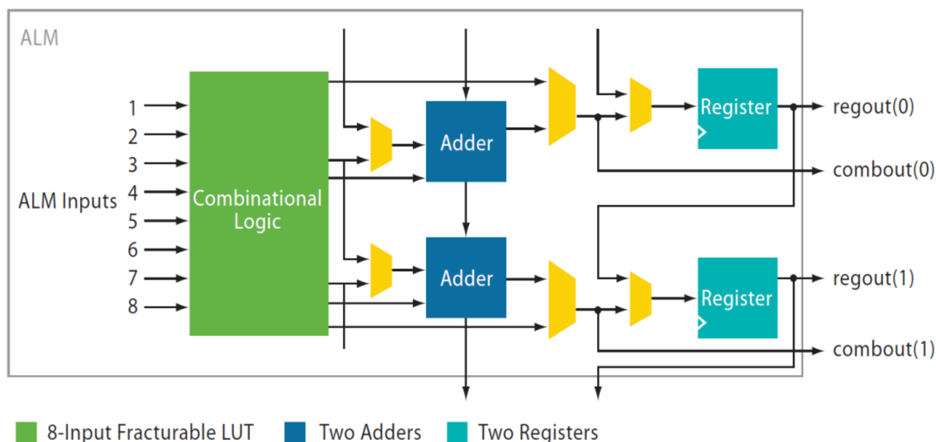
Układy FPGA początkowo stanowiły przede wszystkim alternatywę dla dedykowanych układów cyfrowych typu ASIC (*ang. Application Specific Integrated Circuit*). W przypadku małoseryjnej produkcji, FPGA są rozwiązaniem tańszym oraz pozwalają na skrócenie czasu projektowania i wprowadzania produktu na rynek [14]. Jednak główna zaleta tych układów, czyli elastyczność, jest również źródłem ich wad. Elastyczny charakter układów FPGA sprawia, że są one znacznie większe, wolniejsze

i bardziej energochłonne niż odpowiadające im układy ASIC. Wady te wynikają w dużej mierze z programowalnego trasowania połączeń wewnątrz urządzenia, które mogą obejmować nawet blisko 90% całkowitej powierzchni układu [14]. Jednak pomimo tych wad układy FPGA stanowią atrakcyjną alternatywę dla projektowania dedykowanych układów scalonych typu ASIC oraz mikrokontrolerów ogólnego przeznaczenia i DSP (*ang. Digital Signal Processor*).

	CPU	**FPGA**
Różnice	Stała architektura sprzętowa narzucona przez producenta	Architektura konfigurowana przez użytkownika
	Język programowania wykorzystywany jako sekwencyjnie wykonywane polecenia	Język programowania wykorzystywany do wewnętrznej konfigurowania układu
	Wykonywanie wielu zadań realizowane poprzez przełączanie się pomiędzy nimi	Zadania wykonywane jednocześnie i równolegle
	Porty dedykowane, niewielka możliwość zmiany funkcjonalności: np. PWM, I2C, SPI	W większości porty ogólnego przeznaczenia dowolnie konfigurowane
	Wbudowane przetworniki ADC i DAC	Najczęściej bez wbudowanych przetworników ADC i DAC
Podobieństwa	Układ cyfrowy w formie układu scalonego	
	Przeznaczony do realizacja zaimplementowanego algorytmu	
	Programowany z wykorzystaniem dedykowanego języka	

Tab.1.1: Podobieństwa i różnice pomiędzy układami mikroprocesorowymi a FPGA [12-19]

Układy FPGA nie posiadają stałej architektury, lecz są zbudowane z dużej liczby komórek logicznych (funkcjonalnie bramek logicznych), rejestrów, bloków pamięci oraz sieci połączeń (rys. 1.3). Układy reprogramowalne umożliwiają realizację operacji logicznych i arytmetycznych, pozwalają na przechowywanie zmiennych i przesyłanie danych oraz sygnałów pomiędzy różnymi częściami systemu [17]. Implementacja algorytmu w układzie FPGA najczęściej realizowana jest poprzez opis strukturalny lub behawioralny przy pomocy jednego z języków opisu sprzętu HDL (*ang. Hardware Description Language*). Jednak rezultatem kompilacji kodu źródłowego nie jest program w rozumieniu sekwencyjnie wykonywanych poleceń (jak w przypadku mikrokontrolera/mikroprocesora), lecz wewnętrzna konfiguracja urządzenia. [17]. W tabeli 1.1 przedstawione są podobieństwa i różnice pomiędzy układem FPGA a układem mikroprocesorowym.

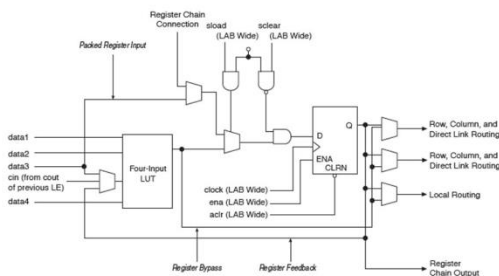


Rys. 1.4: Struktura elementu logicznego ALM układu Stratix II [18]

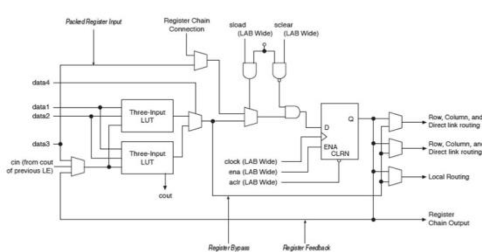
Układy FPGA zbudowane są z elementów, które możemy podzielić na trzy podstawowe kategorie [12][14][20]:

- komórki logiczne – pozwalające na implementację prostego (najczęściej 4-bitowego lub 6-bitowego) układu kombinacyjnego lub sekwencyjnego; w zależności od rodziny układu FPGA, komórki te mają różną budowę i poziom złożoności; najczęściej nazywa się je: *Logic Elements* (LE), *Adaptive Logic Module* (AML) lub *Configurable Logic Block* (CLB), [12][18]; schemat AML układu Stratix II widoczny jest na rys. 1.4;
- sieć połączeń – pozwala na zbudowanie bardziej złożonych struktur układów kombinacyjnych lub/i sekwencyjnych, poprzez łączenie wielu komórek logicznych w większe grupy; sieć połączeń najczęściej występuje na kilku poziomach: lokalnie – pozwalając na łączenie sąsiadujących ze sobą komórek logicznych, oraz globalnie – pozwalając na dystrybucję sygnałów w obrębie całego układu;
- pozostałe elementy architektury: bloki pamięci, wbudowane mnożarki (bloki DSP), pętle PLL, banki wejścia/wyjścia, itd. – dedykowane elementy wspomagające implementację złożonych systemów.

Altera Cyclone III
Logic Element (LE) – Normal Mode



Altera Cyclone III
Logic Element (LE) – Arithmetic Mode



Rys. 1.5: Element logiczny układu Cyclone III w trybie normalnym i arytmetycznym [15]

Architektura układów reprogramowalnych zostanie omówiona na przykładzie rodziny Cyclone, która stanowi niskobudżetową rodzinę układów FPGA firmy Intel (dawniej Altera) [15].

Komórki Logiczne

Najmniejszą jednostką w architekturze układu FPGA jest Element Logiczny. Każda komórka logiczna zawiera [20]:

- tablicę LUT (*ang. Look-Up Table*) – która pozwala na zaimplementowanie dowolnej funkcji logicznej (układu kombinacyjnego), najczęściej 4 lub 6 zmiennych (bitów); tablica LUT jest zwykle zbudowana z bitów pamięci typu SRAM, przeznaczonych do przechowywania maski konfiguracyjnej tablicy (CRAM) oraz zestawu multiplekserów do wyboru bitu CRAM, który ma sterować wyjściem [14][18];
- programowalny rejestr – może być skonfigurowany jako przerzutnik typu D, T, JK lub SR; każdy rejestr ma wejście danych, zegara, włączenia zegara i kasowania; w przypadku funkcji kombinowanych wyjście LUT omija rejestr i kieruje sygnał bezpośrednio do wyjścia LE [17];
- pozostałe elementy – przeznaczone do wewnętrznej konfiguracji LE oraz multipleksowania sygnałów w obrębie LE.

Każda komórka logiczna może pracować w jednym z kilku trybów. Kompilator automatycznie wybiera odpowiedni tryb konfiguracji LE dla typowych funkcji takich jak liczniki, sumatory, subtraktory czy funkcje arytmetyczne. Tryb normalny jest odpowiedni dla ogólnych aplikacji logicznych i funkcji kombinacyjnych. Tryb arytmetyczny jest dedykowany do implementacji sumatorów, liczników, akumulatorów i komparatorów. [17]. Sąsiadujące ze sobą LE tworzą bardziej złożone struktury (klastry) wewnątrz których sieć połączeń jest dużo bardziej rozbudowana – *Logic Array Blocks* (LAB).

Wewnętrzna sieć połączeń

Oprócz struktury bloków logicznych, kolejnym kluczowym elementem architektury układów FPGA jest wewnętrzna sieć połączeń [18]. Sieć ta, w większości rodzin układów zajmuje ponad 50% ich całkowitej powierzchni. Jest również odpowiedzialna w dużej mierze za czasy propagacji sygnałów (opóźnienia) [19] wewnątrz układu, co ma kluczowe znaczenie dla wydajności projektowanego systemu [13]. W układach FPGA sieć połączeń ma strukturę hierarchiczną, tzn. lokalna sieć połączeń odpowiadająca za dystrybucję sygnałów w obrębie sąsiadujących ze sobą komórek logicznych jest bardziej złożona niż sieć globalna odpowiadająca za dystrybucję sygnałów w obrębie całego układu. Jest to konsekwencją założenia, że projekty są z natury hierarchiczne: moduły wyższego poziomu tworzą instancje modułów niższego poziomu i łączą sygnały między nimi. Wymiana danych między modułami, które są blisko siebie w hierarchii projektowej jest większa, a układy FPGA mogą realizować te połączenia za pomocą krótkich tras, w obrębie małych obszarów układu scalonego [13]. Jednym z kluczowych elementów wewnętrznej sieci połączeń jest globalna sieć sygnałów zegarowych, która jest odpowiedzialna za dystrybucję sygnałów zegarowych do wszystkich zasobów układu (komórek logicznych, wbudowanych mnożarek, bloków pamięci, banków wejścia/wyjścia, itd.) [17].

Bloki pamięci

Pierwszą formą wbudowanych elementów pamięci w strukturze układu FPGA są rejestry zintegrowane w komórkach logicznych [13]. Rozwój układów oraz wykorzystanie ich w nowych aplikacjach, które wymagają dużej ilości pamięci do buforowania danych sprawił, że pożądane stało się posiadanie wewnętrznych bloków pamięci typu RAM [13]. Obecnie układy FPGA posiadają wbudowaną pamięć RAM w postaci rozproszonych bloków [17], do których dostęp jest realizowany poprzez wewnętrzną sieć połączeń [13]. Każdy blok pamięci może realizować różne typy pamięci: jednoportowe, dwuportowe, ROM, RAM, rejestry przesuwne, bufory FIFO (*ang. First In First Out*), [17] itd.

Wbudowane mnożarki

Układy FPGA posiadają wbudowane mnożarki (nazywane również blokami DSP) zoptymalizowane pod kątem funkcji cyfrowego przetwarzania sygnału DSP (*ang. Digital Signal Processing*). Najczęściej wykorzystywane są do implementacji: filtrów o skończonej odpowiedzi impulsowej (FIR), funkcji szybkiej transformaty Fouriera (FFT) i funkcji dyskretnej transformaty kosinusowej (DCT). Mnożarki mogą przyjmować jako argumenty liczby bez znaku jak i ze znakiem [17].

Bloki wejścia/wyjścia

Bloki wejścia/wyjścia połączone z portami GPIO (*ang. General Purpose Input/Output*) [17]:

- umożliwiają odczyt sygnałów wejściowych w odpowiednim standardzie napięciowym;
- umożliwiają generowanie sygnałów wyjściowych;
- pozwalają na współpracę z układami peryferyjnymi w standardach napięciowych m.in.: LVTTL (3.3 V, 2.5 V, 1.8 V), LVCMOS (3.3 V, 2.5 V, 1.8 V, 1.5 V), SSTL (klasy I, II), HSTL (klasy I, II), PCI, PCI-X;
- posiadają bufory 3-stanowe, pozwalające na implementację magistrali dwukierunkowej;
- pozwalają na wprowadzenie opóźnienia sygnałów wejściowych i wyjściowych;
- posiadają rezystory typu pull-up aktywowane podczas konfiguracji urządzenia lub konfigurowane przez użytkownika.

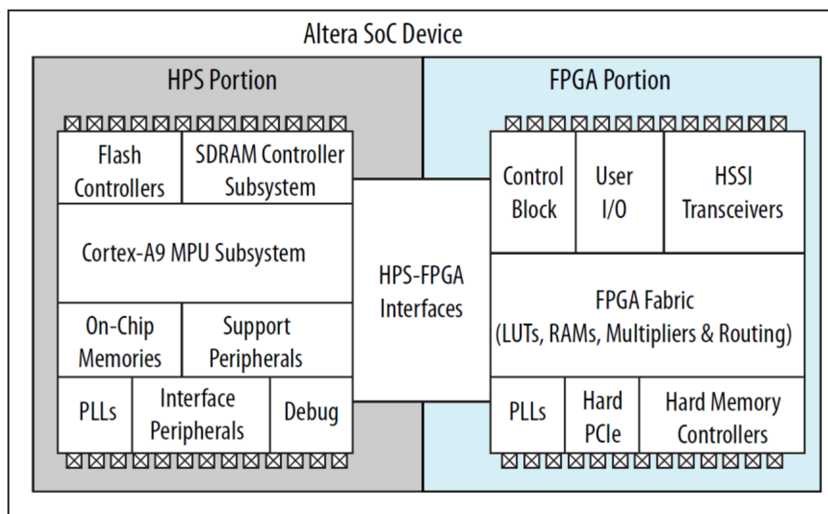
Współczesne układy FPGA obsługują szeroką gamę zewnętrznych interfejsów pamięci, takich jak pamięci zewnętrzne SDR SDRAM, DDR SDRAM, DDR2 SDRAM i QDRII SRAM. Banki wejścia/wyjścia są wyposażone w dedykowane szybkie interfejsy, które przesyłają dane między zewnętrznymi kośćmi pamięci z prędkością do 167 MHz/333 Mb/s dla urządzeń DDR i DDR2 SDRAM oraz 167 MHz/667 Mb/s dla QDRII SRAM [17].

1.4 Układy hybrydowe

Przez wiele lat układy mikroprocesorowe oraz FPGA rozwijały się niezależnie od siebie. FPGA najczęściej stanowiły alternatywę dla procesorów ogólnego przeznaczenia lub DSP, w przypadkach gdy wymagana była wysoka wydajność obliczeniowa lub równoległa realizacja wielu złożonych zadań. Od kilku lat można zauważyć nowy trend polegający na rozwoju układów hybrydowych, stanowiących połączenie układu mikroprocesorowego i FPGA w jednym układzie scalony. Cyclone V SoC stanowi przykład takiego hybrydowego układu składającego się z dwóch odrębnych części, mikroprocesorowej HPS (*ang. Hard Processor System*) oraz części FPGA, w jednym układzie scalonym. Architektura układu Cyclone V SoC widoczna jest na rys. 1.6.

Cześć HPS układu bazuje na dwurdzeniowym procesorze ARM z rdzeniem Cortex-A9, który zawiera m.in.: kontroler zewnętrznej pamięci Flash oraz SDRAM, wbudowaną pamięć RAM, układy peryferyjne oraz interfejsy do komunikacji z układami zewnętrznymi. Część FPGA stanowi typowy układ reprogramowalny składający się z elementów logicznych (AML), wbudowanej pamięci, bloków DSP oraz pętli PLL. Obie części układu (HPS oraz FPGA) posiadają swoje odrębne piny wejścia/wyjścia, nie są one współdzielone. Piny wejścia/wyjścia części FPGA są

kontrolowane poprzez zaimplementowane moduły sprzętowe, a piny części HPS są kontrolowane poprzez oprogramowanie uruchomione na procesorze [21].

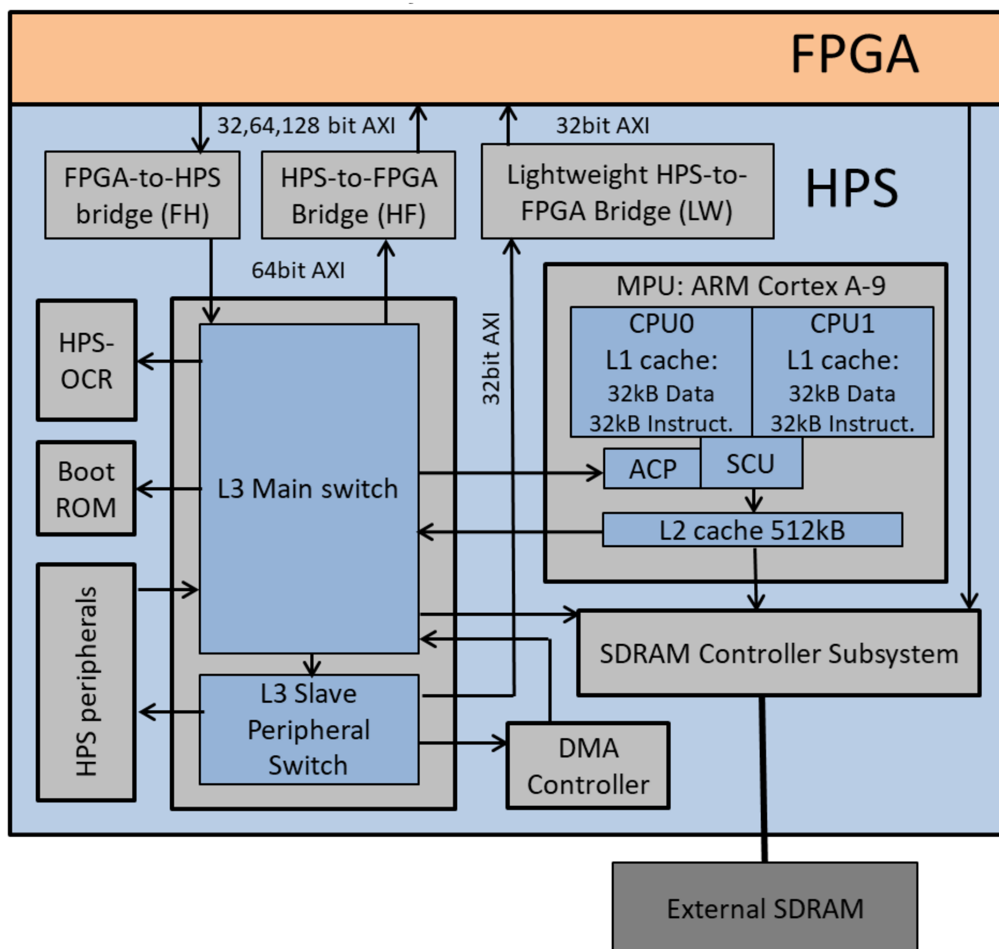


Rys. 1.6: Architektura układu Cyclone V SoC [21]

Części FPGA i HPS posiadają również oddzielne, niezależnie zasilania. Możliwe jest włączenie części HPS bez zasilania części FPGA. Jednak aby włączyć część FPGA układu, należy podać zasilanie do części HPS. Oczywiście najczęściej uruchomione są obie części systemu [21].

Na rys. 1.7 przedstawiona jest architektura części HPS układu, a poniżej znajdują się najważniejsze jego cechy [21]:

- dwurdzeniowy procesor ARM Cortex-A9 (o częstotliwości taktowania 800-925 MHz);
- dwa kontrolery Ethernet 10/100/1000 Mbps EMAC (*ang. Ethernet Media Access Controllers*);
- dwa kontrolery USB 2.0 OTG (*ang. On-The-Go*);
- kontroler DMA (*ang. Direct Memory Access*) ogólnego przeznaczenia;
- kontroler pamięci NAND Flash;
- kontroler karty pamięci SD/MMC (*ang. Secure Digital / MultiMediaCard*);
- cztery interfejsy SPI (*ang. Serial Peripheral Interface*);
- interfejs QSPI (Quad SPI);
- cztery interfejsy I2C (ang. Inter-Integrated Circuit);
- dwa interfejsy UART (*ang. Universal Asynchronous Receiver-Transmitter*);
- dwa interfejsy CAN (*ang. Controller Area Network*);
- cztery układy licznikowe (*ang. Timer*);
- porty wejścia/wyjścia ogólnego przeznaczenia.



Rys. 1.7 Architektura części HPS systemu Cylone V SoC [git hub]

Największą zaletą układu hybrydowego jest możliwość współpracy oraz wymiany danych pomiędzy obiema częściami. W tym celu zostało zaprojektowanych kilka interfejsów pozwalających na komunikację HPS↔FPGA. Najważniejsze z nich przedstawione są poniżej [21][22].

- Mostek FPGA-HPS (*ang. FPGA-to-HPS bridge*) – jednokierunkowa magistrala, o dużej przepustowości danych, z konfigurowalną szerokością (32, 64 lub 128 bitów). Pozwala części FPGA na przesyłanie danych do części HPS. Ten interfejs umożliwia strukturze FPGA pełny wgląd w przestrzeń adresową części HPS.
- Mostek HPS-FPGA (*ang. HPS-to-FPGA bridge*) – jednokierunkowa magistrala, o dużej przepustowości danych, z konfigurowalną szerokością

(32, 64 lub 128 bitów). Pozwala procesorowi HPS na przesyłanie danych do części FPGA.

- Lekki mostek HPS-FPGA (*ang. Lightweight HPS-to-FPGA bridge*) – dwukierunkowa magistrala o szerokości 32-bitów. Pozwala procesorowi na zapis i odczyt parametrów we wspólnej przestrzeni adresowej HPS-FPGA.

1.5. Bibliografia

[1] Grzegorz Góra: *Sterowanie napędami bezpośrednimi – rozprawa doktorska*. Akademia Górniczo-Hutnicza; Kraków; 2019.

[2] J. Yiu: *ARM Cortex-M for Beginners, An overview of the ARM Cortex-M processor family and comparison*. ARM; 2016.

[3] ARM (*http://www.arm.com*): *ARM Cortex-A Series, Programmer's Guide for ARMv8-A, Version: 1.0, ARM DEN0024A, ID050815*. 2015.

[4] K. Paprocki: *Mikrokontrolery STM32 w praktyce*. Legionowo: Wydawnictwo BTC; ISBN 978-83-60233-52-8; 2011.

[5] S. Craske: *ARM Cortex-R Architecture, For Integrated Control and Safety Applications*. ARM; 2013.

[6] STM, Cortex M3 [on-line]: *http://www.st.com/content/st_com/en/products/ microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f1-series/stm32f103/ stm32f103rc.html*. Pobrane 2018.

[7] ARM, Cortex-M4 [on-line*]: https://www.arm.com/products/processors/ cortex-m/cortex-m4-processor.php*. Pobrane 2018.

[8] STM, Cortex M4 [on-line]: *http://www.st.com/content/st_com/en/products/ microcontrollers/stm32-32-bit-arm-cortex-mcus/stm32f4-series.html*. Pobrane 2018.

[9] IEEE Standard for Floating-Point Arithmetic [on-line]: *http://www.dsc.ufcg.edu.br/~cnum/modulos/Modulo2/IEEE754_2008.pdf*. Pobrane 2018.

[10] A. Gawryluk [on-line]: *http://mikrokontroler.pl/2015/01/21/stm32f7-z-cortex-m7-nowe-mikrokontrolery-w-rodzinie-stm32/*. Pobrane 09-04-2018.

[11] ARM, Cotex M7 [on-line]: *https://developer.arm.com/ip-products/ processors/cortex-m/cortex-m7*. Pobrane 2019.

[12] Jacek Majewski, Piotr Zbysiński: *Układy FPGA w przykładach*. Wydawnictwo BTC; ISBN: 978-83-60233-23-8; Legionowo; 2007.

[13] Andrew Boutros, Vaughn Betz: *FPGA Architecture: Principles and Progression*. IEEE CIRCUITS AND SYSTEMS MAGAZINE; Digital Object Identifier 10.1109/MCAS.2021.3071607; 2021.

[14] Rahul Dubey: *Introduction to Embedded System Design Using Field Programmable Gate Arrays*. Wydawnictwo Springer; ISBN 978-1-84882-015-9; 2009.

[15] All about FPGA [on-line]: *https://allaboutfpga.com*.

[16] Intel FPGA [on-line]: *https://www.intel.pl*.
[17] Ognjen Šcekic: *FPGA Comparative Analysis*. University of Belgrade; ETF – School of Electrical Engineering; Belgrade; 2005.
[18] Altera: *White Paper – FPGA Architecture*. Ver. 1.0; 2006.
[19] C. Chiasson, V. Betz: *Should FPGAs abandon the pass gate?* in Proc. Int. Conf. Field-Programmable Logic Appl., pp. 1–8; 2013.
[20] Umer Farooq, Zied Marrakchi, Habib Mehrez: *Tree-based Heterogeneous FPGA Architectures*. Wydawca Springer; ISBN: 978-1-4614-3594-5; 2012.
[21] Intel (Altera): *Cyclone V Hard Processor System Technical Reference Manual*. 2022.
[22] Rihards Novickis, Modris Greitans: *FPGA Master based on chip communications architecture for Cyclone V SoC running Linux*. 5th International Conference on Control, Decision and Information Technologies (CoDIT); 2018.
[23] CycloneVSoC-time-measurements [on-line]: *https://github.com/UviDTE-FPSoC/CycloneVSoC-time-measurements*. Pobrane 05-2023.

Rozdział 2 – Implementacja w układzie FPGA

2.1 Języki opisu sprzętu

Technologia projektowania układów cyfrowych za pomocą opisu logicznego, w poszczególnych dekadach, uległa wielu zmianom. Początkowo w latach 70 i 80 XX w., wykorzystywano proste schematy oparte na symbolach. Od połowy lat 80 i 90 do projektowania układów cyfrowych zaczęto wykorzystywać języki opisu sprzętu – HDL (*ang. Hardware Design Language*). Pierwotnie języki te stosowane były w narzędziach wspierających symulację logiki cyfrowej [1].
Popularyzacja używania HDL do symulacji układów spowodowała, że producenci układów programowalnych opracowali nowe narzędzia przeznaczone do syntezy, w wyniku czego HDL stał się metodą sprzętowego opisu w projektowaniu cyfrowym.
Obecnie języki programowania HDL przeznaczone są do funkcjonalnego opisu sprzętowego układów cyfrowych takich jak: układy FPGA oraz CPLD.

2.1.1 Przegląd języków programowania

Najpopularniejszymi językami HDL są: Verilog i VHDL (*ang. Very High Speed Integrated Circuit Hardware Description Language*), które zostały opracowane w drugiej połowie lat 80 XX w. Obydwa języki są zdefiniowane przez standardy IEEE.

Verilog został stworzony przez przedsiębiorstwo *Gateway Design Automation*, a następnie rozwinięty przez firmę *Cadence Design Systems* [2]. Język ten jest zdefiniowany przez ogólny standard IEEE 1364. Wieloletnie prace, prowadzone nad rozszerzeniem funkcjonalności języka Verilog, spowodowały, że powstały wersje: IEEE 1364-1995[3] i IEEE 1364-2001[4] odnoszące się do Verilog-95 i Verilog-2001.

Język VHDL natomiast został stworzony przez inicjatywę Departamentu Obrony Stanów Zjednoczonych (*ang. U.S. Department of Defense initiative*) w ramach prac nad VHSIC - *Very High-Speed Integrated Circuit*, które miały przede wszystkim integrować i skorelować wyniki symulacji obwodów cyfrowych różnych dostawców sprzętu. Standard języka VHDL oficjalnie powstał w 1987 r. jako: *IEEE std. 1076*. Podobnie jak w przypadku języka Verilog, na przestrzeni lat ulegał on zmianom, w tym rozszerzeniu funkcjonalnemu, w wyniku czego powstało kilka wersji, np.: *IEEE 1076-1993 [5], IEEE 1076-2008 [6], IEEE 1076-2019 [7].*

Oba języki mają swoje unikalne cechy i są używane w różnych zastosowaniach. Verilog jest bardziej popularny w przemyśle i powszechnie stosowany w projektowaniu układów FPGA, natomiast VHDL jest częściej

wykorzystywany w projektowaniu systemów wbudowanych i układów ASIC (*ang. Application-Specific Integrated Circuit*).

Do grupy języków programowania sprzętowego, można zaliczyć również:

- ABEL (*ang. Advanced Boolean Expression Language*) - umożliwia projektantom opisanie funkcjonalności układów w postaci wyrażeń boolowskich, które następnie są przetwarzane przez narzędzia CAD (*ang. Computer-Aided Design*) w celu wygenerowania schematów układów scalonych.
- SystemVerilog - jest rozszerzeniem języka Verilog pozwalającym na: testowanie, modelowanie behawioralne i systemowe oraz programowanie orientowane obiektowo.
- Vhdl-AMS - to rozszerzenie języka VHDL, który dodaje funkcje modelowania analogowego i mieszanej symulacji.
- AHDL (*ang. Altera Hardware Description Language*) – jest typem języka z grupy HDL zawierającym rozszerzenia ułatwiające projektowanie z wykorzystaniem układów FPGA/CPLD firmy Altera (obecnie Intel-Altera). Na przykład, w AHDL można bezpośrednio odwoływać się do zasobów dedykowanych elementów układów (np. blokach DSP, Memory M9K), co pozwala na szybsze i bardziej efektywne projektowanie.

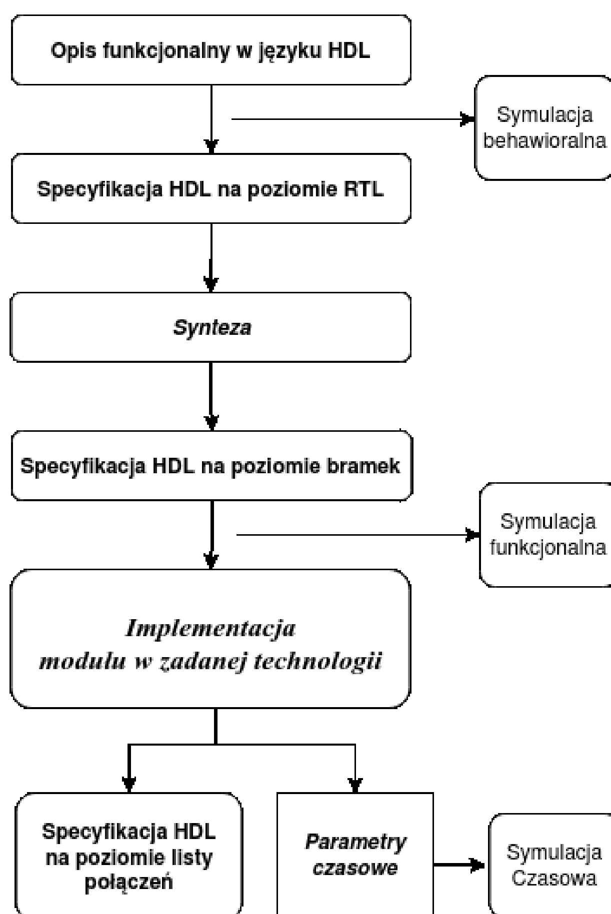
  AHDL nie jest już aktywnie rozwijany przez firmę Intel-Altera, został zastąpiony przez język VHDL i Verilog jako preferowane języki projektowania platform FPGA/CPLD.

  Stale jednak można stosować makra-programowe, które wykorzystują składnia języka AHDL.
- SystemC - to język programowania, który został opracowany jako narzędzie do projektowania i modelowania systemów elektronicznych na poziomie wysokiego poziomu abstrakcji. Wykorzystuje on zmodyfikowaną składnię języka ANSI C. Jego główną zaletą jest prosta konwersja kodu programu (opisu) do postaci maszyny stanów.
- Bluespec - to język programowania, który wykorzystuje modelowanie zachowań cyfrowych przy użyciu automatów stanów i transakcyjnych semantyk.

Każdy z tych języków ma swoje unikalne cechy i zalety, które pozwalają projektantom układów cyfrowych na efektywne projektowanie i testowanie układów reprogramowalnych. Układy te obecnie wykorzystywane są już nie tylko jako platformy sprzętowe do opracowywania nowych struktur układów cyfrowych, w procesie opracowywania funkcjonalno-logicznej architektury układów ASIC, ale również stanowią główne układy cyfrowe w złożonych architekturach systemów w komercyjnym zastosowaniu. Możliwość wielokrotnej rekonfiguracji pozwoliła na modyfikowanie i rozszerzanie funkcjonalności.

2.1.2 Projektowanie modułów sprzętowych przy użyciu języków HDL

W procesie projektowania architektury sprzętowo-programowej dla układów re-programowalnych, np. FPGA, CPLD, można wyszczególnić kolejne etapy projektowania rys. 2.1. Weryfikacja projektu w poszczególnych etapach jest kluczowa do osiągnięcia ogólnych założeń funkcjonalnych systemu. Nie spełnienie warunków modelowania może spowodować zarówno błędy w działaniu pojedynczego modułu (układu), ale również, w wyniku procesu syntezy, przyczynić się do nieprawidłowego działania całego systemu [8,9].



Rys. 2.1: Schemat blokowy procesu projektowania modułów sprzętowych w językach programowania typu HDL

W opisie funkcjonalnym dokonywany jest zapis programowy bloku sprzętowego, za pomocą wybranego języka sprzętowego np. VHDL lub Verilog, na

podstawie specyfikacji modułu, sporządzonej na etapie projektowania architektury systemu (opis na wysokim poziomie abstrakcji). W kodzie programu, zapisanym w języku sprzętowym, możemy wyszczególnić fragmenty, takie jak: interfejs, elementy prekompilacji, ciało modułu rys.2 .2.

Należy zaznaczyć, że w programowaniu przy użyciu języków sprzętowych, deklarowanie interfejsu układu odbywa się najczęściej na początku programu i łatwo go wyszczególnić od pozostałych części zapisu. Liczba i typ wejść/wyjść bloku sprzętowego, może być rozszerzona o dodatkowe porty pozwalające na monitorowanie parametrów podczas testów.

Dobrą praktyką, w projektowaniu sprzętowym jest sporządzenie schematu blokowego w postaci graficznej, która w czytelny sposób przedstawia budowę interfejsu opracowywanego modułu. Obecnie, producenci narzędzi programistycznych (np. Quartus Prime, Intel-Altera) zapewniają możliwość tworzenia reprezentacji graficznych – bloków CAD, na podstawie opisu funkcjonalnego, a następnie dodawania ich do bibliotek sprzętowych projektu (plików typu blok-diagram, *bdf).

Definicja modułu *(nazwa_modulu)*
Interfejs *(deklaracja parametrów)* *(deklaracja portów)*
Elementy prekompilacji *dyrektywa 'include* *makra - odwołanie do ust komp.*
Ciało modułu *deklaracja zmiennych, przypisanie ciągłe instancja podrzędnych modułów, funkcje, instrukcje, taski*
koniec definicji modułu *endmodule*

Rys. 2.2: Uproszczony szablon modułu sprzętowego, język HDL: Verilog

Na etapie opisu funkcjonalnego programista jest w stanie, w pewnym stopniu, ingerować w opis sprzętu na poziomie RTL. Za przykład może posłużyć wykorzystanie podstawowej funkcji „*always*” w składni języka: SystemVerilog, dla kompilatora: Quartus Prime. Wykorzystanie funkcji systemowej „*always_ff*”

spowoduje, że podczas kompilacji wykorzystane zostaną zasoby w postaci elementów sekwencyjnych (np. przerzutników typu flip-flop), natomiast dla zastosowania funkcji „*always_latch*” przerzutniki typu zatrzask. W wyniku użycia funkcji: „*always_comb*”, do budowy modułu sprzętowego na poziomie RTL, powinny być wykorzystane zasoby podstawowych elementów kombinacyjnych (np. bramki logiczne).

Opracowanie opisu na poziomie RTL jest bardzo istotne w procesie projektowania, ze względu na możliwość sporządzenia modułu, który może być wykorzystywany w wielu projektach dla różnych rodzin układów re-programowalnych.

Dlatego należy zwrócić szczególną uwagę, w jaki sposób zostaną zinterpretowane poszczególne konstrukcje języka podczas syntezy, ponieważ w znacznym stopniu wpływa to, na działanie układu (spełnienie wymagań funkcjonalnych i czasowych) oraz wykorzystanie zasobów platform sprzętowych.

Kolejnym etapem procesu projektowania jest synteza modelu, której rezultatem jest strukturalny opis układu składający się z listy połączeń podstawowych elementów logicznych dostępnych w wybranej technologii. Na tym etapie zalecane jest przeprowadzenie symulacji funkcjonalnej, której wyniki, pozwalają na sprawdzenie zgodności otrzymanej struktury logicznej z modelem behawioralnym. Do przeprowadzenia symulacji funkcjonalnej wykorzystywane są najczęściej programy: ModelSim lub Questa.

Ostatnim etapem procesu jest implementacja modułu w zadanej technologii oraz jego symulacja na poziomie połączeń, wewnątrz struktury. Jednocześnie, na podstawie charakterystycznych parametrów układu, przeprowadzana jest symulacja czasowa. W rezultacie wyliczane są czasy propagacji poszczególnych elementów oraz połączeń pomiędzy nimi.

2.2 Quartus Prime

Quartus Prime [10], firmy Intel (dawniej Altera) stanowi zespół narzędzi służących do projektowania i implementacji struktury układów reprogramowalnych. Do podstawowych operacji, które umożliwia pakiet narzędzi programistycznych Quartus Prime można zaliczyć: kompilację, symulację i programowanie struktury PLD (układów FPGA i CPLD). Podstawowym plikiem projektu tzw. plik top-level, zawiera opis funkcjonalny układu oraz jego połączenia z otoczeniem, w formacie programu w języku HDL (plik z rozszerzeniem np.: .vhd, .sv, .v, .vlg) lub plik graficzny - schemat (plik z rozszerzeniem .bdf). Do symulacji czasowej i funkcjonalnej modeli sprzętowych opisanych w języku programowym HDL służy narzędzie Waferom Editor [11] lub ModelSim-Altera [11]. Pozwala na opracowanie modelu symulacyjno-funkcjonalnego oraz weryfikację funkcjonalną w projektowaniu klasycznym układów PLD (*ang. Programmable Logic Device*). Za pomocą Quartus Prime (Compile Design) [11], analiza i synteza modeli sprzętowych, opisanych na kolejnych poziomach abstrakcji, (klasyczne projektowanie układów PLD) jest realizowana automatycznie. Projektant może podglądać (lub też ingerować), reprezentatywny model będący wynikiem analizy i syntezy w postaci graficznej. Do weryfikacji modelu na różnym poziomie opisu, służą odpowiednie narzędzia: RTL Viewer - na poziomie opisu RTL - Techology map [11], a na poziomie podstawowych elementów logiki bramek/przerzutników - Viewer/Post-Fitting [11]. Dodatkowym narzędziem służącym do analizy czasowej jest Timing Analizer [11], pozwalający na optymalizację czasową modeli na poziomie weryfikacji logicznej list połączeń.

Dla zintegrowanego projektowania sprzętu i oprogramowania, firma Intel udostępnia Platform Designer [12] lub Qsys [13] - jako narzędzia do specyfikacji sprzętowej systemu oraz Nios II Software Build Tools - jako narzędzia do specyfikacji programowej systemu z wykorzystaniem procesora Nios II. Pierwszy pakiet narzędzi zawiera biblioteki bloków sprzętowych - ip-core, (sprzętowe modele opisane w języku HDL) pozwalające na implementację układów cyfrowych o określonym przeznaczeniu, które są powszechnymi elementami - układami peryferyjnymi, architektury mikrokontrolerów. Wśród nich można wyszczególnić: układy do obsługi interfejsów pamięci zewnętrznych (np. SRAM) i wewnętrznych (np. On-chip memory), portów ogólnego przeznaczenia (np. I/O parallel ports), jednostki procesorowe typu soft-core (np. Nios II). W przypadku zastosowania systemu opartego na procesorze Nios II, wewnętrzną integrację systemu zapewnia magistrala producenta - Avalon [14]. Producent umożliwia również rozszerzenie listy bibliotek sprzętowych ip-core, o komponenty użytkownika. Pozwala to projektantowi na opis własnych bloków sprzętowych i dodania ich do listy elementów specyfikacji sprzętowej systemu. Inną, dodatkową funkcją, jest rozszerzenie listy rozkazów procesora Nios II, o dodatkowe instrukcje koprocesorowe, realizowane przez

zaprojektowane bloki sprzętowe użytkownika, które stanowią moduły wykonawcze w architekturze procesora. Układy takie przeznaczone są do realizacji określonych operacji/zadań (np. instrukcja koprocesorowa realizujące sprzętowe mnożenie liczb typu zmiennoprzecinkowego). Nios II Software Build Tools [12], jest pakietem narzędzi, pracujących w środowisku Eclipse [15], służących do kompilacji (bsp-compiler), debugowania i programowania, przeznaczonych dla procesora Nios II. Program źródłowy dla kompilatora, może być zapisany w języku C lub C++ (przy czym wyższy poziom optymalizacji kodu uzyskuje się dla języka C). Istotną funkcją jest konfiguracja kompilatora, pozwalająca w różnym stopniu na optymalizację kodu. Uzyskanie najwyższego poziomu optymalizacji czasowej i zminimalizowanie pojemności programowej w pamięci, otrzymuje się poprzez ustawienie poziomu optymalizacji kodu na "3" (Optimization level - 'o3'). Dodatkowo można zredukować zawartość bibliotek sprzętowych HAL po przez zaznaczenie opcji: reduce device driver [12].

2.3 Narzędzia do projektowania systemu metodą zintegrowaną

Jednym z podejść projektowania architektury systemu typu SoPC (*ang. System on Programmable Chip*), jest budowa systemu w oparciu o gotowe moduły sprzętowe (tzw. IP-core), udostępnione przez producentów układów FPGA w postaci bibliotek programowych. Narzędzia programistyczne do projektowania takiej architektury, pozwalają na szybką budowę i implementację struktury sprzętowej. Powszechnie wykorzystywane architektury są oparte o architekturę mikrokontolera z procesorem typu soft-core, z możliwością zapisu kodu programu wykonawczego w standardowych językach ANSI C/C++. Podstawowymi elementami takiej architektury są: pamięć, procesor, interfejs wejściowy i wyjściowy. Komunikacja pomiędzy poszczególnym elementami systemu realizowana jest przez wewnętrzną magistralę systemową. Dodatkową cechą systemu z wykorzystaniem bibliotek ip-core jest możliwość rekonfiguracji podstawowych parametrów elementów systemów, np. rozmiarów (pojemności) pamięci, szerokości interfejsu wej./wyj., modyfikowania typu i ilości instrukcji koprocesora.

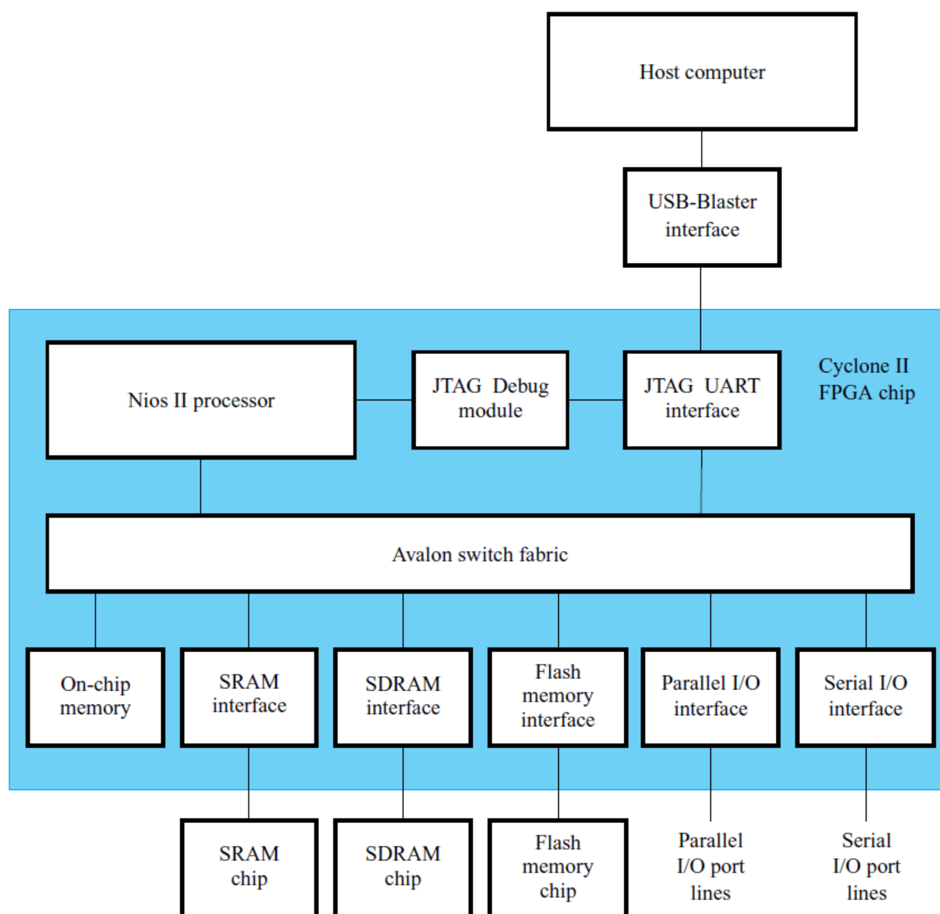
Architektura sprzętowa systemu SoPC oparta na komponentach ip-core, standardowych bibliotek producenta, jest uniwersalna i łatwa w konfiguracji, jednak posiada ograniczenia dotyczące wydajności systemu oraz ilości typów modułów peryferyjnych (funkcji realizowanych sprzętowo). Dlatego producenci narzędzi do budowy takich systemów, umożliwiają wykorzystanie wolnych zasobów układu FPGA do rozbudowy ich architektury o dedykowane bloki sprzętowe użytkownika. Moduły te najczęściej są opisywane na poziomie programowym w językach typu HDL i powinny być wyposażone w interfejs zgodny z opisem wewnętrznej magistrali systemu, udostępnionych przez producenta układów lub/i narzędzi programistycznych. Wśród producentów układów FPGA którzy udostępniają

narzędzia programistyczne o największym wachlarzu możliwości budowy systemów SOPC należą firmy: Xlinx - BSB Base System Builder [16] i Intel – Platform Designer (dawniej SoPC Builder lub QSYS) [12]. Ze względu na przeznaczenie (typ realizowanych zadań) sprzętowe bloki użytkownika mogą być podzielone na bloki sprzętowe będące rozszerzeniem instrukcji procesora (instrukcje koprocesorowe), lub bloki sprzętowe będące rozszerzeniem układów peryferyjnych struktury mikroprocesorowej. Przykładem pierwszego typu bloków sprzętowych może być instrukcja realizująca funkcję trygonometryczną, sinus kąta - sin(), której rezultat zwracany jest bezpośrednio do rejestru (wewnętrznej pamięci systemu). Bloki sprzętowe będące rozszerzeniem listy standardowych instrukcji procesora w systemie, pełnią rolę układów - akceleratorów funkcji potrzebnych do realizacji algorytmu zapisanego w specyfikacji programowej procesora (listy rozkazów). Natomiast przykładem drugiego typu bloków sprzętowych, może być układ realizujący generowanie sygnału PWM, którego parametry wejściowe funkcji (np. okres i wypełnienie) są zadawane poprzez wywołanie instrukcji procesora, natomiast rezultat (sygnał wyjściowy), podawany jest bezpośrednio na port wyjściowy systemu cyfrowego. Bloki sprzętowe będące rozszerzeniem układów peryferyjnych, pełnią funkcję akceleratorów systemu pod względem realizacji zadań, których przeniesie do algorytm oprogramowania (kodu wykonawczego procesora) jest niekorzystne pod względem spadku wydajności systemu lub/i zasobów układu.

2.4 Procesor Nios II

Do budowy architektury sprzętowo-programowej, może zostać wykorzystany soft-procesor Nios II firmy Intel. Nios II jest 32-bitowym procesorem o architekturze typu RISC (*ang. Reduced Instruction Set Computing*), z możliwością rozszerzenia podstawowej listy instrukcji dostarczonej przez producenta o 256 dodatkowych instrukcji użytkownika. System soft-procesorowy składa się z rdzenia procesora, kontrolerów pamięci wewnętrznej lub/i zewnętrznej, zestawu układów peryferyjnych oraz portów wejścia/wyjścia ogólnego przeznaczenia (rys. 2.3). Architektura jednostki projektowana przez użytkownika pod kątem wymagań danej aplikacji, głównie z elementów dostarczonych przez producenta (m.in.: kontroler pamięci, jednostka zmiennoprzecinkowa FPU, liczniki, moduły obsługi interfejsów SPI, I2C, UART, itd.) oraz modułów zaimplementowanych przez użytkownika w jednym z języków HDL [18][19]. Komunikacja rdzenia procesora z innymi elementami architektury realizowana jest poprzez wewnętrzną magistralę o nazwie Avalone [14]. Rdzeń procesora występuje w dwóch wersjach [18]:

- Nios II/e – economy – podstawowy rdzeń z rodziny Nios, wykorzystuje mało zasobów sprzętowych układu FPGA, jest dostępny za darmo bez ograniczeń.
- Nios II /f – fast – bardziej rozbudowany i szybszy rdzeń, wykorzystuje więcej zasobów sprzętowych, przeznaczony do wykorzystania komercyjnego.

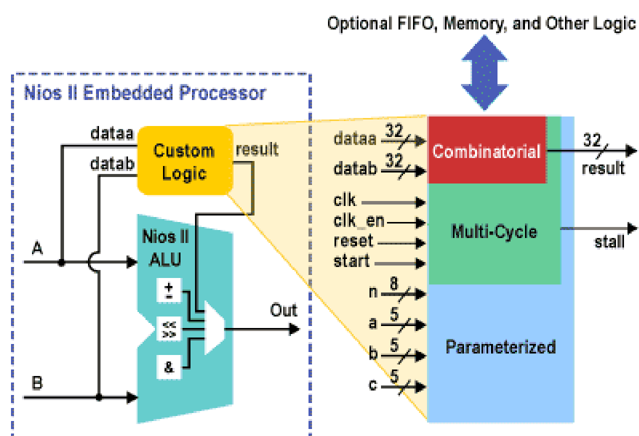


Rys. 2.3: Architektura procesora Nios II [19]

Największą zaletą wykorzystania układu soft-procesorowego w stosunku do standardowych mikrokontrolerów jest jego elastyczna architektura, która może być modyfikowana na potrzeby danej aplikacji. Projektant sam decyduje o rozmiarze i rodzaju użytej pamięci, liczbie portów wejścia/wyjścia ogólnego przeznaczenia, a także zestawie wykorzystanych modułów peryferyjnych. Dzięki temu, dodając lub usuwając moduły, każdy system może osiągnąć wysoką wydajność i funkcjonalność przy odpowiednim wykorzystaniu dostępnych zasobów sprzętowych. Można także wyeliminować nieużywane funkcje procesora i układów peryferyjnych, tak aby dopasować system do tańszego układu o mniejszej ilości dostępnych elementów logicznych [17].

Producent soft-procesora Nios II dostarcza zestaw standardowych układów peryferyjnych powszechnie używanych w mikrokontrolerach, takich jak: liczniki, interfejsy komunikacji szeregowej, kontrolery SDRAM oraz inne interfejsy pamięci.

Dodatkowo użytkownik może zaimplementować własne komponenty, zarówno jako układy peryferyjne, przeznaczone do komunikacji z otoczeniem, jak i bloki sprzętowe dedykowane do wykonania odpowiednich zadań. W przypadku operacji, które obciążają rdzeń procesora, a ich czas wykonywania jest krytyczny dla wydajności systemu, powszechną techniką jest tworzenie własnego bloku sprzętowego, który realizuje tę samą funkcjonalność na poziomie układu kombinacyjnego lub sekwencyjnego. Takie podejście zapewnia podwójną korzyść. Implementacja algorytmu w postaci modułu sprzętowego najczęściej zapewnia krótszy czas realizacji niż w przypadku oprogramowania w języku wysokiego poziomu. Dodatkowo rdzeń procesora może realizować równolegle inne zadania, podczas gdy dedykowany blok sprzętowy wykonuje operacje na danych [17].



Rys. 2.4: Instrukcje projektanta w procesorze Nios II [20]

Jedną z charakterystycznych cech procesorów Nios II jest możliwość rozszerzenia listy instrukcji, o instrukcje własne projektanta (*ang. Custom Instructions*) [12]. Oznacza to, że projektant może dołączyć do rdzenia Nios II blok logiczny rozszerzający możliwości wbudowanej jednostki arytmetyczno-logicznej (rys. 2.4). Przykładem takich instrukcji, są dostępne w środowisku projektowym bloki wykonujące obliczenia zmiennoprzecinkowe. Tworzenie własnego bloku logicznego odbywa się poprzez zaimplementowanie jego architektury w jednym z języków opisu sprzętu używając do tego celu przygotowanego przez producenta interfejsu. Blok sprzętowy może przyjmować na wejściu dwa 32-bitowe argumenty interpretowane jako zmienne typu całkowitoliczbowego (integer) lub zmiennoprzecinkowego pojedynczej precyzji (float). Otrzymany wynik (postaci stałoprzecinkowej lub zmiennoprzecinkowej) można odczytać po zadanej przez projektanta liczbie cykli zegara taktującego blok logiczny lub po wystawieniu odpowiedniej flagi oznaczającej koniec procesu przetwarzania. Instrukcje te są dostępne dla programisty w postaci makr języka C. Aby zapewnić możliwość współpracy bloku funkcyjnego zaprogramowanego przez użytkownika z jednostką arytmetyczno-logiczną procesora

należy zachować odpowiedni, zdefiniowany przez producenta interfejs. W zależności od typu dodawanej instrukcji dostępne są trzy jego rodzaje [17]:

- instrukcja kombinacyjna (*ang. Combinatorial Custom Instructions*) – interfejs tego typu składa się z dwóch 32-bitowych magistrali wejściowych oraz jednej wyjściowej, również o szerokości 32 bitów. Pozwala na zaimplementowanie układu kombinacyjnego którego wynik działania pojawi się na wyjściu po jednym takcie zegara [17].
- instrukcja sekwencyjna (*ang. Multicycle Custom Instructions*) – interfejs realizujący jedną funkcję o stałej lub zmiennej liczbie cykli – podobnie jak w przypadku interfejsu logiki kombinacyjnej posiada magistrale do wczytywania argumentów oraz zwracania wyniku operacji. Rozszerzony jest jednak o sygnały sterujące charakterystyczne dla układu sekwencyjnego [17].
- multi-instrukcje sekwencyjne (*ang. Extended Custom Instructions*) – interfejs umożliwiający realizację wielu funkcji w jednym bloku sprzętowym, rozszerzony o magistralę do wyboru numeru zdefiniowanej funkcji. Podejście takie pozwala na zaoszczędzenie elementów logicznych układu, w przypadku gdy kilka funkcji korzysta z takich samych lub podobnych komponentów sprzętowych [17].

Komunikacja między procesorem Nios II a instrukcjami koprocesora, oraz poszczególnymi elementami systemu mikroprocesorowego realizowana jest poprzez wewnętrzną magistralę Avalon. W zależności od typu bloku sprzętowego użytkownika, dokonuje się konfiguracji magistrali [14]. Na etapie integracji systemu instrukcje użytkownika dodawane są do pliku wynikowego bsp (*ang. board support package*), co umożliwia odwołanie się do tych instrukcji w języku ANSI C poprzez makra zdefiniowane w bibliotece głównej systemu – system.h.

2.5 Bibliografia

[1] Greg Moss, Neal S. Widmer, Ronald J. Tocci: *Digital Systems, Global Edition*. ISBN 978-12-92162-00-3; Publisher: Perason; 2018.

[2] Valery Salauyou, Adam Klimowicz, Tomasz Grześ, Irena Bułatowa: *Język Verilog w projektowaniu systemów wbudowanych na układach FPGA*. ISBN 978-83-67185-07-3; Oficyna Wydawnicza Politechniki Białostockiej; 2022.

[3] *1364-1995 - IEEE Standard Hardware Description Language Based on the Verilog*. ISBN: 978-0-7381-3065-1; Publisher: IEEE; 14.10.1996.

[4] *1364-2001 - IEEE Standard Verilog Hardware Description Languag*. ISBN 978-0-7381-2827-6; Publisher: IEEE; 28.10.2001.

[5] *1076-1993 - IEEE Standard VHDL Language Reference Manual*. ISBN 978-0-7381-0986-2; Publisher: IEEE; 06.06.1994.

[6] *1076-2008 - IEEE Standard VHDL Language Reference Manual*. ISBN 978-0-7381-6853-1; Publisher: IEEE; 26.01.2009.

[7] 1076-2019 - IEEE Standard VHDL Language Reference Manual, ISBN 978-1-5044-6135-1, Publisher: IEEE, 23.12.2019

[8] D.D. Gajski, F. Vahid: *Specification and Design of Embedded Hardware-Software Systems*. IEEE Design & Test of Computers; vol. 12; no 1; pp. 53-67; Spring 1995.

[9] Zbigniew Hajduk: *Wprowadzenie do języka Verilog*. ISBN 978-83-60233-50-4; Wydawnictwo BTC; 2015.

[10] Kaan Erkorkmaz, Yusuf Altintas: *High speed CNC system design, Part I: jerk limited trajectory generation and quintic spline interpolation*. International Journal of Machine Tools & Manufacture; (41):1323–1345; 2001.

[11] Intel, Quartus Prime - Introduction [on-line]: *https://www.intel.com/content/www/us/en/docs/programmable/683475/19-4/introduction-to.html*. Pobrane 04-2023.

[12] Intel (Altera), SoPC Builder - User guide [online]: *https://www.altera.com/en_US/pdfs/literature/ug/ug_sopc_builder.pdf*. Pobrane 04-2023.

[13] Intel, Introduction to the Platform Designer Tool [on-line]: *https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Introduction_to_the_Qsys_Tool.pdf*. Pobrane 04-2023.

[14] Intel, Avalon Interface Specifications [on-line]: *https://cdrdv2-public.intel.com/667068/mnl_avalon_spec-683091-667068.pdf*. Pobrane 04-2023.

[15] Eclipse [online]: *https://www.eclipse.org/*. Pobrane 04-2016.

[16] Xilinx, BSB, [online]: *https://dl.btc.pl/kamami_wa/digilent_6003_210_000_base_system_builder.pdf*. Pobrane 04-2023.

[17] Grzegorz Góra: *Sterowanie napędami bezpośrednimi – rozprawa doktorska*. Akademia Górniczo-Hutnicza; Kraków; 2019.

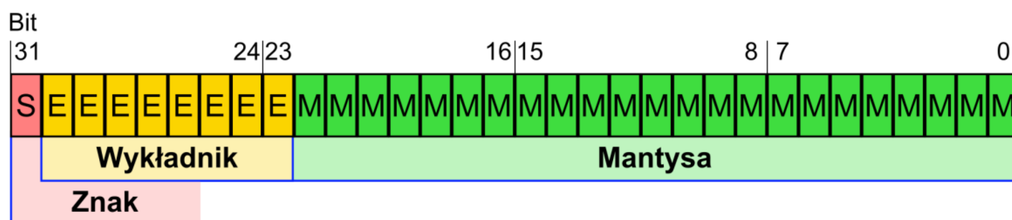
[18] Nios II Processor Reference - Handbook [on-line]: *http://www-ug.eecg.toronto.edu/msl/manuals/n2cpu_nii5v1.pdf*. Pobrane 04-2023.
[19] Introduction to the Altera Nios II Soft Processor [on-line]: *https://people.ece.cornell.edu/land/courses/ece5760/DE2/tut_nios2_introduction.pdf*. Pobrane 04-2023.
[20] Borja Martínez Huerta, Màrius Montón Macián, Jordi Carrabina Bordoll: *Síntesis de Unidades Funcionales para Soft-Cores desde un modelo C/C++*. FPGAs: Metodologías, herramientas y aplicaciones Editores: Juan A. Gómez Pulido, Juan M. Sánchez Pérez, Miguel A. Vega Rodríguez - Universidad de Extremadura ISBN-10: 84-611-1314-4 ISBN-13: 978-84-611-1314-9; Str. 112-117.

Rozdział 3 – Implementacja obliczeń arytmetycznych na liczbach zmiennoprzecinkowych

3.1 Reprezentacja liczby zmiennoprzecinkowej

Liczba zmiennoprzecinkowa pojedynczej precyzji zdefiniowana zgodnie ze standardem IEEE-754 [1] znana z języków programowania C/C++ jako zmienna typu float czy wykorzystywana w języku Pascal zmienna typu single stanowi podstawę zmiennoprzecinkowych obliczeń numerycznych w systemach sterowania i regulacji.

Reprezentacja liczby zmiennoprzecinkowej pojedynczej precyzji składa się z 32 bitów podzielonych na trzy części. Najmłodsze 23 bity reprezentują wartość mantysy, kolejne 8 bitów to wartość wykładnika przesunięta o składową stałą (*ang. bias*) równą 127. Pozwala to na kodowanie zarówno liczb dodatnich jak i ujemnych bez konieczności stosowania bardziej złożonego sposobu kodowania ze znakiem jak Znak-Moduł (*ang. Sign-Magnitude*) czy Kod uzupełnień do 2 - U2 (*ang. Two's Complement*). Ostatnim składnikiem liczby, na pozycji najstarszego bitu jest bit znaku. Wartość '0' oznacza, że liczba jest dodatnia, natomiast wartość '1' oznacza, że kodowana liczba ma wartość ujemną. Postać liczby zmiennoprzecinkowej pojedynczej precyzji przedstawiona na rys 3.1 jest prawdziwa tylko dla przypadku liczb znormalizowanych czyli takich, w których wartość wykładnika jest różna od 0 (poza przypadkiem gdy kodowane jest +/- 0). Normalizacja polega tu na "zaniedbaniu" kodowania '1' w mantysie na pozycji najstarszego bitu. Oznacza to, iż pełna reprezentacja mantysy posiada 24 bity, z których na najstarszej pozycji zawsze znajduje się '1', a pozostałe 23 bity zapisane są bezpośrednio w mantysie. Można zauważyć, że dla takiego zapisu wartość mantysy zawsze mieści się w przedziale <1.0, 2.0) [2].



Rys. 3.1: Reprezentacja liczby zmiennoprzecinkowej pojedynczej precyzji[3]

Wartości mantysy, wykładnika oraz liczby zapisanej w formacie zmiennoprzecinkowym wyrażone są zależnościami:

$$M = 1.0 + \sum_{i=0}^{22} m_i \cdot 2^{-(i+1)} \ , \tag{3.1}$$

$$E = \sum_{i=0}^{7} e_i \cdot 2^{i} \ , \tag{3.2}$$

$$D = (-1)^S \cdot M \cdot 2^{E-127} \ , \tag{3.3}$$

gdzie:

m_i – wartości bitów mantysy [m_{22} m_{21} … m_2 m_1 m_0],
e_i – wartości bitów wykładnika [e_7 e_6 e_5 e_4 e_3 e_2 e_1 e_0],
S – bit znaku,
D – wartość liczby.

Należy zaznaczyć, iż standard IEEE-754 poza kodowaniem wartości ustalonych liczb przewiduje także występowanie znaków specjalnych jak +/- ∞ oraz NaN (*ang. Not a Number*) oznaczający symbol nieoznaczony, który jest wynikiem między innymi operacji takich jak: 0/0 lub 0·∞. W tabeli 3.1 przedstawiono kodowanie mantysy oraz wykładnika wraz z odpowiadającymi im typami liczb [2].

Znak	**Wykładnik**	**Mantysa**	**Rodzaj liczby**
Dowolny	Różny od '00000000' oraz '11111111'	Dowolna	Liczba znormalizowana
Dowolny	'00000000'	Różna od '00000000000000000000000'	Liczba nieznormalizowana
'0'	'00000000'	'00000000000000000000000'	+ 0
'1'	'00000000'	'00000000000000000000000'	- 0
'0'	'11111111'	'00000000000000000000000'	+ ∞
'1'	'11111111'	'00000000000000000000000'	- ∞
Dowolny	'11111111'	Różna od '00000000000000000000000'	NaN

Tab. 3.1 Kodowanie liczby zmiennoprzecinkowej pojedynczej precyzji

3.2 Jednostka zmiennoprzecinkowa FPU

3.2.1 Dodawanie i odejmowanie liczb zmiennoprzecinkowych

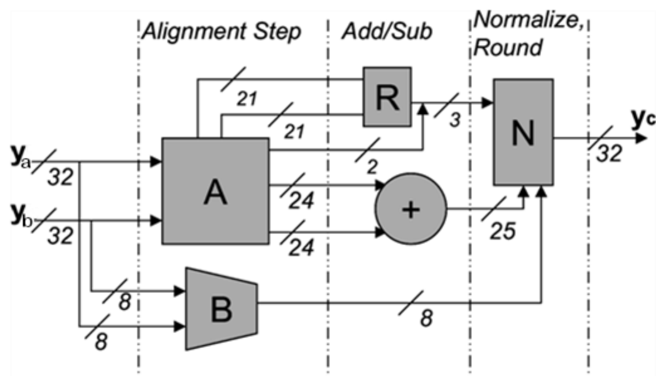
Operacje dodawania i odejmowania dwóch liczb zmiennoprzecinkowych można przedstawić w postaci równania:

$$y_c = y_a \pm y_b = (a \cdot 2^{Ea}) \pm (b \cdot 2^{Eb}). \tag{3.4}$$

Mantysy reprezentowanych liczb y_a i y_b, mogą być dodawane/odejmowane tylko wtedy, gdy wykładniki obu liczb są takie same: $E_a = E_b$. Dla innego przypadku konieczne jest wykonanie wstępnej normalizacji (prenormalizacja). Zapis równania (4) można wtedy przekształcić do postaci:

$$c = a \pm b/2^{Ea-Eb}. \tag{3.5}$$

W celu zachowania ogólności działań, zakłada się, że drugi argument ma mniejszą wartość bezwzględną wykładnika ($|E_b| < |E_a|$). Jeżeli nie jest to prawda, pierwszy i drugi argument operacji muszą być zamienione kolejnością. Po dokonaniu normalizacji wykładników, wykonywane jest dodawanie lub odejmowanie wartości mantys, w zależności od wartości znaku operacji. Dodawanie lub odejmowanie mantys liczb zmiennoprzecinkowych jest działaniem realizowanym tak samo jak w przypadku liczb stałoprzecinkowych. W następnym etapie dokonywana jest post-normalizacja mantysy wyniku. Oznacza to, że znormalizowana postać mantysy powinna mieć zapis $1.M_c$. Natomiast wykładnik, początkowo ustawiony na $E_c = E_a$, jest dostosowany (inkrementowany lub dekrementowany), adekwatnie do normalizacji mantysy. Rys. 3.2 przedstawia potokową realizację zmiennoprzecinkowego dodawania i odejmowania.



Rys. 3.2: Schemat modelu bloku sprzętowego do operacji dodawania lub odejmowania zmiennoprzecinkowego [4]

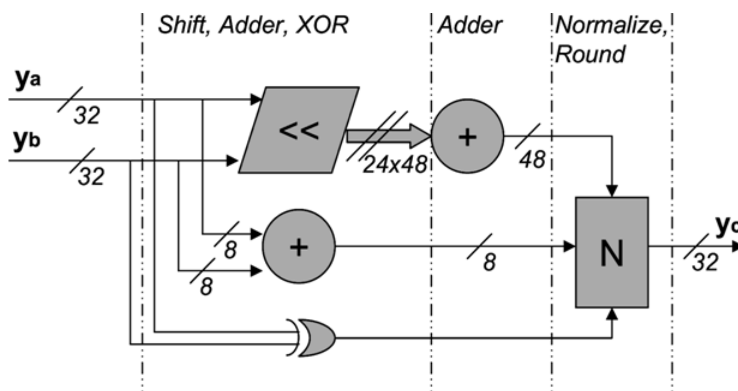
Poszczególne etapy algorytmu, przedstawionego na rys. 3.2, dostosowano do implementacji na układzie FPGA tak, aby uzyskać jak najkrótszy czas realizacji operacji. W pierwszym etapie dokonywane jest wyrównanie wartości wektorów wejściowych. Wektory wyjściowe bloku A reprezentują wartości mantys liczb y_a, y_b, po dokonaniu wyrównania przez rejestry przesuwne do pozycji o numerze określonym na podstawie różnicy bezwzględnych wartości wykładników. Natomiast wynikiem bloku B jest pre-znormalizowana wartość wykładnika. W kolejnym etapie blok R wykonuje operacje przygotowania trzech ostatnich bitów (LSB) wartości mantysy wyjściowej dla operacji zaokrąglenia. Jednocześnie wykonywane jest dodawanie/odejmowanie wyrównanych wartości wektorów mantysy. W ostatnim etapie, blok N realizuje post-normalizację, ustalenie bitu znaku i zaokrąglenie wyniku do najbliższej wartości reprezentowanej zgodnie z kodowaniem zmiennoprzecinkowym opisanym w standardzie IEEE754.

3.2.2 Mnożenie liczb zmiennoprzecinkowych

Operację mnożenia uznaje się za najprostszą pod względem opisu i realizacji operacją na liczbach zmiennoprzecinkowych [5]. Realizację mnożenia dwóch liczb zmiennoprzecinkowych można zapisać w postaci równania:

$$y_c = y_a \cdot y_b = (-1)^{S_a+S_b} \cdot 2^{E_a+E_b+2\cdot bias} \cdot (1.M_a \cdot 1.M_b)\,. \tag{3.6}$$

Mnożenie dokonuje się poprzez pomnożenie mantys liczb y_a, y_b i dodania ich wykładników. Suma wykładników musi zostać powiększona o dwukrotną wartość biasu, co wynika z typu liczb. Znak wartości wyjściowej oblicza się po przez operację logiczną XOR [6].



Rys. 3.3: Schemat modelu bloku sprzętowego do operacji mnożenia zmiennoprzecinkowego [4]

Operacja mnożenia mantys liczb wejściowych dokonywana jest tak samo jak dla przypadku liczb stałoprzecinkowych. Dodanie mantys realizowane jest w sumatorze

równoległym. Na etapie przygotowania dwóch wektorów reprezentujących mantysę liczb wejściowych, dokonywane jest wyrównanie ich postaci do długości 48-bitowej oraz adekwatne przesunięcie cechy.

W pierwszym etapie realizacji mnożenia (rys. 3.3) wykonywane są równolegle trzy operacje: dodanie wykładników argumentów wejściowych, przesunięcia (wyrównania) ich mantys do postaci macierzy składającej się z 24 wektorów o długości 48-bitowej i operacja logiczna XOR najbardziej znaczących bitów, w celu ustawienia wyjściowej wartości bitu określającego znak. Drugi etap składa się z operacji równoległego dodawanie wartości wektorów będących wynikiem operacji wyrównania, dla obliczenia wstępnej wartości mantysy. W ostatnim etapie realizowana jest normalizacja i zaokrąglenie wartości wyjściowej do postaci zgodnej z kodowaniem zmiennoprzecinkowym opisanym w standardzie IEEE754. Dla przypadku mnożenia dwóch takich samych argumentów wejściowych, czyli podniesienia operandu do kwadratu, sumator wykładników może zostać zastąpiony przez układ przesunięcia bitowego, a zapis równania xxx przekształcony do postaci:

$$y_c^2 = y_c \cdot y_c = (-1)^{S_c} \cdot 2^{2 \cdot E_c + 2 \cdot bias} \cdot (1.M_c \cdot 1.M_c)\,. \tag{3.7}$$

3.2.3 Dzielenie liczb zmiennoprzecinkowych

Realizację dzielenia dwóch argumentów wejściowych zapisanych w formacie zmiennoprzecinkowym pojedynczej precyzji można zapisać w postaci równania:

$$y_c = \frac{y_a}{y_b} = (-1)^{S_a + S_b} \cdot 2^{E_a - E_b} \cdot \left(\frac{1.M_a}{1.M_b}\right). \tag{3.8}$$

Na podstawie powyższego zapisu, w celu podzielenia dwóch argumentów zmiennoprzecinkowych, należy dokonać odjęcia wartości wykładników i podzielenia wartości mantys. Wartość wykładnika wyjściowego musi być jednokrotnie skorygowana o wartość biasu. Ustawienie wartości bitu reprezentującego znak wartości wyjściowej jest rezultatem funkcji logicznej XOR, najbardziej znaczących bitów (tak jak w przypadku mnożenia). Dzielenie mantys argumentów jest realizowane na podstawie algorytmu Radix 16 [5], [7] opisującego operacje dzielenia dla dwóch liczb stałoprzecinkowych. Koncepcja algorytmu polega na ustaleniu maksymalnej wartości dzielnika, który może być odjęty od dzielnej w każdym etapie iteracji, nie powodując zmiany znaku wyniku. Dla niezerowej wartości reszty oraz spełnieniu warunku zachowania znaku, będącej rezultatem operacji odjęcia dwóch argumentów w n-tej iteracji algorytmu, dokonuje się inkrementacji jej wartości odpowiadającej przesunięciu 4-bitowemu. Wartość ta reprezentuje dzielnik dla n+1 kroku iteracji algorytmu. W ostatnim etapie algorytmu realizującego dzielenia dwóch argumentów zmiennoprzecinkowych pojedynczej precyzji, dokonywana jest

normalizacja i zaokrąglenie wartości wyjściowej do postaci zgodnej z kodowaniem zmiennoprzecinkowym opisanym w standardzie IEEE754. rys. 3.4 przedstawia potokową realizację operacji dzielenia zmiennoprzecinkowego, na podstawie algorytmu Radix 16.

Rys. 3.4: Schemat modelu bloku sprzętowego do operacji dzielenia zmiennoprzecinkowego na podstawie algorytmu Radix 16 [4]

3.2.4 Testy jednostki zmiennoprzecinkowej FPU

W ramach testów, zaprojektowaną jednostkę FPU zaimplementowano w postaci bloku sprzętowego do akceleracji obliczeń (instrukcji koprocesorowych) dla procesor Nios II, będącego główną jednostką systemu SoPC, na platformie sprzętowej Terasic DE2-115 [8]. Częstotliwość zegara taktującego jednostkę wynosiła 50 MHz. Pierwszy przeprowadzony test polegał na pomiarze czasu oraz dokładności obliczeń, dla zadanych 150 000 par pseudolosowych liczb zmiennoprzecinkowych pojedynczej precyzji dla każdej z instrukcji. Pomiarów dokonano dla opracowanej jednostki FPU i porównano je, z wynikami standardowych instrukcji sprzętowych producenta narzędzia programistycznego Platform Designer (dawniej SoPC Builder). Wyniki uzyskanych czasów obliczeń, zostały przedstawione w tab. 3.2. Analiza wyników wykazała zerowy błąd obliczeniowy pomiędzy wynikami uzyskanymi dla opracowanego FPU a instrukcjami koprocesorowymi producenta.

Typ operacji	**Czas wykonania operacji [liczba cykli]**		**Redukcja czasu [%]**
	Standardowe instrukcje (biblioteki	**Instrukcje użytkownika (opracowana**	

	producenta)	jednostka FPU)	
Dodawanie (Add)	17	10	41
Odejmowanie (Sub)	17	11	35
Mnożenie (Mul)	19	10	47
Dzielenie (Div)	42	16	62

Tab. 3.2: Rezultat testów pomiaru czasów obliczeń dla typów operacji FPU

3.3 Funkcje trygonometryczne sinus i kosinus

Dla obliczania funkcji trygonometrycznych można wyróżnić dwa klasyczne podejścia. Pierwsze z nich bazuje na tablicowaniu wartości czyli metodyce wykorzystującej strukturę w formie tablicy do przechowywania wcześniej przygotowanych danych, co umożliwia zaoszczędzenie czasu wymaganego na ich obliczenie kosztem większego zużycia zasobów sprzętowych. Rozmiar tablicy jest zależny od dokładności i dla przypadku rozdzielczości większej niż około 10 bitów metoda jest mało efektywna. Druga metoda wykorzystuje aproksymację funkcji trygonometrycznych za pomocą wielomianów wysokich rzędów, co pozwala uzyskać wyższą dokładność wyników w stosunku do metody LUT (*ang. Look Up Table*), dla takiej samej ilości rezerwowanej pamięci. Dokładność obliczeń jest zależna od liczby iteracji, składających się z wykonywania podstawowych operacji arytmetycznych, i pociąga za sobą dużą złożoność obliczeniową. Metodą wykorzystującą zalety obu klasycznych podejść obliczania funkcji trygonometrycznych, pozwalającą na uzyskanie dużych dokładności przy niewielkim zajęciu zasobów sprzętowych nazywana jest CORDIC (*ang. Coordinate Rotation Digital Computer*) [9], [10], [11], [12]. Algorytm bazuje na geometrii płaskiej, działaniach arytmetycznych: dodawania, odejmowania, logicznych: przesunięcia bitowego oraz tablicy pamięci o zredukowanych rozmiarach w porównaniu do metody LUT. Algorytm CORDIC wykonywany jest przez cykliczne wyznaczanie współrzędnych punktu znajdującego się na okręgu jednostkowym poprzez rotację aktualnej pozycji o przyjętą wartość kąta, stałą dla danego kroku iteracji (rys. 3.5).

Wartość tego kąta musi spełniać zależność:

$$\mathrm{tg}(\Phi_i) = \pm 2^{-i} \quad , \qquad (3.9)$$

gdzie:

Φ_i – wartość kąta obrotu dla danego kroku iteracji,

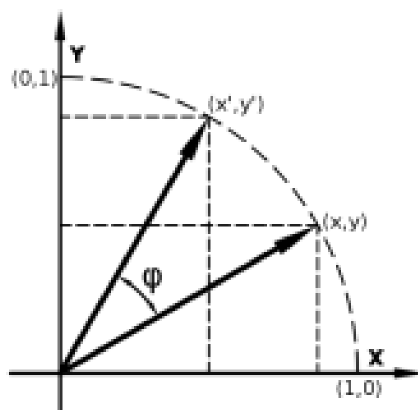
i – krok iteracji,

dlatego dla większości przypadków jego wartość nie jest obliczana w każdym powtórzeniu lecz tablicowana.

Powyższa zależność pozwala na zastąpienie operacji mnożenia tangensa kąta obrotu, operacją logiczną - przesunięciem bitowym. Składniki $\cos(\Phi_i)$ ulegają uproszczeniu do postaci iloczynu, którego wartość jest stała dla zadanej liczb kroków iteracji (dalej oznaczana jako C). Zależność: $\cos(-\Phi) = \cos(\Phi)$ powoduje że wartość stałej C jest zawsze dodatnia i nie zależy od znaku kąta obrotu. Dodatkowo końcowe mnożenie przez współczynnik C można zastąpić, ustawiając współrzędne początkowe punktu [X0,Y0] = [C, 0]:

$$\begin{aligned} x' &= x\cos(\Phi) - y\sin(\Phi) \rightarrow x' = \cos(\Phi)\,[x - y\,\mathrm{tg}(\Phi)] \;, \\ y' &= y\cos(\Phi) + x\sin(\Phi) \rightarrow y' = \cos(\Phi)\,[y + x\,\mathrm{tg}(\Phi)] \;, \end{aligned} \qquad (3.10)$$

$$\begin{gathered} x'' = \cos'(\Phi)\,[x' - y'\mathrm{tg}(\Phi)]\,, \\ x'' = \cos'(\Phi)\,[\cos(\Phi)\,(x - y\,\mathrm{tg}(\Phi)) - \cos(\Phi)(y + x\,\mathrm{tg}(\Phi)) * \mathrm{tg}'(\Phi)\,]\;, \\ x'' = \cos(\Phi) * \cos'(\Phi)\,[(x - y\,\mathrm{tg}(\Phi)) - (y + x\,\mathrm{tg}(\Phi)) * \mathrm{tg}'(\Phi)\,]\;, \\ x'' = C * [(x - y\,\mathrm{tg}(\Phi)) - (y + x\,\mathrm{tg}(\Phi)) * \mathrm{tg}'(\Phi)\,]\;, \\ \\ y'' = \cos'(\Phi)\,[y' + x'\,\mathrm{tg}'(\Phi)]\,, \\ y'' = \cos(\Phi) * \cos'(\Phi)\,[(y + x\,\mathrm{tg}(\Phi)) + (x - y\,\mathrm{tg}(\Phi)) * \mathrm{tg}'(\Phi)\,]\;, \\ y'' = C * [(y + x\,\mathrm{tg}(\Phi)) + (x - y\,\mathrm{tg}(\Phi)) * \mathrm{tg}'(\Phi)\,]\;. \end{gathered} \qquad (3.11)$$



Rys. 3.5: Transformacja układów współrzędnych na okręgu jednostkowym [4]

W przypadku implementacji funkcji trygonometrycznych w układzie FPGA, ich wartość wyznaczana się stosując jedną z trzech procedur. Wybór procedury jest zależny od przedziału, w którym mieści się argument funkcji. Można wyróżnić trzy przedziały wartości bezwzględnej operandu, co odpowiada trzem sposobom wyznaczania wartości sinusa bądź cosinusa: $| x | < 2^{-12}$, $2^{-12} < | x | < 2\pi$, $| x | > 2\pi$.

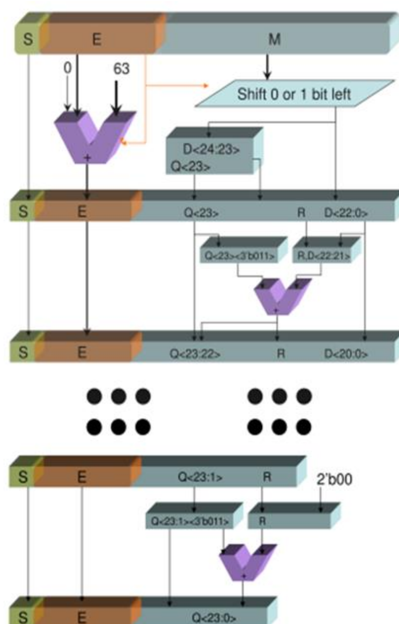
Jeśli wartość bezwzględna argumentu jest mniejsza od 2^{-12} stosuje się aproksymację liniową sin(x) = x oraz cos(x) = 1, która w omawianym przedziale pozwala na uzyskanie poprawnego wyniku z dokładnością nie mniejszą niż 24 bity znaczące.

W przypadku gdy argument funkcji należy do przedziału $\pm 2\pi$, jednak nie należał do przedziału liniowego, stosuje się obliczanie wartości sinusa oraz kosinusa za pomocą algorytmu CORDIC. Aby można było z niego skorzystać, konieczna jest konwersja operandu z postaci zmiennoprzecinkowej na stałoprzecinkową oraz normalizacja kąta do wartości z przedziału $(0,\pi/2)$. Następnie wyznaczana jest wartości sinusa bądź cosinusa w zadanej liczbie kroków iteracji oraz z żądaną dokładnością numeryczną. Zarówno liczba kroków iteracji jak i liczba znaczących bitów, na których wykonywane są obliczenia jest parametrem i może być dostosowywana, w zależności od wymagań czasowych oraz żądanej dokładności numerycznej. Ostatnim etapem jest zamiana postaci stałoprzecinkowej na zmiennoprzecinkową, połączona z normalizacją i zaokrągleniem.

Jeśli argument znajduje się poza przedziałem $\pm 2\pi$, wstępnym etapem wyznaczana wartości funkcji jest obliczenie reszty z dzielenia przez 2π, co uzyskuje się dzięki wykorzystaniu algorytmu dzielenia stałoprzecinkowego Radix8.

3.4 Pierwiastek kwadratowy

Do najbardziej znanych algorytmów opisujących metodę wykonania funkcji pierwiastka kwadratowego w sposób sprzętowy można zaliczyć: metodę realizacji Newton-Raphsody [13] metodę SRT-Redundat [14], oraz metodę Non-Redundat [15] która ma postać restoring lub non-restoring. Metodę Non-Redundat (non-restoring), w porównaniu do pozostałych metod charakteryzuje zredukowana liczba cykli, oraz mniejsza liczba wykorzystanych podstawowych elementów logicznych układu FPGA. Opisanie tej metody na logicznym poziomie abstrakcji i możliwość efektywnej implementacji pozwoliły na opracowanie metody zwanej PASQRT [16] (*Paralel Array for SQuare RooTing*), którą wykorzystano do realizacji instrukcji koprocesora. Działanie algorytmu PASQRT realizującego funkcję pierwiastka kwadratowego zostało przedstawione na rys. 3.6.



Rys.3.6: Schemat blokowy algorytmu PASQRT [2]

Wartości argumentu wejściowego, 32-bitowego pojedynczej precyzji zapisana zgodnie z kodowaniem IEEE-754 ma postać:

$$D = (-1)^S \cdot 1.m \cdot 2^{e-127}, \tag{3.12}$$

gdzie:

S – bit znaku;
e – wartość wykładnika;
m – wartość mantysy.

Założeniem poprawnego wykonania instrukcji jest warunek, że liczba D należy do zbioru liczb dodatnich. W pierwszym etapie algorytmu dokonywane jest sprawdzenie warunku parzystości wykładnika. Jeżeli wartość e należy do zbioru liczb parzystych to wartość mantysy reprezentowanej przez wektor *1.xxx* i nie ulega przesunięciu, w przeciwnym przypadku dokonywana jest operacja przesunięcia bitowego jeden w lewo. Na tym samym etapie wykonywana jest pre-normalizacja wykładnika. Ponieważ postać wykładnika jest dekodowana na podstawie wartości reprezentowanej przez liczbę e pomniejszoną o stałą równą 127, dokonując poszczególnych operacji: dekodowania liczby, dzielenia przez 2 oraz kodowania liczby, operację pierwiastkowania wykładnika można uprościć do postaci:

$$\frac{e-127}{2} + 127 = \frac{e}{2} + 63 + e\%2, \tag{3.13}$$

gdzie:

% oznacza moduł z operacji dzielenia, a wartość e%2 stanowi najmniej znaczący bit liczby e. Z pierwszego etapu otrzymuje się wartość wykładnika, oraz wartość mantysy w postaci [0.1 m_{22} ... m_1] lub [1.m_{22} m_{21} ... m_0]. Wyznaczenie wartości pierwiastka z mantysy sprowadza się do działań wykonywanych na typie liczby stałoprzecinkowej [2].

Czas wykonania opracowanej instrukcji pierwiastka kwadratowego sprawdzono poprzez porównanie wyników obliczeń zadania kinematyki odwrotnej dla manipulatora równoległego frezarki. Testy porównawcze bazują na równaniach będących rozwiązaniem zadania kinematyki odwrotnej manipulatora równoległego oraz pochodnych pierwszego, i drugiego stopnia. Do ich realizacji wykorzystywane są operacje dodawania/odejmowania, mnożenia, dzielenia oraz obliczanie wartości pierwiastka kwadratowego. Do przeprowadzenia testów przyjęto, że trajektorię zadaną będzie stanowić okrąg o promieniu 150 mm usytuowany na wysokości 0.3 m. Do implementacji jednostki zmiennoprzecinkowej wykorzystano platformę sprzętową DE2-115 firmy Terasic [8], z układem FPGA Cyclone IV EP4C4115F29C7N.

Procesor Nios II	**Czas obliczeń [µs]**	**Liczba wykorzystanych elementów logicznych (114480 dostępnych)**
Bez instrukcji koprocesorowych (math.h)	505	4501 (4%)
Z instrukcjami koprocesoro-wymi producenta FPU_ALT	129	5840 (5%)
FPU_ALT + własna instrukcja pierwiastka	31	7219 (6%)

Tab. 3.3: Rezultat testu, czasu obliczeń i wykorzystanych zasobów FPGA, sprzętowej realizacji funkcji pierwiastka kwadratowego

Częstotliwość zegara taktującego wynosiła 50MHz. W tab. 3.3 przedstawiono jak zmieniał się czas potrzebny na wykonanie obliczeń oraz ilość wykorzystanych zasobów układu FPGA w zależności od wersji jednostki. Dla pierwszego rozwiązania cały system użyty do testów, zajmował 4501 elementów logicznych (LE) układu FPGA. Czas obliczeń dla jednego punktu trajektorii wyniósł 505 µs. W przypadku dodania standardowych (dostarczanych przez producenta oprogramowania), sprzętowych instrukcji zmiennoprzecinkowych dla procesora Nios II (FPU_ALT), czas obliczeń skrócił się do 129 µs. Dodanie koprocesora arytmetycznego nieznacznie zwiększyło ilość wykorzystanych zasobów. Największą wydajność obliczeń osiągnięto po dodaniu własnej instrukcji pierwiastka kwadratowego. Ilość wykorzystanych zasobów wzrosła nieznacznie (ok. 1400 LE w stosunku do wersji z samym koprocesorem producent), natomiast czas obliczeń wyniósł 31 µs.

Dokładność numeryczną opracowanej instrukcji pierwiastka kwadratowego sprawdzono poprzez porównanie wyników z trajektorii testowanej z trajektorią

wcześniej obliczoną na komputerze PC z podwójną precyzją. Dla określenia maksymalnych błędów obliczeń, zestawionych w tab. 3.4, porównano punkt po punkcie otrzymane trajektorie. Do porównania wyników wykorzystano zbiór liczb zmiennoprzecinkowych pojedynczej precyzji otrzymany z obliczeń systemu zaimplementowanego na FPGA, oraz zbiór wyników liczb zmiennoprzecinkowych podwójnej precyzji otrzymanych z obliczeń na komputerze PC. Dzięki takiemu zestawieniu wyników uzyskano informacje dotyczące rozkładu błędu zaokrąglenia dla zadanej trajektorii względem wyników o podwójnej precyzji.

Wielkość	Maksymalny błąd
Pozycja [m]	$2 \cdot 10^{-7}$
Prędkość [m/s]	$5 \cdot 10^{-8}$
Przyspieszenie [m/s^2]	$3 \cdot 10^{-8}$

Tab. 3.4: Maksymalny błąd obliczeń jednostki - procesorem NiosII z instrukcjami koprocesorowymi do operacji zmiennoprzecinkowych dostarczonych przez producenta oprogramowania i opracowaną sprzętową funkcją pierwiastka kwadratowego.

W wyniku zastosowania własnej instrukcji pierwiastka kwadratowego, uzyskano czterokrotne skrócenie czasu obliczeń. Liczba wykorzystanych elementów logicznych wzrosła o 1% układu Cyclone IV firmy Altera. Przedstawione opracowanie instrukcji koprocesorowej dla procesora Nios II, na podstawie modelu bloku sprzętowego realizującego operację pierwiastka kwadratowego wraz z wynikami testów zostały opublikowane w pracach [17], [18].

3.5 Funkcja nasycenia (saturacji)

Saturacja realizuje ograniczenie wartości liczby do przedziału określonego przez maksymalną i minimalną wartość. Operand oraz wartości granicy nasycenia są argumentami wejściowymi funkcji saturacji. Funkcja nasycenia nie jest obecnie opisana jako funkcja w standardowych bibliotekach języka ANSI C, lub jako instrukcja koprocesorowa w standardowych bibliotekach producenta narzędzia programistycznego (Platform Designed [19]). Występuje natomiast jako funkcja np. w programie Matlab. Opis sprzętowy modelu funkcjonalnego (za pomocą języka HDL), ogranicza się do sformułowania trzech instrukcji warunkowych, sprawdzających wartość argumentu wejściowego. Funkcję tę można zapisać w postaci równania:

$$y = \begin{cases} p_{max} & dla \quad x \geq p_{max}; \\ p_{min} & dla \quad x \leq p_{min}; \\ x & dla \quad p_{max} > x > p_{min}; \end{cases} \tag{3.14}$$

gdzie:

x - argument funkcji saturacji (typu float),
p_{max}, p_{min} - parametry funkcji saturacji, ekstrema (maksimum, minimum),
y - wynik funkcji saturacji (typu float).

3.6 Funkcja signum

Signum jest drugą funkcją, która nie jest obecna w standardowych bibliotekach ANSI C lub jako instrukcja koprocesorowa, w standardowych bibliotekach producenta narzędzia programistycznego (Platform Designed [19]). Funkcja przyjmuje jeden argument w reprezentacji zmiennoprzecinkowej. Na podstawie wartości argumentu zwraca rezultat: 1, dla dodatniej wartości, -1, dla ujemnej wartości lub 0, dla zerowej wartości. W przypadku liczby kodowanej w standardzie IEEE 754, znak liczby reprezentuje najstarszy, 31-bit (*ang. MSB*). Jeżeli wartość liczby (argumentu wejściowego funkcji) jest równa zero, co jest określone mianem wartości szczególnej, bit znaku może przyjmować wartość 1 lub 0 (co odpowiada wartości liczby "+0", lub "-0"). Dlatego w pierwszej kolejności w realizacji sprzętowej, sprawdzany jest warunek, dotyczący niezerowej wartości cechy i mantysy argumentu wejściowego. Funkcję signum, można zapisać w postaci równania:

$$y = \begin{cases} 0 \quad dla \quad x = 0; \\ 1 \quad dla \quad x > 0; \\ -1 \quad dla \quad x < 0; \end{cases} \tag{3.15}$$

gdzie:
x - argument funkcji signum (typu float),
y - wynik funkcji signum (typu float).

3.7 Funkcja rzutowania float ⇔ int

Kolejną opracowaną sprzętową instrukcją koprocesorową, jest konwersja 32-bitowego argumentu typu całkowitego bez znaku (ang. *unsigned integer*), na typ zmiennoprzecinkowy pojedynczej precyzji (IEEE 754, float), stanowiącą rezultat funkcji. Standardowa (programowa) funkcja konwersji, realizująca powyższe zadanie, jest nazywana: jawnym rzutowaniem typów. Na podstawie przeprowadzonych testów, dla zapisu powyższej funkcji w języku ANSI C, jednostka, której architektura oparta jest na procesorze Nios II (bez instrukcji koprocesorowej *Cast integer to float*), potrzebuje 111 cykli zegara taktującego do jej realizacji. W celu zredukowania czasu wykonania operacji rzutowania, opracowano własną instrukcję sprzętową, na podstawie poniższego opisu. Następujące równania opisują argument i rezultat operacji Cast integer to float:

$$y_a = b_{31} \cdot 2^{31} \ldots + b_i \cdot 2^{X_i} \ldots + b_a \cdot 2^{X_a} \ldots + b_0 \cdot 2^0$$
$$y_c = (-1)^0 \cdot 2^{E_c - bias} \cdot 1.M_c \qquad (3.16)$$

gdzie:

y_a - argument funkcji (32-bitowy, typu całkowitego bez znaku);
y_c - rezultat funkcji (32-bitowy, typu zmiennoprzecinkowego pojedynczej precyzji - waga i-tego bitu;
b_i - wartość i-tego bitu.

W pierwszym etapie sprawdzany jest warunek o zerowej wartości argumentu wejściowego funkcji:

$$y_c = 0 \text{, jeżeli: } y_a = 0, \qquad (3.17)$$

Dla $\forall \; y_a \neq 0$:

$$2^{E_c - bias} = 2^{X_a} \qquad (3.18)$$

$$\text{jeżeli: } b_a = 1 \wedge b_{i>a} = 0, \quad \forall \quad i,a \in \{31,30,\ldots,0\},$$

$$2^{E_c} = 2^{X_a + bias} \qquad (3.19)$$

W przypadku gdy argument ma wartość inną niż zero, dokonywane jest wyznaczenie bitu o niezerowej wartości i najwyższej wadze, w zapisie liczby stałoprzecinkowej, bez znaku (16), y_a. Jeżeli bit b_a będzie miał niezerową wartość, a pozostałe bity b_i, o wyższej wadze w zapisie (16), y_a, będą miały zerową wartość, to poprzez znajomość wagi tego bitu – 2^{X_a}, dokonywany jest zapis, na podstawie

równania (16), w reprezentacji zmiennoprzecinkowej, części wykładnika (2^{Ec}). Wzór (18) można przekształcić do postaci (19).

Pozostałe wartości bitów, o niższej wadze od bitu *ba* (czyli: $2^{X_{a-1}},...,2^{X_{a=0}}$), w reprezentacji stałoprzecinkowej argumentu wejściowego, zostają przepisane do reprezentacji zmiennoprzecinkowej części mantysy M_c, zaczynając od najstarszego bitu (wartość bitu b_{a-1} przepisana do wartości bitu b_{22} o wadze 2^{22} , itd.). Ponieważ zakres wartości mantysy jest ograniczony do postaci 23-bitowej, dokonywane jest sprawdzenie warunku zakresu (długości wektora) przepisania. W przypadku gdy długość wektora przepisania jest krótsza od 23-bitów, pozostałe wartości bitów, o niższej wadze ($c < (a=0)$) w postaci pre-znormalizowanej mantysy, zostają uzupełnione zerami ($b_c =0$). W przypadku gdy długość wektora przepisania jest dłuższa od 23-bitów, wartość bitu b_c (*dla* $c=0$) o najniższej wadze, czyli 2^0, w postaci pre-znormalizowanej mantysy M_c, stanowi wartość zaokrąglenia n, części wektora przepisania, będącej poza zakresem 23-bitowym, obliczona zgodnie z wytycznymi standardu IEEE-754.

$$M_c = b_{22} \cdot 2^{22} \dots + b_c \cdot 2^{X_c} \dots + b_0 \cdot 2^0, \forall\, c \in \{22, 20, \dots, 0\} \qquad (3.20)$$

jeżeli $a \leq 23$,

$$M_c = b_{a-1} \cdot 2^{X_{a-1}} \dots + b_{a=0} \cdot 2^{X_{a=0}} \dots + b_c \cdot 2^{X_c} + b_0 \cdot 2^0 \qquad (3.21)$$

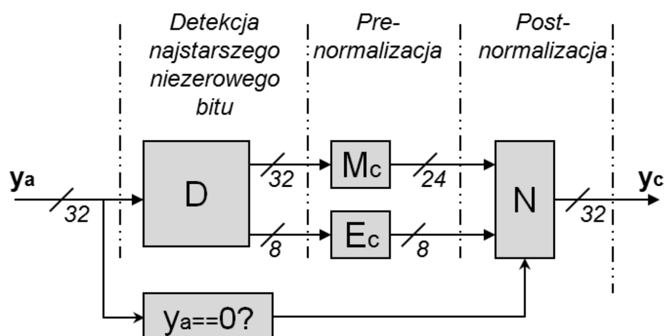
gdzie: $b_c = 0, \forall\, c < (a = 0)$.

Jeżeli: $a > 23$,

$$M_c = b_{a-1} \cdot 2^{X_{a-1}} \dots + b_{a-23} \cdot 2^{X_{a-23}} \dots + n \cdot 2^0 \qquad (3.22)$$

gdzie: $n = b_c$, *dla* $c = 0$.

W realizacji bloku sprzętowego realizującego operację konwersji typów kodowania liczb, z y_a na y_c, można wyszczególnić dwa podstawowe procesy równoległego przetwarzania danych, odnoszące się do warunku sprawdzenia niezdrowej i zerowej wartości argumentu wejściowego. W przypadku $y_a \neq 0$, element bloku D (*Detekcja najstarszego, niezerowego bitu, rys. 3.7*), realizuje funkcję określenia (pozycji) niezerowego bitu, o najwyższej wadze (najstarszego) argumentu y_a. Na podstawie tych informacji elementy bloku: M, E (rys. 3.7), dokonują pre-normalizacji. Otrzymane postacie zostają post-znormalizowane na podstawie wytycznych, co do metody zaokrągleń, w standardzie IEEE-754. Część algorytmu, odpowiedzialna za post-normalizację, została opisana w elemencie bloku – N.



Rys. 3.7: Schemat modelu bloku sprzętowego do operacji Cast integer to float

Wszystkie funkcje opisane w poszczególnych elementach bloku sprzętowego realizującego operację konwersji typów kodowania liczb, zostały opracowane tak, aby wykorzystać cechy równoległego przetwarzania danych przez układy FPGA. W rezultacie pozwoliło to na zredukowanie czasu powyższej operacji do 12 cykli zegara taktującego jednostkę, której architektura oparta jest na procesorze Nios II (doposażonej w instrukcję koprocesorową: *Cast integer to float*). Rzutowanie wartości typu całkowitego na zmiennoprzecinkowy, może być wykorzystywane m.in. w przypadku zapisu wartości odczytanych z enkoderów.

3.8 Testy wydajności obliczeniowej pod kątem wykorzystania w układzie sterowania frezarką

W celu wyznaczenia wydajności obliczeniowej jednostki opartej na soft-procesorze Nios II oraz opracowanych instrukcjach koprocesora zostały przeprowadzone testy pod kątem dokładności numerycznej i czasów wykonania wszystkich opracowanych instrukcji koprocesorowych zaimplementowanych, na platformie sprzętowej (płyta DE3 firmy Terasic [20]), wyposażonej w układ FPGA Stratix III 3SL150 [21], zbudowany z 142k podstawowych elementów logicznych (LEs). Przeprowadzono pomiary czasów realizacji funkcji, oraz dokładność wyników poszczególnych operacji arytmetycznych, trygonometrycznych, signum, saturacji, rzutowania. System wzorcowy, stanowiła struktura SoPC którego budowa była oparta na procesorze Nios II, taktowanym zegarem o częstotliwości 100 MHz. Dokonano 10000 iteracji na pseudolosowych, zmiennoprzecinkowych (float) wartościach argumentów wejściowych, dla każdego z typów operacji. Do opisu programu testowego dla procesora Nios II napisano wykorzystując składnie języka ANSI C, oraz podstawowe biblioteki programowe w tym math.h. Następnie, dla takiej samej liczby prób i tych samych argumentów wejściowych, dokonano pomiarów dla systemu SoPC z procesorem Nios II, rozbudowanym o opracowane zmiennoprzecinkowe instrukcje sprzętowe (koprocesorowe). Częstotliwość

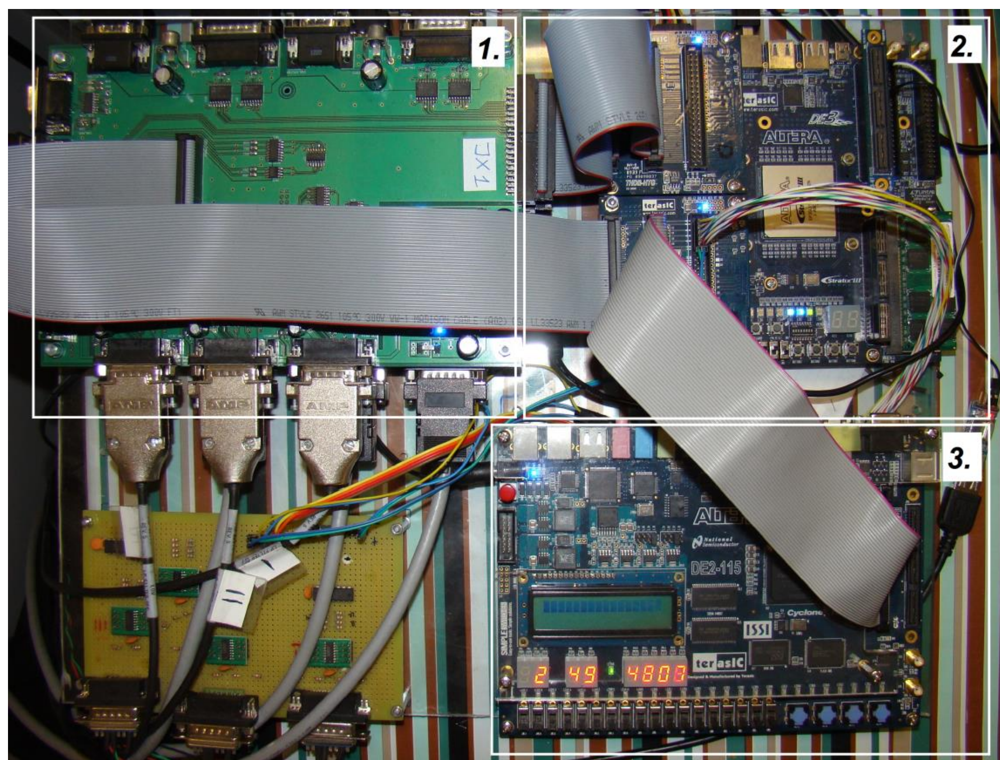
taktowania procesora nie uległa zmianie względem systemu wzorcowego i wynosiła 100 MHz. Średnie czasy operacji przedstawiono w tab. 3.5. Błąd bezwzględny między wartościami zmierzonymi dla tych samych argumentów wejściowych wyniósł zero.

Typ operacji	**Czas realizacji [liczba cykli zegara]**		**Redukcja czasu wykonania operacji [%]**
	Bez instrukcji koprocesorowych (MATH)	**Instrukcje koprocesorowe (FPU_CI + CI)**	
+/-	207	13	93.7
*	183	13	92.9
/	452	20	95.6
sqrt(x)	912	16	98.2
sin(x)	4164	56	98.7
cos(x)	4294	56	98.7
sgn(x)	220	7	96.8
(float)int	111	12	89.2
sat(x,y)	265	8	97.0

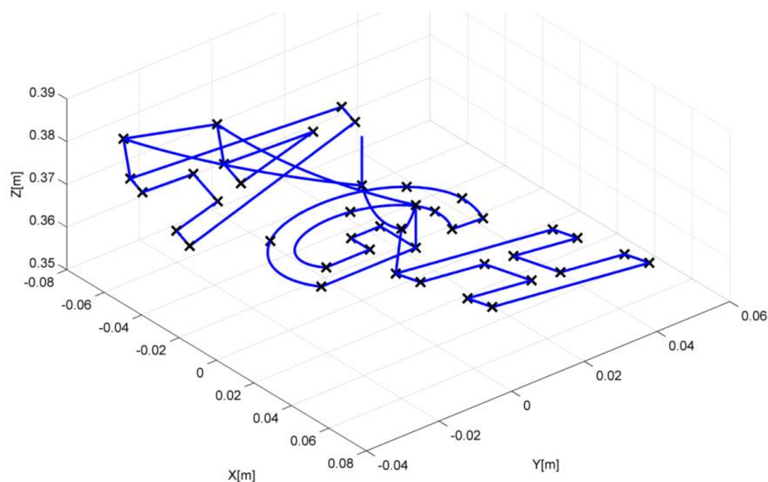
Tab. 3.5: Porównanie rezultatów testu - porównania czasów obliczeń pomiędzy jednostkami: bez instrukcji koprocesorowych (MATH) i z instrukcjami koprocesorowymi (FPU_CI + CI)

W kolejnych przeprowadzonych testach na płycie DE3 firmy Terasic, zaimplementowano architekturę sterownika 5-osiowej frezarki numerycznej typu CNC. Do realizacji buforu wewnętrznego parametrów trajektorii dla algorytmu on-line, użyto pamięci DDRAM [20]. Częstotliwość podstawowego zegara taktującego, użytego do testów, wynosiła 100 MHz. Jako platformę pomiarową wykorzystano płytę DE2-115 firmy Terasic wyposażoną w układ FPGA Cyclone IV (rys. 3.8).

Na rys. 3.9 została zaprezentowany testowy tor narzędzia, składający się z liter "AGH", opisany w G-kodzie. W celu dokładnej eksploracji przestrzeni roboczej, poszczególne elementy toru (litery), zostały rozmieszczone w różnych płaszczyznach. Jednocześnie, zadana toru została tak dobrana aby przebadać działanie prototypu sterownika, pod względem poprawności wykonania algorytmu jak i czasu realizacji, dla różnych typów segmentów toru (liniowe i łukowe) oraz przejazdów jałowych



Rys. 3.8: Zdjęcie: 1. Układu integracji elementów/układów peryferyjnych,
2. Platformy sprzętowa sterownika, Terasic DE3,
3. Platformy sprzętowej pomiarowo-testowej Terasic, DE2-115.



Rys. 3.9: Tor narzędzia użyty do testów

Założono przeprowadzenie testów dla różnych konfiguracji sprzętowej jednostki zmiennoprzecinkowej prototypu sterownika, co pozwoliło na analizę wpływu zastosowania poszczególnych instrukcji na skrócenie czasu (akcelerację) wykonywania realizowanego algorytmu sterowania. W tym celu zaimplementowano 6 struktur sprzętowych systemu, różniących się typem i liczbą instrukcji koprocesorowych jednostki zmiennoprzecinkowej:

- Pierwsza architektura testowa nie została wyposażona w instrukcje koprocesorowe. Wszystkie operacje zmiennoprzecinkowe pojedynczej precyzji realizowane były przez standardowe instrukcje procesora Nios II, przy zastosowaniu funkcji ze standardowych bibliotek języka ANSI C: stdio.h math.h. Kompilator programowy został skonfigurowany na najwyższy poziom optymalizacji (level 3) pod względem obszerności kodu i redukcji czasu wykonania.
  (Oznaczenie architektury w testach: MATH)

- Kolejna architektura testowa wyposażona była w instrukcje koprocesorowe: pierwiastek kwadratowy (sqrt), funkcji sinus (sin), funkcji kosinus (cos), funkcji znaku (sgn), funkcji rzutowania (cast) i funkcji nasycenia (sat). *(Oznaczenie architektury w testach: MATH+CI)*

- Trzecia wersja składała się z podstawowych instrukcji koprocesorowych dostarczonych przez producenta narzędzia programistycznego (SoPC builder [22]): suma, odejmowanie, mnożenie, bez operacji dzielenia. Sprzętowa instrukcja zmiennoprzecinkowa dzielenia, dostarczana przez producenta, została programowo wyłączona (przez ustawienie komendy w pliku konfiguracyjnym: makefile), ze względu na błędne wartości rezultatów działania.
  (Oznaczenie architektury w testach: FPU_ALT)

- Czwarta architektura testowa stanowiła rozszerzoną wersję trzeciej, o opracowane funkcje koprocesorowe (CI) - te same co w wersji drugiej, takie jak: pierwiastek kwadratowy, funkcje trygonometryczne, itp. *(Oznaczenie architektury w testach: FPU_ALT+ CI)*

- Piąta architektura testowa zawierała jednostkę zmiennoprzecinkową z opracowanymi instrukcjami: FPU_CI (operacje: dodawanie, odejmowanie, mnożenie i dzielenie). Pozostałe operacje były wykonywane przez standardowe instrukcje procesora Nios II (bez dodatkowych instrukcji koprocesorowych - CI).
  (Oznaczenie architektury w testach: FPU_CI)

- Ostatnia architektura testowa składała się z procesora Nios II oraz wszystkich opracowanych niestandardowych instrukcji sprzętowych. W systemie tym, większość operacji/instrukcji wykonywana była, przez zmiennoprzecinkowe akceleratory sprzętowe.
  (Oznaczenie architektury w testach: FPU_CI+CI)

Dla każdej z architektur dokonano pomiaru czasu potrzebnego do wykonania zadanej trajektorii oraz sprawdzono dokładność wyników obliczeń, będących wartościami: położenia, prędkości i przyspieszeń, dla każdego okresu sterowania. Pomiar czasu zrealizowano poprzez niezależny (poza systemowy) licznik, będący modułem sprzętowym, zaimplementowanym w układzie FPGA platformy testowo-pomiarowej. Natomiast analizy błędu numerycznego dokonano na podstawie wyznaczenia maksymalnego błędu, będącego różnicą otrzymanych wyników pomiarowych dla każdej iteracji programu, z wartości wzorcowych, zapisanych w formacie liczb podwójnej precyzji (64-bit double). W tab. 3.6 przedstawione zostały czasy obliczeń w zależności od typu segmentu toru dla różnych wersji jednostki zmiennoprzecinkowej.

Jednym z podstawowych wymagań zapewniających poprawną pracę prototypowej frezarki jest wykonywanie obliczeń części on-line algorytmu z częstotliwością nie mniejszą, niż 10 kHz, co odpowiada okresowi próbkowania 100 µs. Na podstawie wyników, jedyną wersją jednostki zmiennoprzecinkowej, spełniająca ten wymóg jest jednostka dwuprocesorowy Nios II rozbudowany o niestandardowe instrukcji FPU i wszystkie dodatkowe instrukcje koprocesorowe, oznaczony w testach jako system: FPU_CI + CI.

Test wykazał jednocześnie że czas realizacji zmiennoprzecinkowego algorytmu części on-line, wykonywany przez procesor na którym zaimplementowany jest układ regulacji wymaga najdłuższego czasu - 82 µs. Realizacja tej części algorytmu w odpowiednio krótkim czasie jest zatem krytyczna dla działania całego systemu.

Oznaczenie wersji jednostki zmiennoprzecinkowej	Czas obliczeń [us]			
	Generator trajektorii			Układ regulacji
	Przejazdy jałowe	Segmenty liniowe	Segmenty kołowe	
MATH	605	625	742	806
MATH + CI	581	523	580	832
FPU_ALT	198	251	291	218
FPU_ALT + CI	175	115	133	167
FPU_CI	109	178	215	149
FPU_CI + CI	63	52	55	82

Tab. 3.6: Rezultat testów czasów obliczeń - przypadku o najdłuższym czasie, potrzebnym do realizacji algorytmu sterownika, dla różnych wersji jednostki zmiennoprzecinkowej

Wielkość	Przejazdy jałowe	Segmenty liniowe	Segmenty kołowe
l_1 [*m*]	1.15e-07	1.13e-07	1.86e-06
l_2 [*m*]	9.19e-08	1.04e-07	1.17e-06
l_3 [*m*]	9.19e-08	1.13e-07	1.05e-06
θ_4 [*rad*]	4.47e-08		
θ_5 [*rad*]	4.47e-08		
v_1 [*m/s*]	2.95e-07	3.86e-08	5.94e-07
v_2 [*m/s*]	1.81e-07	4.43e-08	8.26e-07
v_3 [*m/s*]	1.81e-07	3.22e-08	7.67e-07
ω_4 [*rad/s*]	1.00e-07		
ω_5 [*rad/s*]	1.00e-07		
a_1 [m/s^2]	3.02e-06	1.99e-07	1.33e-06
a_2 [m/s^2]	1.88e-06	1.76e-07	2.70e-06
a_3 [m/s^2]	1.88e-06	2.24e-07	2.43e-06
ε_4 [rad/s^2]	3.20e-07		
ε_5 [rad/s^2]	3.20e-07		

Tab. 3.7: Maksymalny błąd numeryczny, obliczeń dla: pozycji, orientacji, prędkości i przyspieszeń we współrzędnych złączowych

Rezultaty testów dokładności numerycznych dla poszczególnych wersji jednostek zostały przedstawione w tab. 3.7. Wartości błędów: pozycji, orientacji, prędkości, i przyspieszeń dla przejazdów jałowych oraz segmentów liniowych i łukowych, dla zadanego toru ruchu (rys. 3.9) były takie same dla każdej z wersji architektury. Oznacza to, że wynik każdego pojedynczego działania dla opracowanego FPU i instrukcji koprocesorowych oraz standardowego koprocesora zmiennoprzecinkowego firmy Intel (z wyłączeniem sprzętowego dzielenia) był taki sam jak dla pierwszej wersji systemu, działającej wyłącznie na instrukcjach stałoprzecinkowych procesora i wykorzystującej funkcje standardowej biblioteki

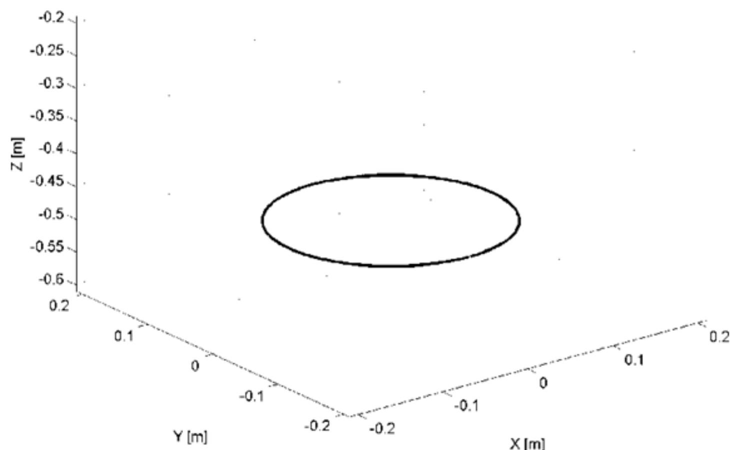
matematycznej języka ANSI C. Maksymalny błąd obliczania pozycji wyniósł około 2 μm, a orientacji 10 μrad, co jest dopuszczalnym wynikiem dla przyjętych wymagań.

W tab. 3.8 przedstawiono, ilość wymaganych zasobów układu FPGA (Stratix III 3SL150 [20]), w zależności od wersji jednostki zmiennoprzecinkowej prototypu sterownika frezarki numerycznej. Do implementacji najbardziej rozszerzonej wersja systemu zmiennoprzecinkowego (FPU_CI + CI), wykorzystano 22% elementarnych jednostek logicznych LUT (*ang. Look-Up Tables*).

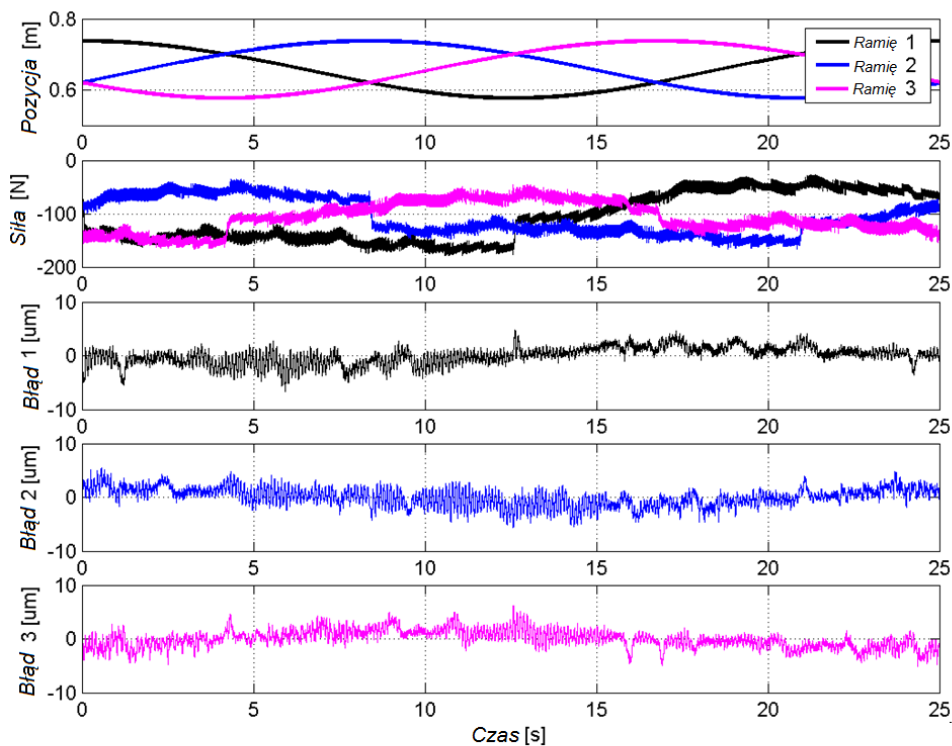
Oznaczenie wersji jednostki zmiennoprzecinkowej	**Typ elementu w strukturze FPGA**			
	LUTs	**Dedykowane rej. log.**	**Pamięć wew. [kB]**	**Bloki DSP**
MATH	14 126 (12%)	7 341 (6%)	159 (41%)	28 (7%)
MATH + CI	19 490 (17%)	8 243 (7%)	159 (41%)	28 (7%)
FPU_ALT	15 718 (14%)	9 355 (8%)	159 (41%)	36 (9%)
FPU_ALT + CI	20 981 (18%)	10 129 (9%)	159 (41%)	36 (9%)
FPU_CI	19 936 (18%)	7 935 (7%)	159 (41%)	28 (7%)
FPU_CI + CI	25 307 (22%)	8 703 (8%)	159 (41%)	28 (7%)

Tab. 3.8: Liczba wykorzystanych zasobów FPGA (Stratix III 3SL150]), do implementacji poszczególnych typów jednostek zmiennoprzecinkowych prototypu sterownika

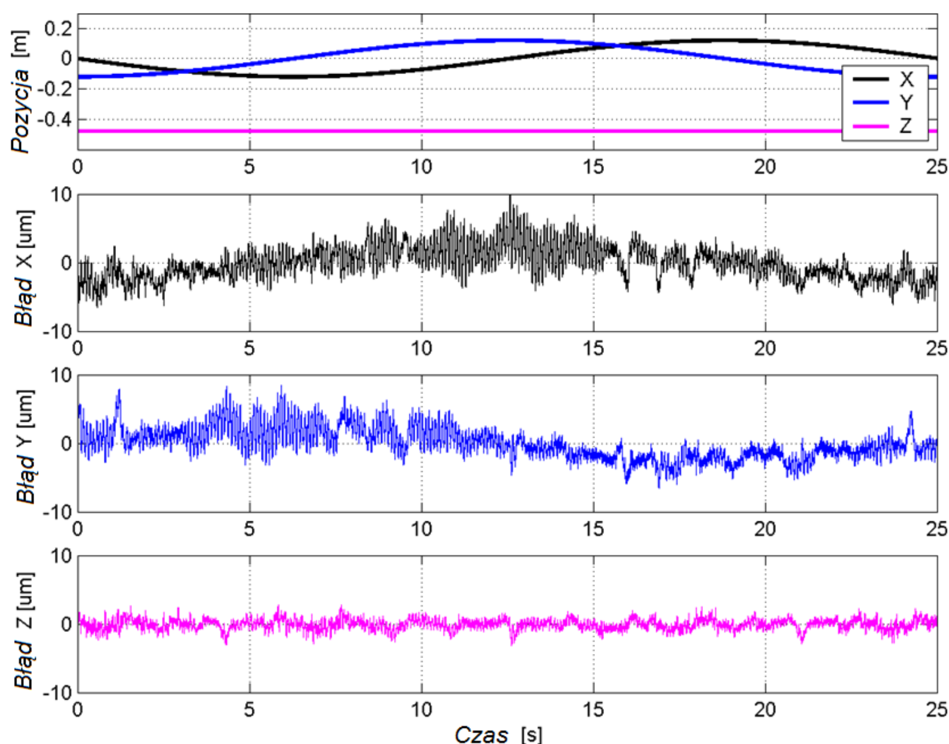
W ostatnim przeprowadzonym teście dla prototypu sterownika, zaimplementowano pełną architekturę systemu CNC, która realizuje część on-line algorytmu sterownika frezarki. Dla zadanego toru narzędzia, przeprowadzono pomiary na rzeczywistym obiekcie, prototypie frezarki 5-osiowej. Tor testowy dla manipulatora równoległego została przedstawiona na rys. 3.10. Chwilową prędkość posuwu (narzędzia) ustalono na wartość 2 m/min, pozwalającą na frezowanie w materiałach typu aluminium, dla obróbki wysokoobrotowej HSM. Wykorzystanie takiego toru do testów - okręgu o średnicy 300 mm, umiejscowionego w płaszczyźnie XY dla stałej wartości Z, pozwoliło w szczególności na przebadanie powtarzalności wykonania zadanej trajektorii i na analizę zachowania napędów dla stanu nawrotów (zmiany zwrotu wektora prędkości). Jednocześnie taka trajektoria zapewniła możliwie wysoki poziom bezpieczeństwa w przypadku utraty kontroli.



Rys. 3.10: Tor użyty do testu manipulatora równoległego frezarki



Rys. 3.11: Przebiegi czasowe pozycji, we współrzędnych złączowych dla poszczególnych ramion manipulatora równoległego (1), Przebiegi czasowe wartości sił zadawanych do układów komutacji napędów manipulatora rów. (2), Przebiegi czasowe błędów pozycji, we współrzędnych złączowych, dla poszczególnych ramion manipulatora rów. (3,4,5).



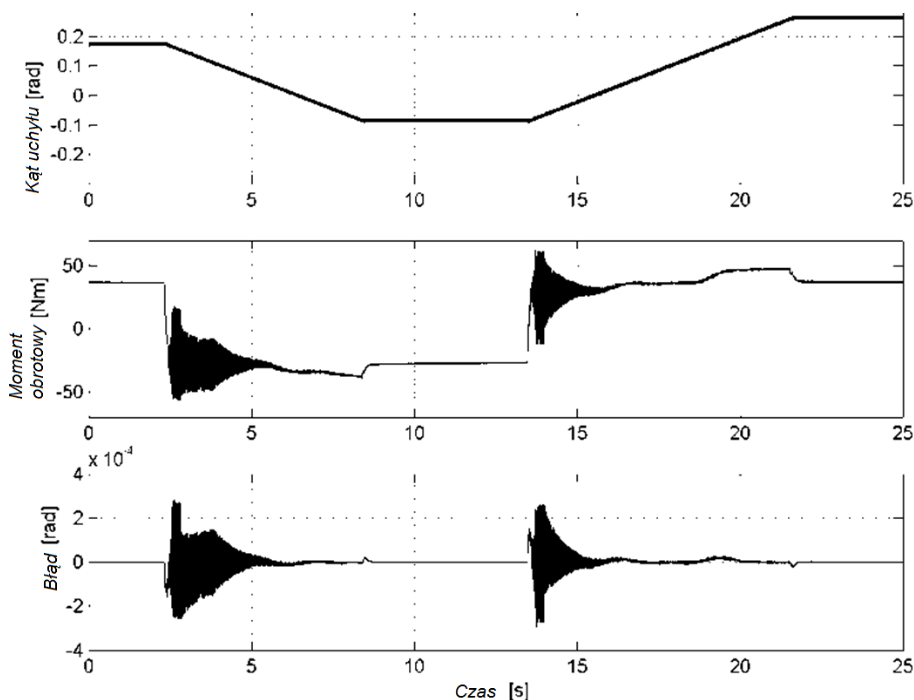
Rys. 3.12: Przebiegi czasowe pozycji, we współrzędnych kartezjańskich dla manipulatora rów. (1), Przebiegi czasowe błędów pozycji, we współrzędnych kartezjańskich, dla poszczególnych osi X (2), Y(3), Z(4) dla manipulatora rów.

Dla testów manipulatora szeregowego (stołu uchylno-obrotowego) tor zadany stanowił G-kod opisujący cykliczne sinusoidalne zmiany kąta obrotu i uchyłu. Wykorzystanie takiej trajektorii, podobnie jak w przypadku trajektorii dla manipulatora równoległego, zapewniło przeprowadzenie badań pod kątem powtarzalności odwzorowania zadawanej trajektorii przez sterownik frezarki.

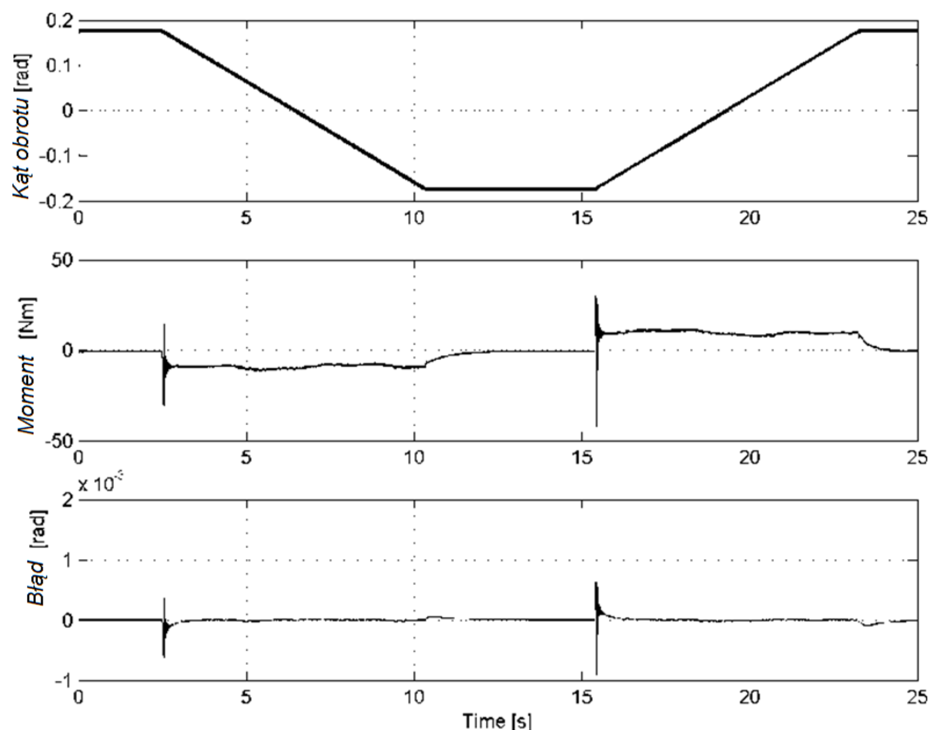
Wyniki eksperymentu przeprowadzonego dla pierwszego prototypu sterownika zostały przedstawione na rys. 3.11, rys. 3.12, rys. 3.13, rys. 3.14. Przebieg czasowe zmiany pozycji, we współrzędnych złączowych manipulatora równoległego, dla zadanego toru, zostały przedstawione na pierwszym przebiegu czasowym rys. 3.11. Kolejny wykres na rys. 3.11, prezentuje przebiegi czasowe zmiany wartości sił, zadawanych do układów komutacji poszczególnych napędów manipulatora równoległego. Dla doboru parametrów algorytmu sterowania dla układu śledzenia trajektorii istotna była rejestracja zmiany błędów pozycji dla poszczególnych ramion robota. Wyniki przeprowadzonych eksperymentów pozwoliły na analizę wpływu zmiany wartości tych parametrów na dokładność odwzorowania zadanej trajektorii. Ostatecznie dla przyjętych wartości parametrów sterownika - śledzenia trajektorii,

uzyskane maksymalne błędy pozycji, we współrzędnych kartezjańskich, nie przekroczyły wartości 10 μm, co zostało przedstawione na rys. 3.12.

Przebiegi czasowe zmiany kąta uchyłu (θ_4) i kąta obrotu (θ_5) manipulatora szeregowego we współrzędnych złączowych, dla zadanej trajektorii, zostały przedstawione odpowiednio: na pierwszym przebiegu czasowym rys. 3.14 i na pierwszym przebiegu czasowym rys. 3.14. Kolejne wykresy (rys. 3.13,(2) i rys. 3.14,(2)) prezentują przebiegi czasowe zmiany wartości momentów zadawanych do układu komutacji odpowiednich napędów stołu uchylno-obrotowego frezarki. Dla doboru parametrów algorytmu sterowania dla układu śledzenia trajektorii, istotna była rejestracja zmiany błędów kąta uchyłu (θ_4), rys. 3.13, (3), i obrotu (θ_5) rys. 3.14, (3). Wyniki przeprowadzonych eksperymentów pozwoliły na analizę wpływu zmiany wartości parametrów na dokładność odwzorowania zadanej trajektorii. Ostatecznie dla przyjętych wartości parametrów sterownika śledzenia trajektorii, uzyskany maksymalny błąd kąta uchyłu we współrzędnych kartezjańskich nie przekroczył wartości 300 μrad. Natomiast maksymalny błąd kąta obrotu we współrzędnych kartezjańskich nie przekroczył wartości 1,2 mrad.



Rys. 3.13: Przebiegi czasowe: (1) kąta uchyłu θ_4,
(2) wartości momentu zadawanego do układów komutacji napędu
(3) błąd orientacji uchyłu



Rys. 3.14: Przebiegi czasowe: (1) kąta obrotu θ_5,
(2) wartości momentu zadawanego do układów komutacji napędu,
(3) błąd orientacji dla manipulatora szeregowego

3.9 Bibliografia

[1] IEEE Computer Society: *IEEE Standard for Floating-Point Arithmetic*. IEEE STD 754-2019. IEEE; str. 1–84. doi:10.1109/IEEESTD.2019.8766229; ISBN 978-1-5044-5924-2; IEEE Std 754-2019.

[2] Grzegorz Góra: *Implementacja sprzętowa obliczeń kinematyki robota hybrydowego w układzie FPGA - Praca dyplomowa inżynierska*. Akademia Górniczo-Hutnicza; Kraków; 2013.

[3] Wikipedia - IEEE 754 [on-line]: *https://pl.wikipedia.org/wiki/IEEE_754*. Pobrane 04-2023.

[4] Konrad Gac: Sterownik robota hybrydowego do frezowania – Rozprawa doktorska; Akademia Górniczo-Hutnicza; Kraków; 2016.

[5] Meyer-Baese, U: *Digital Signal Processing with Field Programmable Gate Arrays*. Springer Netherlands; 2007.

[6] Maciej Petko, Konrad Gac, Grzegorz Karpiel, Grzegorz Góra: *Acceleration of parallel robot kinematic calculations in FPGA*. ICIT 2013; IEEE International Conference on Industrial Technology; Cape Town; South Africa; 2013.

[7] Sutte, J. P. Deschamps G.: *Decimal division: Algorithms and FPGA implementations*. r, Programmable Logic Conference (SPL); 2010 VI Southern; 67 - 72 str.; ISBN: 978-1-4244-6309-1; 24-26 March 2010.

[8] Terasic, DE2-115, [online]: *http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=China&No=550&FID=ce9e16096a25d9f0c23b70d8901daabd*. Pobrane: 04-2023.

[9] Madheswaran, T. Menakadevi and M.: *FPGA Implementation of Direct Digital Synthesizer using Pipelined Cordic Algorithm*. European Journal of Scientific Research; Vol.79; No.2, pp.269-278; 2012.

[10] Kabuo, H.: *Accurate rounding scheme for the Newton-Raphson method using redundant binary representation*. IEEE Transactions on Computers; Vol. 43; No. 1 pp. 43-51; 1994.

[11] Navdeep Prasha, Balwinder Singh: *FPGA Implementation of Pipelined CORDIC Sine, Cosine Digital Wave Generator*. David C. Wyld, et al. (Eds): CCSEA, SEA, CLOUD, DKMP, CS & IT 05; pp. 435–440; 2012.

[12] Vashkevich, Maxim: *FPGA Implementation of Short Critical Path CORDIC-Based Approximation of the Eight-Point DCT*. Belarusian State University of Informatics and Radioelectronics; 2011.

[13] Kabuo, H.: *Accurate rounding scheme for the Newton-Raphson method using redundant binary representation*. IEEE Transactions on Computers; Vol. 43; No. 1 pp. 43-51; 1994.

[14] M. Ercegovac, T. Lang and P. Montuschi.: *Very-high radix division with prescaling and selection by rounding*. IEEE Transactions on Computers; Vol. 43; No. 8, pp. 909-918; 1994.

[15] Chu, Y. Li and W.: *A new non-restoring square root algorithm and its VLSI implementations*. IEEE International Conference on Computer Design: VLSI in Computers and Processors; ICCD ’96 Proceedings; pp. 538 –544; 1996.

[16] *Implementation of single precision floating point square root on FPGAs*. Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines; pp. 226–232; IEEE; 1997.

[17] Konrad Gac, Grzegorz Karpiel, Maciej Petko: *FPGA based hardware accelerator for calculations of the parallel robot inverse kinematics*. ETFA'2012; 17th IEEE International conference on Emerging Technologies & Factory Automation; 2012.

[18] Konrad Gac, Grzegorz Karpiel, Maciej Petko: *Akceleracja sprzętowa sterowania robotem równoległym*. Projektowanie mechatroniczne : zagadnienia wybrane : praca zbiorowa / pod red. Tadeusza Uhla; Kraków; 2013.

[19] Intel (Altera), SoPC Builder - User guide [online]: *https://www.altera.com/en_US/pdfs/literature/ug/ug_sopc_builder.pdf*.
Pobrane 04-2023.
[20] Terasic, DE-3, Stratix III [online]: *https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=260&PartNo=1*.
Pobrane 04-2023.
[21] Altera: Stratix III 3SL150 Development Board - Reference Manual. Altera Corporation; May 2013.
[22] L. Luo, L. Wang, and J. Hu.: *On the modeling and analysis of an improved cnc interpolation algorithm*. Materials Science Forum, 626-627:459–464, 2009.

Rozdział 4 – Implementacja z wykorzystaniem języków opisu sprzętu HDL

4.1. VHDL

4.1.1 Język VHDL

VHDL (*ang. **V**ery High Speed Integrated Circuits **H**ardware **D**escription **L**anguage*) jest jednym z dwóch podstawowych języków opisu sprzętu używanych w komputerowym projektowaniu układów cyfrowych typu FPGA i ASIC. Począwszy od założeń, VHDL miał obsługiwać wszystkie poziomy cyklu projektowania układów elektronicznych. Wynika to jasno z przedmowy podręcznika referencyjnego języka (*ang. Language Rerefence Manual – IEEE-1076, 2008*), który definiuje język jako:

VHDL to formalna notacja przeznaczona do stosowania we wszystkich fazach tworzenia systemów elektronicznych. Ponieważ jest czytelny zarówno dla komputera, jak i dla człowieka, wspiera rozwój, weryfikację, syntezę i testowanie projektów układów elektronicznych [1].

Najważniejsze jest tu stwierdzenie „we wszystkich fazach". Oznacza to, że VHDL ma obejmować każdy poziom cyklu projektowego, od specyfikacji systemu aż do listy połączeń na poziomie technologii wykonania [1]. Początkowo język używany był do tworzenia dokumentacji oraz testowania i symulacji układów typu ASIC. Obecnie głównie wykorzystywany jest do implementacji w układach reprogramowalnych. System zdefiniowany przy pomocy języka VHDL można zasymulować, a także zaimplementować jego fizyczną realizację w ramach architektury układu FPGA lub CPLD.

Podstawowymi cechami języka VHDL są [1][2][3][6]:

- równoległość realizacji zadań – każdy ze zdefiniowanych modułów jest zaimplementowany w przestrzeni układu FPGA i działa niezależnie od innych elementów (w tym samym czasie);
- uniwersalność – VHDL jest językiem „przenośnym", może być stosowany przez różnych producentów układów FPGA/CPLD oraz daje możliwość przenoszenia projektów pomiędzy różnymi platformami sprzętowymi (rodzinami układów);
- opis hierarchiczny – projekt może być opisany na wielu poziomach; w opisie hierarchicznym projekt zbudowany jest z połączonych bloków o dużym stopniu złożoności, które z kolei zbudowane są z bloków o mniejszym poziomie złożoności, które zawierają w sobie jeszcze prostsze bloki i tak

dalej, aż dochodzi się do bloków podstawowych, którymi w przypadku układów cyfrowych są bramki logiczne;

- projektowanie na bazie istniejących modułów – możliwe jest korzystanie z bibliotek gotowych elementów oraz projektowanie nowych komponentów; nowy projekt może powstawać na bazie modułów już opracowanych i przetestowanych; projektanci wykorzystują poprzednio zdobyte doświadczenia przenosząc opracowane i poznane uprzednio rozwiązania do nowych projektów; proces ten określa się pochodzącym z języka angielskiego słowem *redesign*;
- możliwość sekwencyjnego opisu algorytmu – oznaczająca możliwość definiowania czynności wykonywanych jedna po drugiej, w sposób analogiczny jak w tradycyjnych językach programowania;
- zawiera użyteczne konstrukcje semantyczne – umożliwiające zwięzłą specyfikację złożonych układów cyfrowych.

W przypadku wykorzystania języka VHDL jako narzędzia do implementacji modułów sprzętowych w układach reprogramowalnych należy pamiętać o kilku jego ograniczeniach wynikających z fizycznej realizacji w układzie cyfrowym [1][3][5].

- Definicja języka VHDL zawiera konstrukcje, które nie są wspierane przez narzędzia syntezy kompilatorów lub nie są w ogólnym przypadku możliwe do realizacji w układzie cyfrowym. Częściowo są to konstrukcje wykorzystywane podczas symulacji np. instrukcje:
  wait for … ns;
  lub
  Q<=‘0’ after … ns;
  nie mogą być zsyntezowane – tzn. nie mogą zostać zaimplementowane w układzie fizycznym zgodnie z ich funkcją. Ich wykorzystanie ogranicza się wyłącznie do symulacji.
- Zaleca się wykorzystywanie elementów bibliotecznych – są wykonane w sposób optymalny dla danej docelowej architektury sprzętowej. Głównie odnosi się to do obsługi zewnętrznych pamięci lub interfejsów.
- Zaleca się stosowanie globalnych sygnałów zegarowych jako wejść zegarowych do przerzutników i rejestrów.

Język VHDL występuje w trzech podstawowych standardach [1]:

VHDL-1987	Pierwotny, oryginalny standard języka.
VHDL-1993	Dodano: rozszerzone identyfikatory, operatory przesunięć bitowych oraz xnor, bezpośrednie tworzenie instancji komponentów, ulepszono wykorzystanie portów wejścia/wyjścia w opisie symulacji.

VHDL-2008	Dodano pakiety do obsługi liczb w standardzie stałoprzecinkowym i zmiennoprzecinkowym. Dodano typy ogólne i pakiety, umożliwiając użycie typów ogólnych do definiowania pakietów i podprogramów wielokrotnego użytku. Ulepszono wykorzystanie trybów warunkowych. Dodano odczyt portów wyjściowych. Ujednolicono składnię.

4.1.2 Biblioteki i pakiety

Pakiet jest zbiorem definicji, które mogą być dostępne przez wiele projektów w tym samym czasie. Jest odrębną jednostką projektową w języku VHDL. Pakiet może zawierać definicje stałych, typy definiowane przez użytkownika, deklaracje komponentów oraz podprogramy [3]. Poniżej przedstawiona jest deklaracja pakietu, w którym znajduje się zdefiniowany przez użytkownika typ tablicowy oraz stała.

```
library ieee;
use ieee.std_logic_1164.all;

package package _name is

    type REG_TAB : is array(0 to 7) of std_logic_vector(31 downto 0);
    constant CONST_A : std_logic_vector(31 downto 0) := 32d"50000000";
    …

end package;
```

Aby dołączyć pakiet do projektu należy wywołać dyrektywę:

```
use work.package_name.all;
```

Bibliotekę można traktować jako kontener zawierający kompletne jednostki projektowe. Od pakietu różni ją to, że występuje w skompilowanej formie. Najważniejszą biblioteką jest biblioteka systemowa IEEE. Zawiera ona definicje standardowych typów danych. Odwołanie do biblioteki, a także polecenie umożliwiające dostęp do standardowego pakietu std_logic_1164 musi poprzedzać każdą jednostkę projektową. Uzyskuje się to poprzez wywołanie dyrektyw:

```
library ieee;
use ieee.std_logic_1164.all;
```

4.1.3 Jednostka projektowa

Podstawowym elementem reprezentującym funkcjonalny fragment układu cyfrowego jest jednostka projektowa zwana najczęściej modułem sprzętowym lub blokiem sprzętowym. Składa się z dwóch elementów opisanych przy pomocy dyrektyw ***entity*** oraz ***architecture***. Elementy ***entity*** i ***architecture*** funkcjonują w parach - pełny opis obwodu będzie na ogół zawierał oba te komponenty.

Fragment kodu opisany dyrektywą ***entity*** stanowi interfejs deklarowanego modułu. Zawiera jego nazwę oraz definicje portów (sygnałów) wejściowych i wyjściowych. Może również zawierać definicje parametrów (np. szerokość magistrali, liczba bitów rejestru, licznik pętli) i stałych [1][3]. Parametry modułu można zadeklarować wykorzystując klauzulę ***generic.*** Poniżej, opisany blok o nazwie halfadder posiada dwa wejścia (a oraz b) oraz dwa wyjścia (s oraz carry). Wszystkie wejścia i wyjścia są typu ***std_logic***.

Fragment kodu opisany dyrektywą ***architecture*** stanowi definicję zawartości bloku. Opisuje on sposób działania modułu sprzętowego lub jego budowę (strukturę), w szczególności definiuje zachowanie portów (sygnałów) wyjściowych. W poniższym przykładzie architekturę stanowi układ kombinacyjny, opisany przy pomocy bramek logicznych.

Możliwe jest występowanie wielu architektur posiadających wspólny interfejs (***entity***). Każda z nich może reprezentować inny sposób implementacji tego samego układu cyfrowego, np. behawioralny i strukturalny.

```
library ieee; -- dołączenie biblioteki ieee
use ieee.std_logic_1164.all; -- załadowanie bibliotek podstawowych typów logicznych

entity halfadder is -- deklaracja interfejsu
    port(
        a, b : in std_logic; -- deklaracja portów wejścia
        s, carry : out std_logic -- deklaracja portów wyjścia
        );
end entity;

architecture halfadder_arch of halfadder is -- deklaracja architektury
-- sekcja deklaracji obiektów: sygnałów, stałych, zmiennych, aliasów
begin
    s <= a xor b;
    carry <= a and b;
end architecture;
```

4.1.9 Identyfikatory i komentarze

Identyfikatory są nazwą obiektu w języku VHDL. Podstawowe zasady tworzenia identyfikatorów w języku VHDL [3][5]:

- identyfikatory mogą się składać jedynie z liter, cyfr i znaków podkreślenia ‘_’;
- nazwa musi się rozpoczynać od litery;
- nazwa nie może się kończyć znakiem ‘_’;
- niedozwolone są dwa znaki podkreślenia ‘__’ w nazwie;
- wielkość liter nie ma znaczenia.

Komentarze stosuje się w celu opisu i dokumentacji kodu, są one ignorowane przez kompilator. Klasyczny komentarz w języku VHDL rozpoczyna się od dwóch znaków ‘--‘, wszystko co występuje po nich w danej linijce jest ignorowane. W wersji VHDL-2008 można używać komentarzy analogicznie jak w języku C, czyli /* komentarz */. Dobrą praktyką jest umieszczanie na początku pliku oraz przed każdym modułem nagłówka, zawierającego podstawowe informacje dotyczące projektu lub modułu, w formie komentarza. Poniżej znajduje się przykład użycia komentarzy do opisu jednostki projektowej.

```
-- ------------------------------------------------------------------------------
-- Autor: G.Gora
-- Wersja: 1.1
-- Data modyfikacji: 30.10.2022
-- Opis: moduł sprzętowy półsumatora
-- ------------------------------------------------------------------------------

library ieee; -- dołączenie biblioteki ieee
use ieee.std_logic_1164.all; -- załadowanie bibliotek zawierających podstawowe typy logiczne

entity halfadder is -- deklaracja interfejsu
    port(
        a, b : in std_logic; -- deklaracja portów wejścia
        s, carry : out std_logic -- deklaracja portów wyjścia
        );
end entity;

architecture halfadder_arch of halfadder is -- deklaracja architektury
-- sekcja deklaracji obiektów: sygnałów, stałych, zmiennych, aliasów, itd.
begin
    s <= a xor b;
    carry <= a and b;
end architecture;
```

4.1.4 Porty wejścia/wyjścia

Rodzaj portu determinuje kierunek przepływu danych. W języku VHDL występuje pięć typów portów: in, out, inout, buffer i linkage. Jeśli rodzaj portu nie jest podany, domyślnie zostanie on zaklasyfikowany jako wejście [1].

Znaczenie poszczególnych typów [1][4][5][6]:

in jednokierunkowy port wejściowy – może zostać odczytany, nie można przypisać do niego wartości;
out jednokierunkowy port wyjściowy – można przypisać do niego wartość, nie może być odczytany;
inout port dwukierunkowy – można przypisać do niego wartość oraz może zostać odczytany; ten typ portu powinien być zastosowany jeśli mamy do czynienia z magistralą dwukierunkową (np. linia danych magistrali I2C);
buffer port wyjściowy, który może zostać odczytany wewnątrz projektowanego układu umożliwiając stosowanie wewnętrznego sprzężenia zwrotnego [2];
linkage nie podlega syntezie [1].

W przypadku wykorzystania języka VHDL jako narzędzia do implementacji modułów sprzętowych w układach reprogramowalnych, rekomendowane jest użycie:
in jako portu wejściowego;
out jako portu wyjściowego;
inout jako portu dwukierunkowego.

4.1.5 Sygnały, stałe, zmienne, aliasy

Obiekty w języku VHDL to elementy posiadające nazwę i przechowujące wartości określonego typu. Do obiektów m.in. należą [2]:

- sygnały (signal);
- stałe (constant);
- zmienne (variable);
- aliasy (alias).

Zasięg obiektów [2]:

- obiekty zadeklarowane w pakietach są widoczne w wszystkich projektach VHDL wykorzystujących te pakiety;
- obiekty zadeklarowane w interfejsie (***entity***) jednostki projektowej są dostępne dla wszystkich architektur z nią związanych;
- obiekty zadeklarowane w bloku architektury (***architecture***) są dostępne dla wszystkich konstrukcji wewnątrz tej architektury;
- obiekty zadeklarowane w procesie dostępne są jedynie wewnątrz procesu.

Najczęściej występującym obiektem w języku VHDL jest sygnał. Jest on wykorzystywany do komunikacji pomiędzy poszczególnymi elementami architektury. Rzeczywiste, fizyczne połączenia w systemie, często są reprezentowane poprzez sygnały [2]. Drugą możliwością syntezy sygnału jest utworzenie rejestru,

przeznaczonego do przechowywania tymczasowych wartości. Sposób deklaracji sygnału jest następujący:

signal nazwa_sygnalu : typ_danych;

Sygnały są deklarowane w sekcji deklaracji bloku architektury (pomiędzy słowem kluczowym ***is*** a ***begin***). Poniżej przykłady deklaracji sygnałów w architekturze modułu oraz przypisania sygnałom wartości z portów wejściowych modułu.

```
…

port(
        x3, x2, x1, x0 : in std_logic
    );
end entity;

architecture mod_arch of mod_name is
    signal a, b : std_logic;
    signal c : std_logic := ‘0’;
    signal d : std_logic_vector(3 downto 0) := “0000”;
begin

a <= x3;
b <= not x2;
c <= x1 and x0;
d <= x3&x2&x1&x0;

…

end architecture;
```

Istnieje możliwość zainicjowania sygnału, czyli podania jego początkowej wartości. Jednak wartość ta zostanie wykorzystana tylko w symulacji, ponieważ operacja ta nie jest syntezowana. Nie ma sprzętowej interpretacji wartości początkowej. Nie jest możliwe zainicjowanie wszystkich sygnałów w obwodzie ze znaną wartością po włączeniu zasilania [1][2]. Wszystkie przypisania wartości sygnałom odbywają się z opóźnieniem zależnym od technologii, w której system jest realizowany [2]. Wyjątkiem od tej sytuacji jest synteza sygnału do rejestru, wtedy rejestr zostanie zainicjowany wartością początkową.

Drugim obiektem wykorzystywanym w języku VHDL jest stała. Reprezentuje ona wartość, która nie może zostać zmieniona. Stosowanie stałych zwiększa czytelność kodu i ułatwia jego modyfikacje [2]. Sposób deklaracji stałej jest następujący:

constant nazwa_stalej : typ_danych := wartość;

Stałe mogą być deklarowane w pakietach, sekcji deklaracji bloku architektury (pomiędzy słowem kluczowym ***is*** a ***begin***) lub w sekcji deklaracji procesu (pomiędzy słowem kluczowym ***process(…)*** a ***begin***). Poniżej przykłady deklaracji stałych w architekturze modułu oraz wykorzystanie stałej jako wartości maksymalnej licznika.

```
…

port(
        clk : in std_logic;
        reset_n : in std_logic := ‘1’;
        …
    );
end entity;

architecture mod_arch of mod_name is

    constant CNT_MAX : unsigned(31 downto 0) := 32d”49999999”;
    constant IDX_MAX : std_logic_vector(3 downto 0) := 4x”12”;
    …

begin

    process(clk, reset_n)
    begin
        if(reset_n = '0')then
            cnt <= 32d"0";
        elsif rising_edge(clk) then
            if(cnt < CNT_MAX)then
                cnt <= cnt + 32d"1";
            else
                cnt <= 32d"0";
            end if;
        end if;
    end process;

…
```

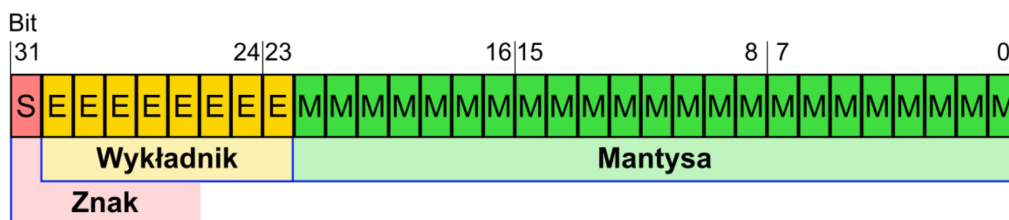
Kolejnym elementem wykorzystywanym w języku VHDL jest zmienna. Reprezentuje ona symboliczny element pamięciowy, w którym można przechowywać i modyfikować wartość. Jej funkcja jest zbliżona do funkcji zmiennej znanej z klasycznych języków programowania jak C/C++, jednak zmienna nie ma swojej bezpośredniej interpretacji fizycznej w układzie cyfrowym. Może być deklarowana i wykorzystywana jedynie w procesach (pomiędzy słowem kluczowym ***process(…)*** a ***begin***), a jej głównym zadaniem jest abstrakcyjny opis funkcjonowania systemu. Sposób deklaracji zmiennej jest następujący:

variable nazwa_zmiennej : typ_danych;

Poniżej przykłady deklaracji oraz wykorzystania zmiennej w procesie.

```
…

process(clk, reset_n)
    variable vCnt : integer := 0;
begin
    vCnt := vCnt + 1;

    if(vCnt = 5)then
        vCnt := 0;
    end if;

end process;

…
```

Ostatnim często wykorzystywanym elementem języka VHDL jest alias. Stanowi on alternatywą nazwę dla istniejących obiektów. Przykładem wykorzystania aliasu może być bezpośrednie odwołanie się do poszczególnych elementów liczby zmiennoprzecinkowej pojedynczej precyzji, która składa się z 23 bitów mantysy, 8 bitów wykładnika oraz 1 bitu znaku (rys. 4.1).



Rys. 4.1: Reprezentacja liczby zmiennoprzecinkowej pojedynczej precyzji [7]

Zakładając, że moduł, w którym deklarowany jest alias przyjmuje na wejście liczbę w formacie float, w postaci ciągu 32-bitów z magistrali danych o typie ***std_logic_vector(31 downto 0)***, za pomocą aliasu możliwe jest bezpośrednie

odwołanie się do poszczególnych składowych liczby bez konieczności definiowania pomocniczych sygnałów. Sposób deklaracji aliasu jest następujący:

```
alias nazwa_aliasu : typ_danych is nazwa_sygnalu;
```

Poniżej przykłady deklaracji sygnałów w architekturze modułu.

```
…

entity
    port(
        dataf : in std_logic_vector(31 downto 0);

        …

        );
end entity;

architecture float_arch of float_mod is

    alias sgn_float : std_logic is dataf(31);
    alias exp_float : std_logic_vector(7 downto 0) is dataf(30 downto 23);
    alias man_float : std_logic_vector(22 downto 0) is dataf(22 downto 0);

begin

…
```

4.1.6 Typy danych

Pewne typy danych w języku VHDL są predefiniowane. Oznacza to, że powinny być zdefiniowane w pakiecie ***standard*** i powinny stanowić bezpośredni element języka [1] [3]. Należą do nich, m.in.:

- ***boolean*** – zmienna logiczna, przyjmuje wartości TRUE lub FALSE;
- ***bit*** – przyjmuje wartość ‘0’ lub ‘1’;
- ***bit_vector*** – wektor wartości typu bit, np. „1010”;
- ***charakter*** – umożliwia używanie znaków alfanumerycznych np. ‘a’;
- ***string*** – reprezentuje ciąg znaków, np. ”ABCD”
- ***ingeter*** – reprezentuje liczby całkowite.

Podstawowy typ logiczny ***bit*** nie pozwala na przypisanie sygnałowi stanu wysokiej impedancji ‘Z’ (*ang. high Z*). W konsekwencji uniemożliwia to implementację m.in. dwukierunkowej magistrali (np. obsługi portu danych SDA magistrali I2C). Logika wielowartościowa posiada więcej typów niż tylko logiczne ‘0’ logiczna ‘1’. Pakiet ***std_logic_1164*** wchodzący w skład języka VHDL zawiera definicję typów ***std_logic*** (typ "resolved") oraz ***std_ulogic*** (typ "unresolved") o następujących wartościach [1][3][4][5]:

'U' wartość nie została zainicjalizowana;
'X' wartość była znana ale aktualnie nie można podać jej konkretnej wartości;
'0' logiczne 0;
'1' logiczna 1;
'Z' stan wysokiej impedancji;
'W' wartość była znana ale aktualnie nie można podać jej konkretnej wartości;
'L' słabe logiczne 0;
'H' słaba logiczna 1;
'-' wartość sygnału nie ma znaczenia.

Syntezie podlegają jedynie stany: logiczne 0, logiczna 1 oraz stan wysokiej impedancji. Pozostałe wartości mogą zostać wykorzystane jedynie w symulacji. Pakiet ***std_logic_1164*** zawiera również definicje typów ***std_logic_vector*** oraz ***std_ulogic_vector***, czyli wektorów składających się z elementów ***std_logic/std_ulogic***. Pakiet zawarty jest w bibliotece IEEE. Aby z niej skorzystać należy umieścić przed opisem jednostki projektowej poniższe dyrektywy:

```
library ieee;
use ieee.std_logic_1164.all;
```

Język VHDL zapewnia wiele sposobów przypisania wartości obiektom. Najpopularniejsze sposoby przypisania wartości 200 do sygnału y, widoczne są poniżej:

```
y <= "11001000"; -- bezpośrednio binarnie
y <= b"1100_1000"; -- binarnie z separatorem _
y <= 8b"1100_1000"; -- binarnie z podaniem długości magistrali
y <= (7 | 6 | 3 => '1', others => '0'); -- wyliczeniowo, podając wartości bitów
y <= x"C8"; -- szesnastkowo bez podania długości magistrali
y<= 8x"C8"; -- szesnastkowo z podaniem długości magistrali
y <= d"200"; -- dziesiętnie bez podania długości magistrali
y <= 8d"200"; -- dziesiętnie z podaniem długości magistrali
```

Poza predefiniowanymi typami danych oraz logiką wielowartościową ***std_logic_1164,*** można skorzystać z typów wektorowo-skalarnych, które są pomocne w przypadku obliczeń na wartościach ze znakiem oraz bez znaku. Są to:

- ***unsigned*** – tablica elementów ***std_logic*** interpretowana jako liczba całkowita bez znaku (w kodzie NKB), typ danych przeznaczony do wykonywania działań arytmetycznych i logicznych;

- ***signed*** - tablica elementów ***std_logic*** interpretowana jako liczba całkowita ze znakiem (w kodzie U2), typ danych przeznaczony do wykonywania działań arytmetycznych i logicznych.

Oba typy znajdują się w bibliotece ***numeric_std***. Poniżej przykład deklaracji typów ***unsigned*** oraz ***signed***.

```
library ieee;
use ieee.numeric_std.all;
…

signal cnt : unsigned(31 downto 0) := 32d”0”;
signal adc_value : signed(15 downto 0) := 16d”0”;
…
```

4.1.7 Tablice

Tablica to zbiór elementów tego samego typu. Elementy tablicy są dostępne poprzez indeks, który może być dowolną liczbą całkowitą nieujemną lub typem wyliczeniowym [1]. Aby użyć tablicy należy zadeklarować nowy, tablicowy typ danych, a następnie zadeklarować sygnał, zgodnie ze schematem poniżej:

type nazwa_typu_tablicowego ***is*** array (0 to N-1) of typ_elementów_tablicy;
signal nazwa_zmiennej_typu_tablicowego : nazwa_typu_tablicowego;

Odwołanie do elementów tablicy następuje poprzez podanie indeksu w nawiasach okrągłych. Poniżej przedstawione są sposoby deklaracji tablicy jedno- i dwuwymiarowej z elementami typu integer oraz sposób odwołania się do elementów tablicy.

```
-- Przykład deklaracji tablicy
type t_vector is array (0 to 3) of integer; -- wektor 4 elementowy
type t_matrix is array (0 to 3, 0 to 7) of integer; -- macierz 4 x 8

signal vector  : t_vector;
signal matrix  : t_matrix;

-- Przykład zapisu wartości do tablicy
vector(1)  <= 42;
matrix(0,2) <= 5;
```

4.1.8 Operatory

Operator	Przykład operacji	Opis
not	y <= not a;	negacja
and	y <= a and b;	and
nand	y <= a nand b;	nand
or	y <= a or b;	or
nor	y <= a nor b;	nor
xor	y <= a xor b;	xor
xnor	y <= a xnor b;	xnor
+	y <= a + b;	dodawanie
-	y <= a - b;	odejmowanie
*	y <= a * b;	mnożenie
/	y <= a / b;	dzielenie
mod	y <= a mod b;	wartość modulo
rem	y <= a rem b;	reszta z dzielenia
abs	y <= abs a;	wartość bezwzględna
**	y <= a ** b;	potęgowanie
=	if (a = b) then	równe
/=	if (a /= b) then	różne
>	if (a > b) then	większe
<	if (a < b) then	mniejsze
>=	if (a >= b) then	większe lub równe
<=	if (a <= b) then	mniejsze lub równe
sll	y <= a sll 1;	logiczne przesunięcie w lewo
srl	y <= a srl 1;	logiczne przesunięcie w prawo
sla	y <= a sla 1;	arytmetyczne przesunięcie w lewo
sra	y <= a sra 1;	arytmetyczne przesunięcie w prawo
rol	y <= a rol 1;	rotacja w lewo
ror	y <= a ror 1;	rotacja w prawo
&	y <= a&b;	łączenie

Operator & jest operatorem konkatenacji i umożliwia łączenie elementów (fragmentów) tablic w większe tablice. Poniżej przedstawione są przykłady wykorzystania operatora konkatenacji.

```
y <= ”00”&a(7 downto 2); -- przesunięcie logiczne w prawo o dwa miejsca;
y <= a(7)&a(7)&(7 downto 2); -- przesunięcie arytmetyczne w prawo dwa miejsca;
y <= a(1 downto 0)&a(7 downto 2); -- rotacja w prawo o dwa miejsca;
```

4.1.9 Funkcje

Funkcje i procedury stanowią dwa rodzaje podprogramów, które występują w języku VHDL [1][5]. Różnica pomiędzy nimi polega na zwracanej wartości: procedura nie zwraca żadnej wartości i w związku z tym nie może zostać użyta np. w instrukcjach przypisania, podczas gdy funkcja zwraca wartość określonego typu [2]. Podobnie jak w językach ogólnego przeznaczenia funkcja stanowi fragment kodu, realizujący pewne zadanie, który może być wielokrotnie wywoływany z różnych miejsc projektu. Każda funkcja przed wykorzystaniem musi zostać zdefiniowana. Sposób definiowania funkcji jest następujący:

```
function nazwa_funkcji(argumenty : in typ_danych) return typ_danych is
begin
    -- ciało funkcji
end function;
```

Poniżej przedstawiona jest deklaracja funkcji, realizującej układ kombinacyjny pozwalający na wyświetlenie cyfry na wyświetlaczu 7-segmentowym, oraz sposób jej wywołania w innym fragmencie kodu.

```
-- Przykład deklaracji funkcji
function bcd_to_hex(x : in unsigned) return std_logic_vector is
begin
    case x is
        when "0000" => return "1111110"; -- 0
        when "0001" => return "0110000"; -- 1
        when "0010" => return "1101101"; -- 2
        when "0011" => return "1111001"; -- 3
        when "0100" => return "0110011"; -- 4
        when "0101" => return "1011011"; -- 5
        when "0110" => return "1011111"; -- 6
        when "0111" => return "1110000"; -- 7
        when "1000" => return "1111111"; -- 8
        when "1001" => return "1111011"; -- 9
        when others => return "0000000"; -- Pozostałe
    end case;
end function;

…

-- Przykład wywołania funkcji
hex <= bcd_to_hex(x_value);
```

4.1.10 Komponenty

Komponent jest elementem hierarchicznej struktury opisu układu. Istnieje wiele powodów, dla których warto używać hierarchii w projektowaniu oraz implementacji układów cyfrowych. Przede wszystkim każdy komponent może być

zaprojektowany i przetestowany oddzielnie, zanim zostanie włączony do wyższych poziomów projektu [1]. Testowanie poszczególnych elementów systemu jest znacznie prostsze niż testowanie całego systemu, a w konsekwencji jest zwykle dokładniejsze i pozwala na wykrycie większej liczby błędów.

Dobrą praktyką jest gromadzenie komponentów w bibliotekach, co pozwala na ich użycie zarówno w innym miejscu w tym samym projekcie, jak również później w innych projektach. Jedną z największych zalet implementacji w układach reprogramowalnych jest to, że moduły sprzętowe są w większości przypadków niezależne od technologii (są niezależne funkcjonalnie, są zależne pod kątem maksymalnej częstotliwości pracy w przypadku układów sekwencyjnych oraz czasu propagacji sygnału w przypadku układów kombinacyjnych), dzięki czemu mogą zostać ponownie wykorzystane. Oznacza to, że z biegiem czasu poziom ponownego wykorzystania elementów uprzednio zaprojektowanych w każdym systemie wzrasta. Ważne jest, aby posiadać strategię gromadzenia użytecznych komponentów w bibliotekach do ponownego wykorzystania [1]. Pozwala to na znaczne skrócenie czasu projektowania.

Każdy komponent musi być wcześniej zadeklarowany i zdefiniowany jako osobna jednostka projektowa. Następnie aby wykorzystać moduł jako komponent w nadrzędnej jednostce projektowej należy zadeklarować jego użycie w sekcji deklaracji bloku architektury (pomiędzy słowem kluczowym ***is*** a ***begin***). Sposób deklaracji komponentu jest następujący:

```
component nazwa_komponentu is
    port(
        … -- porty wejścia/wyjścia
    );
end component;
```

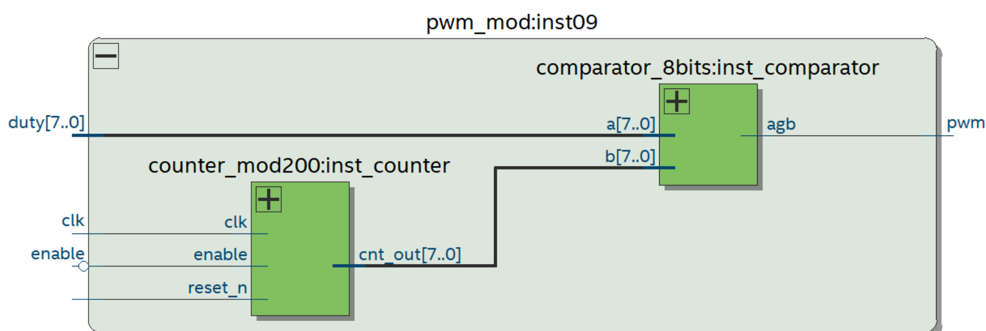
Jeśli moduł zawierał sekcję ***generic*** należy ją również umieścić w deklaracji komponentu. W praktyce deklaracja komponentu sprowadza się do skopiowania interfejsu jednostki projektowej (sekcji ***entity***) oraz zamianie słów kluczowych ***entity*** na ***component***.

Gdy komponent jest zadeklarowany może zostać użyty w sekcji głównej architektury modułu. Aby opisać wzajemne połączenia komponentu z obwodem wyższego poziomu, należy wykonać tzw. „mapowanie portów”. Sposób mapowania portów jest następujący:

```
nazwa_etykiety: nazwa_komponentu
port map(
        nazwa_portu_komponentu1 => nazwa_sygnału1,
        nazwa_portu_komponentu2 => nazwa_sygnału2,
        …
        nazwa_portu_komponentuN => nazwa_sygnałuN
        );
```

Nazwa etykiety jest dowolną, nadaną przez użytkownika nazwą instancji danego komponentu.

Projektowanie hierarchiczne z wykorzystaniem komponentów zostanie przedstawione na przykładzie prostego modulatora szerokości impulsów PWM. Główna jednostka projektowa składa się z dwóch komponentów: licznika modulo 200 oraz 8-bitowego komparatora. Na rys. 4.2 znajduje się schemat modulatora z nazwami portów wejścia/wyjścia, a poniżej przedstawiony jest fragment kodu w języku VHDL.



Rys. 4.2: Modulator PWM

```
--
-- Deklaracja jednostki projektowej licznika mod 200
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter_mod200 is
    port(
        clk , reset_n: in std_logic;
        enable : in std_logic;
        cnt_out : out unsigned(7 downto 0)
    );
end entity;

architecture  counter_mod200 of counter_mod200 is

    constant CNT_MAX : unsigned(7 downto 0) := 8d"199";
    signal cnt : unsigned(7 downto 0) := 8d"0";

begin

    process(clk, reset_n)
    begin
        if(reset_n = '0')then
            cnt <= 8d"0";
        elsif rising_edge(clk)then
```

```
                if(enable = '1')then
                    if(cnt < CNT_MAX)then
                        cnt <= cnt + 8d"1";
                    else
                        cnt <= 8d"0";
                    end if;
                else
                    cnt <= 8d"0";
                end if;
            end if;
        end process;

        cnt_out <= cnt;

end architecture;

-- ------------------------------------------------------------------------




--
-- Deklaracja jednostki projektowej8-bitowego komparatora
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity comparator_8bits is
    port(
        a, b : in unsigned(7 downto 0);
        agb : out std_logic
    );
end entity;

architecture comparator_8bits of comparator_8bits is
begin

    agb <= '1' when(a > b) else '0';

end architecture;

-- ------------------------------------------------------------------------









--
-- Deklaracja jednostki projektowej 8-bitowego komparatora
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pwm_mod is
    port(
        clk: in std_logic;
        reset_n: in std_logic := '1';
        enable : in std_logic;
        duty : in unsigned(7 downto 0);
```

```
        pwm : out std_logic
    );
end entity;

architecture  pwm_mod of  pwm_mod is

    -- Delkaracja komponentu - licznik mod 200
    component counter_mod200 is
        port(
            clk , reset_n: in std_logic;
            enable : in std_logic;
            cnt_out : out unsigned(7 downto 0)
        );
    end component;

    -- Deklaracja komponentu - 8-bit komparator
    component comparator_8bits is
        port(
            a, b : in unsigned(7 downto 0);
            agb : out std_logic
        );
    end component;

    -- Deklaracja sygnałów
    signal cnt_value : unsigned(7 downto 0);

begin

    -- Mapowanie portów licznika
    inst_counter : counter_mod200
    port map(
        clk => clk,
        reset_n => reset_n,
        enable => enable,
        cnt_out => cnt_value
        );

    -- Mapowanie portów komparatora
    inst_comparator : comparator_8bits
    port map(
        a => duty,
        b => cnt_value,
        agb => pwm
        );

end architecture;

-- -----------------------------------------------------------------------
```

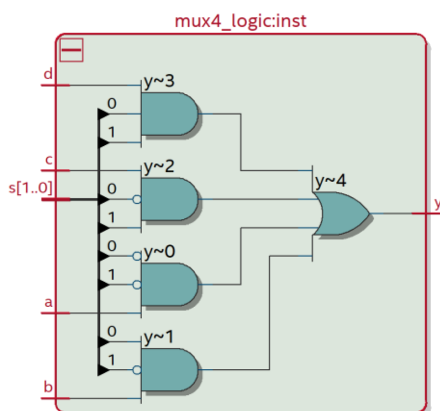
4.2 Implementacja układów cyfrowych

4.2.1 Układy kombinacyjne

Układy kombinacyjne to rodzaj układów cyfrowych, w którym stan wyjść zależy jedynie od stanu wejść [5][6]. W języku VHDL można je zaimplementować wykorzystując kilka różnych podejść. Podstawowym podejściem jest **wykorzystanie operatorów arytmetycznych i logicznych**. Zazwyczaj jest to poprzedzone procesem upraszczania przy pomocy np. tablic Karnaugh. Poniżej przedstawiony jest przykład

implementacji 1-bitowego multipleksera z czterema wejściami. Na rys. 4.3 widoczny jest rezultat interpretacji kodu przez kompilator – wynikowy schemat RTL.

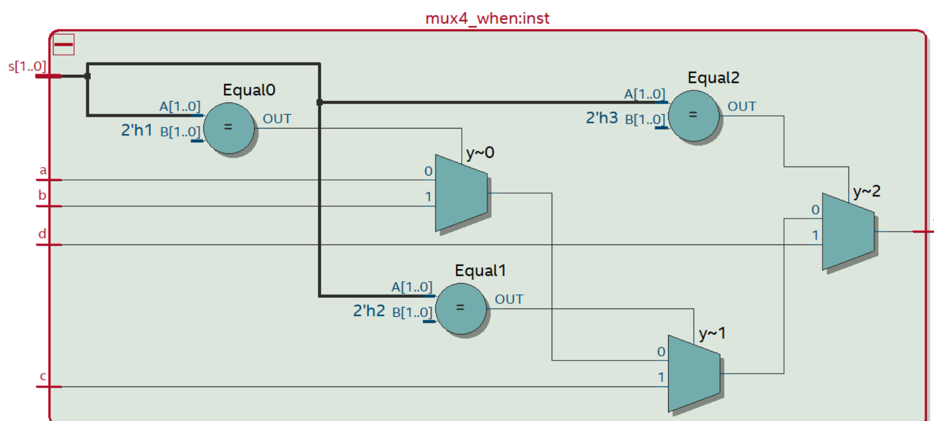
```
library ieee;
use ieee.std_logic_1164.all;

entity mux4_logic is
    port(
        a, b, c, d : in std_logic;
        s: in std_logic_vector(1 downto 0);
        y: out std_logic
    );
end entity;

architecture mux4_logic of mux4_logic is
begin

    y <= (a and not(s(1)) and not(s(0))) or
        (b and not(s(1)) and s(0)) or
        (c and s(1) and not(s(0))) or
        (d and s(1) and s(0));

end architecture;
```



Rys. 4.3: Schemat RTL 1-bitowego multipleksera zaimplementowanego przy użyciu podstawowych bramek logicznych

Drugim możliwym podejściem jest wykorzystanie **przypisania warunkowego**. W najprostszej wersji przypisanie warunkowe jest realizowane sprzętowo jako multiplekser, którego zadaniem jest wybór pomiędzy dwoma źródłami sygnału na podstawie wyrażenia warunkowego. W przypadku większej liczby warunków, do struktury dodawane są kaskadowo kolejne multipleksery, w wyniku czego powstaje złożona szeregowa sieć [1]. Powoduje to wydłużenie czasu propagacji sygnału pomiędzy wejściem a wyjściem oraz znaczne wykorzystanie zasobów sprzętowych układu. Z tego powodu, przypisanie warunkowe powinno być wykorzystywane jedynie w przypadku występowania niewielkiej liczby warunków.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux4_when is
	port(
		a, b, c, d : in std_logic;
		s: in std_logic_vector(1 downto 0);
		y: out std_logic
	);
end entity;

architecture mux4_when of mux4_when is
begin

	y <= d when(s = "11") else
		c when(s = "10") else
		b when(s = "01") else
		a;

end architecture;
```



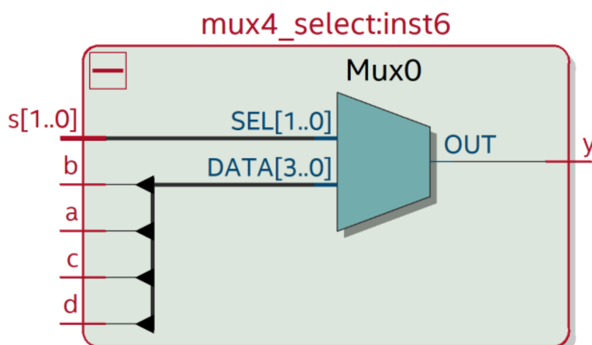
Rys. 4.4: Schemat RTL 1-bitowego multipleksera zaimplementowanego przy użyciu przypisania warunkowego

Występowanie kaskadowo połączonych multiplekserów powoduje, że warunki sprawdzane są w sposób sekwencyjny, dzięki czemu można uzyskać efekt priorytetów. Zawsze ostatni warunek musi być zakończony słowem kluczowym ***else***, tak aby jedno z wyrażeń źródłowych każdorazowo mgło zostać przypisane do celu [1]. Na rys. 4.4 przedstawiony schemat RTL będący rezultatem zastosowania przypisania warunkowego. Widoczne są trzy multipleksery, z których każdy odpowiada za jeden warunek ***when(…) else***.

Kolejną strukturą języka VHDL, która może posłużyć do zaimplementowania układu kombinacyjnego jest **przypisanie selektywne**. Struktura umożliwia wybranie jednego z wielu wyrażeń źródłowych na podstawie warunku. Główna różnica pomiędzy przypisaniem warunkowym, a selektywnym polega na tym, że to ostatnie

tworzy warunki poprzez porównanie wartości sygnału referencyjnego z wieloma wzorcami. Wszystkie warunki mają równy priorytet i wzajemnie się wykluczają [1]. Kolejność występowania warunków nie ma znaczenia, nie występuje tutaj efekt priorytetu. Wynikiem działania przypisania selektywnego jest implementacja układu kombinacyjnego w postaci multipleksera z wieloma wejściami. Dzięki temu, w większości przypadków czas propagacji sygnału będzie krótszy, a wykorzystanie zasobów sprzętowych mniejsze w porównaniu do implementacji z wykorzystaniem konstrukcji przypisania warunkowego. Poniżej przedstawiony jest moduł sprzętowy zaimplementowany z wykorzystaniem struktury przypisania selektywnego, a na rys. 4.5 widoczny jest schemat RTL multipleksera zaprojektowanego przy użyciu struktury ***with … select***.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux4_select is
    port(
        a, b, c, d : in std_logic;
        s: in std_logic_vector(1 downto 0);
        y: out std_logic
    );
end entity;

architecture mux4_select of mux4_select is
begin

    with s select
    y <= d when("11"),
        c when("10"),
        b when("01"),
        a when others;

end architecture;
```



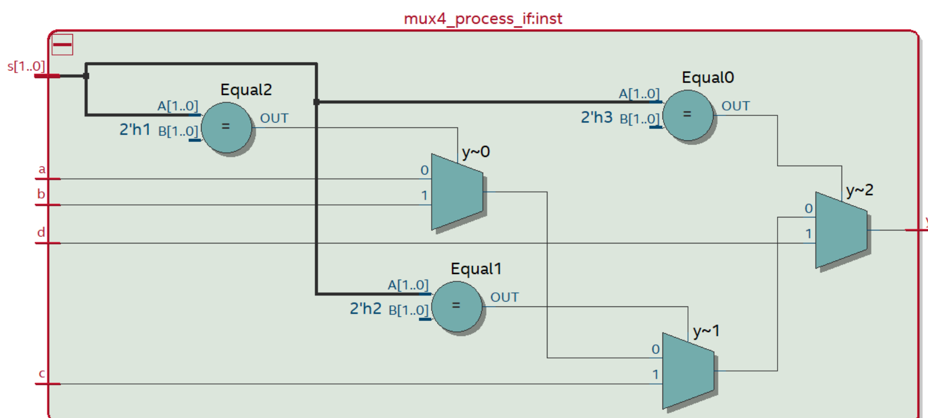
Rys. 4.5: Schemat RTL 1-bitowego multipleksera zaimplementowanego przy użyciu przypisania selektywnego

Następne podejście pozwalające na implementację układów kombinacyjnych wykorzystuje instrukcję procesu. Proces należy do behawioralnych sposobów implementacji występujących w języku VHDL. Nie opisuje on struktury wewnętrznej projektowanego obwodu lecz jego „zachowanie”. Projektant opisuje funkcjonalność (działanie) obwodu traktując go jako czarną skrzynkę, bez znajomości wewnętrznej budowy obiektu.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux4_process_if is
    port(
        a, b, c, d : in std_logic;
        s: in std_logic_vector(1 downto 0);
        y: out std_logic
    );
end entity;

architecture mux4_process_if of mux4_process_if is
begin

    process(a, b, c, d, s)
    begin
        if(s = "11")then
            y <= d;
        elsif(s = "10")then
            y <= c;
        elsif(s = "01")then
            y <= b;
        else
            y <= a;
        end if;
    end process;

end architecture;
```

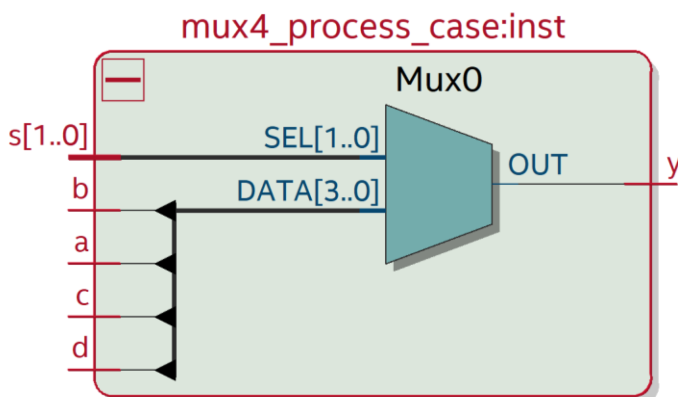
Struktura procesu składa się z listy wrażliwości, czyli zbioru sygnałów, które powodują „wykonanie” instrukcji sekwencyjnych umieszczonych wewnątrz procesu, gdy tylko którykolwiek z tych sygnałów zmieni stan. Wewnątrz procesu mogą być użyte instrukcje warunkowe znane z klasycznych języków programowania jak ***if*** oraz ***case***. W sekcji deklaracji procesu mogą być umieszczone deklaracje stałych oraz zmiennych, widoczne w obrębie danego procesu.

Powyżej przedstawiony jest moduł sprzętowy zaimplementowany z wykorzystaniem struktury procesu oraz instrukcji warunkowych ***if … elsif … else***, a na rys. 4.6 widoczny jest powstały na jego bazie schemat RTL multipleksera. Jak widać w tym przypadku kompilator zinterpretował kod w taki sposób, że struktura bloku sprzętowego jest taka sama jak dla przypisania warunkowego. Obie struktury mają pewne cechy wspólne ponieważ pozwalają na osiągnięcie efektu priorytetów podczas sprawdzania warunków.



Rys. 4.6: Schemat RTL 1-bitowego multipleksera zaimplementowanego przy użyciu struktury procesu oraz instrukcji warunkowej if

Poniżej przedstawiony jest moduł sprzętowy zaimplementowany również z wykorzystaniem struktury procesu, lecz jako instrukcje warunkowe została wykorzystana inareukxj ***case***. Na rys. 4.7 widoczny jest efekt interpretacji kompilatora w postaci schematu RTL. Główna różnica pomiędzy instrukcjami warunkowymi ***if…elsif…else***, a strukturą ***case*** polega na tym, że ta ostatnia tworzy warunki poprzez porównanie wartości sygnału referencyjnego z wieloma wzorcami, podobnie jak w przypadku przypisania selektywnego. Wszystkie warunki mają równy priorytet i wzajemnie się wykluczają. Kolejność występowania warunków nie ma znaczenia.



Rys. 4.7: Schemat RTL 1-bitowego multipleksera zaimplementowanego przy użyciu struktury procesu oraz instrukcji warunkowej case

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity mux4_process_case is
    port(
        a, b, c, d : in std_logic;
        s: in std_logic_vector(1 downto 0);
        y: out std_logic
    );
end entity;

architecture mux4_process_case of mux4_process_case is
begin

    process(a, b, c, d, s)
    begin
        case s is
            when "11" => y <= d;
            when "10" => y <= c;
            when "01" => y <= b;
            when others => y <= a;
        end case;
    end process;

end architecture;
```

Jak widać na powyższych przykładach sposób opisu układu wpływa na jego strukturę, a w konsekwencji będzie miał również bezpośredni wpływ na liczbę wykorzystanych zasobów sprzętowych oraz szybkość działania układu.

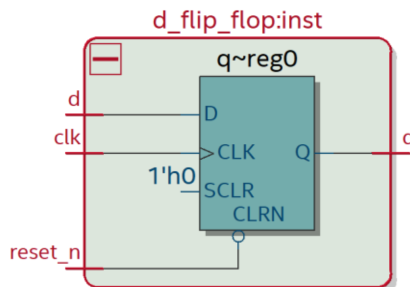
4.2.2 Układy sekwencyjne

Układ sekwencyjny to rodzaj układu cyfrowego charakteryzujący się tym, że stan wyjść zależy od stanu wejść oraz od poprzedniego stanu układu. Informacja o stanie poprzednim (z poprzedniego cyklu zegarowego) przechowywana jest w rejestrach, których fizyczną interpretacją są przerzutniki. Podstawową strukturą do opisu układu sekwencyjnego jest proces, w którym na liście wrażliwości umieszczony został sygnał zegarowy. W konsekwencji na każde narastające (lub opadające) zbocze zegara zostanie „wykonany" sekwencyjnie algorytm opisany w głównej części procesu. Ogólna struktura procesu wywoływanego na narastające zbocze sygnału zegarowego jest następująca:

```
process(clk, reset_n)
    -- deklaracja zmiennych procesu
begin
    if(reset_n = '0')then
        -- stan po resecie
    elsif rising_edge(clk) then
        -- główna część procesu
        -- wykonywana zawsze na narastające zbocze zegara
    end if;
end process;
```

Najprostszym elementem sekwencyjnym jest przerzutnik typu D (*ang. D flip-flop*). Na każde zbocze narastające wartość z wejścia d „przepisywana" zostaje na wyjście q. Wartość ta zostaje podtrzymana (zapamiętana) do momentu wystąpienia kolejnego zbocza narastającego sygnału zegarowego. Przerzutnik posiada dodatkowe, asynchroniczne wejście resetujące (reset_n), aktywne stanem niskim. Realizacja przerzutnika w kodzie VHDL widoczna jest poniżej, a rys. 4.8 przedstawia jego schemat RTL.

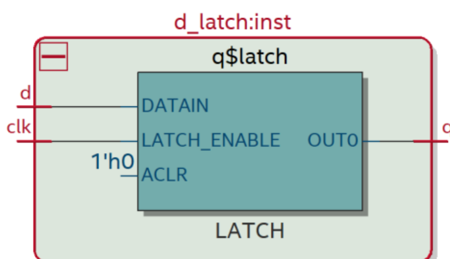
```
…
process(clk, reset_n)
begin
    if(reset_n = '0')then
        q <= '0';
    elsif rising_edge(clk) then
        q <= d;
    end if;
end process;
…
```



Rys. 4.8: Schemat RTL przerzutnika D

Innym podstawowym elementem układów cyfrowych jest zatrzask (*ang. D latch*), reagujący na stan, a nie zbocze sygnału zegarowego. Poniżej widoczny jest fragment kodu w języku VHDL realizujący zatrzask D oraz schemat RTL (rys. 4.9).

```
…
process(clk, d)
begin
    if (clk = '1') then
        q <= d;
    end if;
end process;
…
```

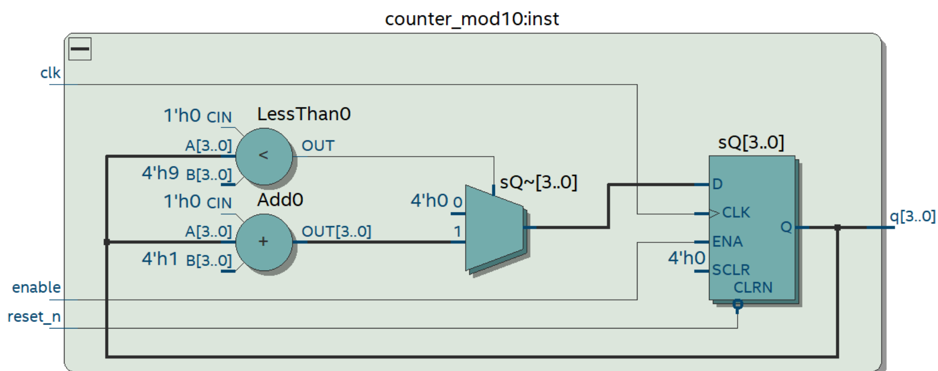


Rys. 4.9: Schemat RTL zatrzasku D

Kolejny przykład przedstawia moduł sprzętowy licznika modulo 10. Do zaprojektowania licznika wykorzystano behawioralny sposób implementacji. Realizacja sprzętowa modułu w postaci schematu RTL, widoczna jest na rys. 4.10. Można na niej wyróżnić:

- **przerzutniki D** – do zapamiętania wartości aktualnej licznika;
- **sumator** – realizujący inkrementację wartości sq (***sq <= sq + 4d"1"***);
- **komparator** – sprawdzający warunek mniejszości (***if(sq < 9)then***);
- **multiplekser** – pozwalający na wybór wartości z sumatora lub stałej (0), na podstawie warunku pochodzącego z komparatora.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter_mod10 is
	port(
		clk , reset_n: in std_logic;
		enable : in std_logic;
		q : out unsigned(3 downto 0)
	);
end entity;

architecture  counter_mod10 of  counter_mod10 is

	signal sq : unsigned(3 downto 0) := 4d"0";

begin

	process(clk, reset_n)
	begin
		if(reset_n = '0')then
			sq <= 4d"0";
		elsif rising_edge(clk) then
			if(enable = '1')then
				if(sq < 9)then
					sq <= sq + 4d"1";
				else
					sq <= 4d"0";
				end if;
			end if;
		end if;
	end process;

	q <= sq;

end architecture;
```



Rys. 4.10: Schemat RTL licznika modulo 10

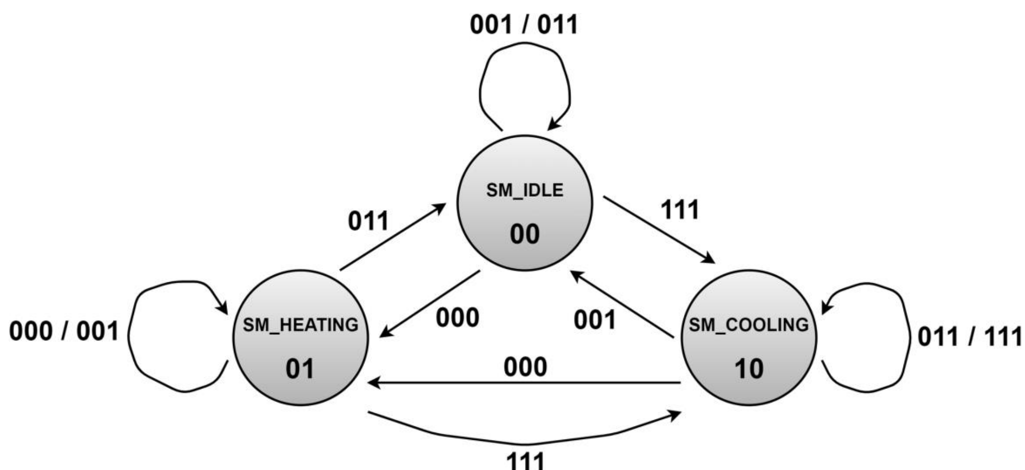
Ostatni przykład implementacji układu sekwencyjnego to maszyna stanu (automat Moore'a). Zostanie on zaprezentowany na przykładzie termostatu. Zadaniem termostatu jest utrzymanie temperatury 22 °C z histerezą ±2 °C. Jeśli temperatura spadnie poniżej 20 °C to maszyna stanu powinna włączyć grzanie i wyłączyć je kiedy temperatura wzrośnie do (nieco powyżej) 22 °C. Natomiast jeśli temperatura wzrośnie powyżej 24 °C to maszyna stanu powinna włączyć chłodzenie i wyłączyć je kiedy temperatura spadnie (nieco poniżej) 22 °C. Fluktuacja temperatury pomiędzy 20-24 °C nie powinna powodować włączenia grzania ani chłodzenia. Wejściami do automatu są 3 cyfrowe (1-bitowe) czujniki temperatury, których stan wysoki (logiczna 1) oznacza, że temperatura jest wyższa niż ustawiony poziom, a stan niski (logiczne 0) oznacza, że temperatura jest niższa niż ustawiony poziom. Progi czujników ustawiono dla temperatur 24 °C (czujnik a), 22 °C (czujnik b) oraz 20 °C (czujnik c).

Temperatura	Wskazania czujników		
	a	b	c
24 °C	1	1	1
22 °C	0	1	1
20 °C	0	0	1
	0	0	0

Wskazania czujników temperatury (kolejno a b c), dla poszczególnych przedziałów temperatury:

> 24 °C – 111;
22 °C ÷ 24 °C – 011;
20 °C ÷ 22 °C – 001;
< 20 °C – 000.

Na rys. 4.11 widoczny jest graf opisujący działanie termostatu. Automat składa się z trzech stanów: SM_IDLE (bezczynności – wyłączone grzanie, wyłączone chłodzenie), SM_HEATING (włączone grzanie) oraz SM_COOLING (włączone chłodzenie). Powyższe stany zdefiniowane są jako elementy typu wyliczeniowego zadeklarowanego przez użytkownika. Aktualny stan automatu przechowywany jest w rejestrze (sygnale) sm_curstate. Kolejny stan automatu określony jest na podstawie logiki przejść automatu, która stanowi układ kombinacyjny bazujący na stanie aktualnym oraz wartościach sygnałów wejściowych. W tym wypadku logika ta zdefiniowana jest przy pomocy struktury ***process***.



Rys. 4.11: Graf opisujący termostat

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ThermoSM is
	port(
		clk : in std_logic;
		reset_n : in std_logic := '1';

		a, b, c : in std_logic;
		enable_heating : out std_logic;
		enable_cooling : out std_logic
);
end entity;


architecture ThermoSM of ThermoSM is

	type THERMO_STATE_MACHINE is (SM_IDLE, SM_HEATING, SM_COOLING);
	signal sm_curstate, sm_nextstate : THERMO_STATE_MACHINE := SM_IDLE;

begin


	-- Rejestry (przerzutniki) do przechowywania aktualnego stanu
	-- ---------------------------------------------------------------------------
	process(clk, reset_n)
	begin
		if(reset_n = '0')then
			sm_curstate <= SM_IDLE;
		elsif rising_edge(clk) then
			sm_curstate <= sm_nextstate;
		end if;
	end process;




	-- Logika przejść automatu
```

```
    -- ---------------------------------------------------------------------------
    process(sm_curstate, a, b, c)
    begin
        case sm_curstate is

            when SM_IDLE =>
                if((a = '0') and (b = '0') and (c = '0'))then
                    sm_nextstate <= SM_HEATING;
                elsif((a = '1') and (b = '1') and (c = '1'))then
                    sm_nextstate <= SM_COOLING;
                else
                    sm_nextstate <= SM_IDLE;
                end if;

            when SM_HEATING =>
                if((a = '0') and (b = '1') and (c = '1'))then
                    sm_nextstate <= SM_IDLE;
                elsif((a = '1') and (b = '1') and (c = '1'))then
                    sm_nextstate <= SM_COOLING;
                else
                    sm_nextstate <= SM_HEATING;
                end if;

            when SM_COOLING =>
                if((a = '0') and (b = '0') and (c = '1'))then
                    sm_nextstate <= SM_IDLE;
                elsif((a = '0') and (b = '0') and (c = '0'))then
                    sm_nextstate <= SM_HEATING;
                else
                    sm_nextstate <= SM_COOLING;
                end if;

        end case;
    end process;



    -- Logika wyjść automatu
    -- ---------------------------------------------------------------------------
    enable_heating <= '1' when(sm_curstate = SM_HEATING) else '0';

    enable_cooling <= '1' when(sm_curstate = SM_COOLING) else '0';


end architecture;
```

Logika wyjść automatu definiująca sygnały do sterowania włączeniem grzania i chłodzenia realizowana jest przy pomocy struktur przypisania warunkowego.

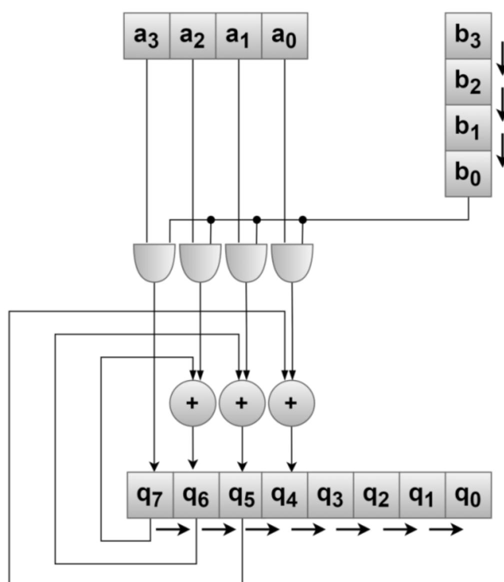
4.3 Implementacja obliczeń stałoprzecinkowych

4.3.1 Dodawanie i odejmowanie

Dodawanie i odejmowanie należy do podstawowych operacji arytmetycznych i podlega bezpośredniej syntezie przy użyciu elementów logicznych (LE/ALM). Długości (ilość bitów) wektorów stanowiących argumenty operacji mogą być dowolnie definiowane przez użytkownika. Operacje mogą być wykonywane na liczbach ze znakiem jak i bez znaku. Jeśli użytkownik chce zdefiniować typ liczby, może zadeklarować sygnał/zmienną typu ***signed*** lub ***unsigned***. Oba typy znajdują się w bibliotece ***numeric_std***. Operacje dodawania i odejmowania mogą być także wykonywane na wektorach ***std_logic_vector***, wtedy jednak należy dołączyć jedną z bibliotek ***std_logic_unsigned*** lub ***std_logic_signed***.

4.3.2 Mnożenie

Mnożenie w układach FPGA może być realizowane wykorzystując jedno z dwóch podejść: zastosowanie dedykowanych do tego celu bloków DPS (tzw. wbudowanych mnożarek) lub implementacja wykorzystując elementy logiczne (LE/ALM). Pierwsze podejście jest łatwiejsze i szybsze w implementacji. Jednak ograniczona ilość bloków DSP w układzie, sprawia, że w niektórych aplikacjach korzystne jest zastosowanie do implementacji operacji mnożenia, podstawowych zasobów układu FPGA, czyli elementów logicznych.



Rys. 4.12: Schemat blokowy 4-bitowego mnożenia sekwencyjnego

Projektowanie układu mnożącego w układzie reprogramowalnym jest kompromisem pomiędzy szybkością działania bloku sprzętowego a zajętością zasobów logicznych układu [8]. Najczęściej można wyróżnić dwie podstawowe metody: mnożenie równoległe – stosowane gdy kluczowe są parametry częstotliwościowe i należy dostosować układ mnożący pod kątem skrócenia czasu obliczeń oraz mnożenie sekwencyjne.

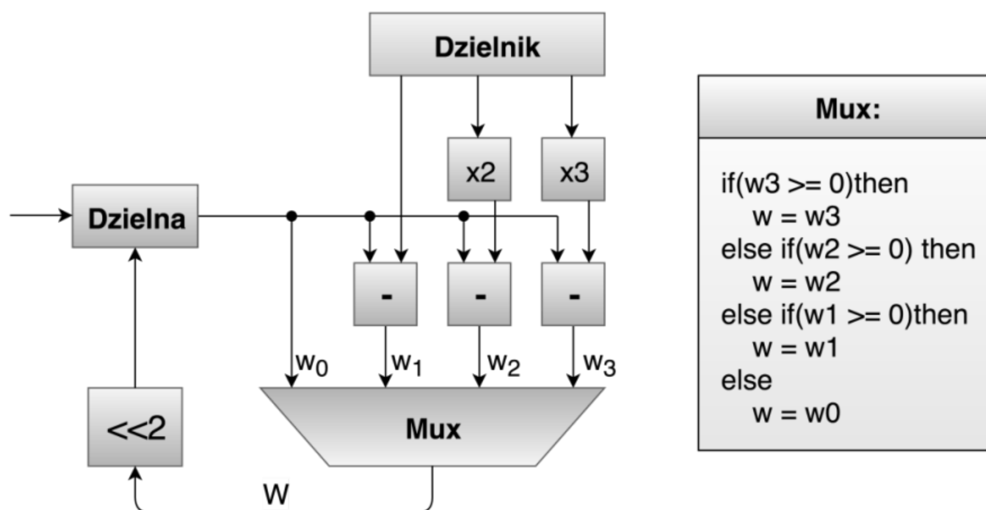
Mnożenie sekwencyjne pozwala zaoszczędzić zasoby sprzętowe układu FPGA kosztem wydłużenia czasu otrzymania wyniku. Najczęściej wykorzystywaną metodą mnożenia sekwencyjnego jest metoda przesuń i dodaj (*ang. shift and add*) [8][9]. Danymi wejściowymi w mnożeniu sekwencyjnym mogą być dwie N-bitowe liczby ze znakiem przedstawione w kodzie uzupełnień do dwóch U2, lub bez znaku przedstawione w reprezentacji NKB. Wynikiem mnożenia jest liczba o długości (ilości bitów) wektora równa sumie długości argumentów. W tej metodzie, iloczyn logiczny argumentu **a** (mnożnika) oraz jednego bitu argumentu **b** (mnożnej) stanowi częściowy element wyniku końcowego. W każdym kroku iteracji otrzymany rezultat cząstkowy należy dodać do otrzymanych rezultatów z poprzednich kroków iteracji, a następnie przesunąć o jeden bit w prawo [10]. Po liczbie cykli wynikających z długości mnożonych liczb otrzymany zostaje wynik końcowy. Na rys. 4.12 przedstawiony został schemat blokowy mnożenia sekwencyjnego wykorzystującego metodę *shift-and-add* dla czterobitowych wartości wejściowych.

Mnożenie równoległe wykorzystuje większą ilość zasobów sprzętowych układu FPGA jednak pozwala znacząco skrócić czas operacji. W tej metodzie iloczyny cząstkowe mnożnika oraz przesuniętej o odpowiednią liczbę bitów mnożnej stanowią element końcowej sumy, która jest wynikiem całej operacji mnożenia.

4.3.3 Dzielenie

Kolejnym podstawowym działaniem arytmetycznym jest dzielenie. Najczęściej do implementacji tej operacji używany jest jeden z grupy algorytmów Radix 2n o odpowiednio dobranej liczby bitów, zapewniającej kompromis pomiędzy liczbą cykli koniecznych do uzyskania wyniku, a wykorzystaniem zasobów sprzętowych. Algorytm Radix polega na sekwencyjnym wyznaczaniu największej nieujemnej różnicy wartości tymczasowej dzielnej oraz dzielnika lub jego wielokrotności. Poprzez porównywanie dzielnej z wielokrotnościami dzielnika można uzyskać wynik w mniejszej liczbie taktów zegara, co pociąga jednak za sobą wzrost liczby wykorzystywanych elementów logicznych [11]. Wartość tymczasowa to dzielna w pierwszym kroku iteracji oraz wartość wyznaczana z poprzedniej fazy dla każdego następnego etapu. Wynik pojedynczej sekwencji staje się argumentem kolejnego kroku poprzez przesunięcie bitowe o n-bitów otrzymanej różnicy. Wybór składnika, który należy odjąć od dzielnej dokonuje się poprzez określenie największej nieujemnej różnicy wartości tymczasowej dzielnej i wielokrotności dzielnika. Można

to zaobserwować analizując działanie bloku funkcyjnego multipleksera (rys. 4.13). Wybór ten wyznacza jednocześnie wartość kolejnych n-bitów wyniku rozpoczynając od pozycji najbardziej znaczącego bitu (*ang. Most Significant Bit, MSB*). Proces jest kontynuowany do momentu uzyskania żądanej długości bitowej wyniku, a wartość chwilowa po zakończeniu traktowana jest jako reszta z dzielenia [12].

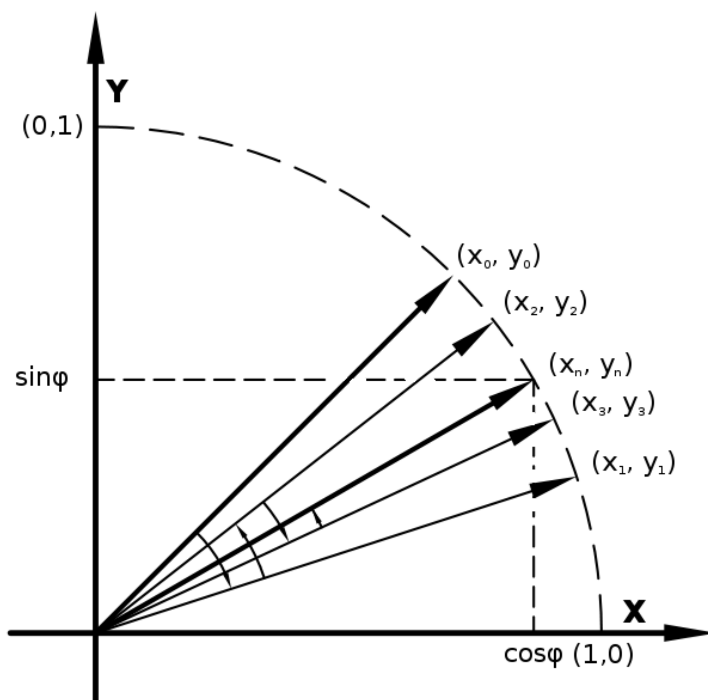


Rys. 4.13: Schemat blokowy dzielenia sekwencyjnego Radix 4

4.3.4 Funkcje trygonometryczne

W systemach wbudowanych (*ang. Embedded System*), działających w trybie on-line wyznaczanie wartości funkcji trygonometrycznych jest zadaniem wymagającym wykorzystania znacznych zasobów sprzętowych (elementy logiczne, pamięć) lub czasu obliczeń. Powszechną praktyką, gdy wartości te mogą być wyznaczana z niską dokładnością (do kilkunastu bitów) staje się tablicowanie LUT czyli, wykorzystanie struktury w formie tablicy do przechowywania wcześniej obliczonych wyników. Daje to oszczędność czasu wymaganego do ich wyznaczenia kosztem większego zużycia pamięci. Niestety rozmiar tablicy szybko rośnie wraz z długością wektora danych, dlatego ta metoda staje się mało efektywna dla wysokich rozdzielczości. Druga, powszechnie znana metoda wykorzystuje aproksymację funkcji trygonometrycznych za pomocą wielomianów wysokich rzędów, co pozwala uzyskać duże dokładności otrzymywanych wyników w stosunku do metody tablicowania. Zwiększenie precyzji obliczeń odbywa się jednak kosztem wykonywania wielu podstawowych operacji arytmetycznych jak: dodawanie, odejmowanie mnożenie lub/i dzielenie, dlatego też nie jest to rozwiązanie korzystne w przypadku implementacji na platformach sprzętowych, w których operacje mnożenia/dzielenia są czasochłonne.

Ostatnim możliwym do zastosowania podejściem jest użycie algorytmu dedykowanego do obliczania funkcji hiperbolicznych i trygonometrycznych zwanego CORDIC (*ang. COordinate Rotation DIgital Computer*), który swoje działanie opiera jedynie na operacjach dodawania, odejmowania, przesunięciach bitowych oraz tablicy o niewielkich rozmiarach (najczęściej nie przekraczających kilkudziesięciu pozycji). Brak operacji mnożeń oraz dzieleń jest niewątpliwą zaletą tej metody [10]. Algorytm CORDIC opiera się na cyklicznym wyznaczaniu współrzędnych punktu znajdującego się na kole jednostkowym poprzez rotację jego aktualnej pozycji o pewien ustalony kąt [13]. Wartość tego kąta musi spełniać zależność: $\tan(\varphi) = 2^{-i}$ dla i należącego do zbioru liczb naturalnych, dlatego też jego wartość nie jest obliczana w każdym kroku iteracji, lecz tablicowana [14]. Dzięki takiemu założeniu operacje mnożenia przez tangens kąta obrotu można zastąpić przesunięciem bitowym.



Rys. 4.14: Kolejne kroki iteracji algorytmu CORDIC

Składniki $\cos(\varphi_i)$ ulegają skumulowaniu w postaci iloczynu, którego wartość jest stałą dla zadanej liczby kroków iteracji (dalej oznaczana jako C). Wartość stałej C jest zawsze dodatnia i nie zależy od znaku kąta obrotu. Dodatkowo końcowe mnożenie przez skumulowany współczynnik C można zastąpić ustawiając współrzędne początkowe punktu na [X0,Y0] = [C, 0]. Transformacja układów współrzędnych na kole jednostkowym (rys. 4.14) oraz zależności ją opisujące przedstawiają równania:

$$
\begin{aligned}
x' &= x\cos(\varphi_i) - y\sin(\varphi_i)\,, \\
y' &= y\cos(\varphi_i) + x\sin(\varphi_i)\,, \\
x' &= \cos(\varphi_i)\cdot(x - y\,\mathrm{tg}(\varphi_i))\,, \\
y' &= \cos(\varphi_i)\cdot(y + x\,\mathrm{tg}(\varphi_i)).
\end{aligned}
\tag{4.1}
$$

4.5. Bibliografia

[1] Andrew Rushton: *VHDL for logic synthesis – Third edition*; ISBN 9780470688472; 2011.
[2] Mariusz Rawski: *Język VHDL – podstawy*. [on-line 2015]: http://rawski.zpt.tele.pw.edu.pl/.
[3] Krzysztof Kołek: *Język opisu sprzętu VHDL - Materiały pomocnicze dla przedmiotu "Technika cyfrowa i mikroprocesorowa" II/III RA*. Katedra Automatyki, Akademia Górniczo-Hutnicza Al. Mickiewicza 30, 30-059 Kraków; 2012.
[4] Steve Kilts: *Advanced FPGA Design Architecture, Implementation, and Optimization*. ISBN 978-0-470-05437-6; Published by John Wiley & Sons, Inc., Hoboken, New Jersey; 2007.
[5] Douglas L. Perry: *VHDL: Programming by Example - Fourth Edition*. ISBN 978-0071400701; 2002.
[6] Mark Zwoliński: *Projektowanie układów cyfrowych z wykorzystaniem języka VHDL*. Wydawnictwa Komunikacji i Łączności; ISBN 978-83-206-1635-4; Warszawa; 2007.
[7] Wikipedia - IEEE 754 [on-line] *https://pl.wikipedia.org/wiki/IEEE_754*. Pobrane 04-2023.
[8] Z. Navabi: *Digital Design and Implementation with Field Programmable Devices*. Boston: Springer Science; ISBN: 1-4020-8011-5; 2005.
[9] S. Mirzaei, A. Hosangadi i R. Kastner: *FPGA Implementation of High Speed FIR Filters Using Add and Shift Method*. San Jose: Computer Design, 2006. ICCD 2006. International Conference on, 2006.
[10] G. Góra, G. Karpiel, P. Mars, R. Sitek i M. Goczał: *Sprzętowa implementacja transformacji Clarka i Parka*. Maszyny Elektryczne : zeszyty problemowe; tom 1; nr 113, pp. 23-28; 2017.
[11] G. Sutter i J.-P. Deschamps: *High speed fixed point dividers for FPGAs*. w Field Programmable Logic and Applications; Prague; 2009.
[12] D. Wang, N. Zheng i M. D. Ercegovac: *A radix-8 complex divider for FPGA implementation.* w Field Programmable Logic and Applications; FPL 2009. International Conference on; Prague; 2009.
[13] J. Li, J. Fang, B. Li i Y. Zhao: *Study of CORDIC Algorithm based on FPGA*. Control and Decision Conference (CCDC), 2016.
[14] S. Bhuria i P. Muralidhar: *FPGA implementation of sine and cosine value generators using Cordic Algorithm for Satellite Attitude Determination and calculators*. Power, Control and Embedded Systems (ICPCES); 2010.