

Politechnika Warszawska

W Y D Z I A Ł E L E K T R O N I K I
I T E C H N I K I N F O R M A C Y J N Y C H



Instytut Systemów Elektronicznych

Praca dyplomowa inżynierska

na kierunku Elektronika
w specjalności Elektronika i Fotonika

Parametryzowalny wieloprocesorowy system FPGA do implementacji
modułów zabezpieczeń

Szymon Łysoń

Numer albumu 318540

promotor
dr inż. Andrzej Wojeński

WARSZAWA 2025

Spis treści

1. Wstęp	8
2. Systemy zabezpieczeń	10
2.1. Warunki pracy systemów zabezpieczeń	12
2.2. Czujniki	13
2.3. Istniejące systemy zabezpieczeń	14
3. Jednostki wykonawcze	16
3.1. Procesory	17
3.1.1. Mikroarchitektury procesorów	17
3.1.2. Zestawy instrukcji procesorów	19
3.1.3. Klasy architektur równoległych	21
3.1.4. Potokowanie	22
3.2. Mikrokontrolery	23
3.2.1. Architektury interfejsów	24
3.2.2. Zastosowania mikrokontrolerów	24
3.3. Układy FPGA	27
3.3.1. Implementacje procesorów na układach FPGA	29
4. Cel i założenia pracy	30
5. Opracowanie architektury systemu zabezpieczeń	31
5.1. Procesor	32
5.2. Realizacja modułu wyboru interfejsów	33
5.2.1. Symulacje modułu wyboru interfejsów	34
5.3. Realizacja komunikacji z użytkownikiem	34
5.3.1. Moduł transmisji danych do użytkownika	35
5.3.2. Moduł wgrywania programu CPU/DPU	35

5.4. Realizacja komunikacji między CPU a DPU	39
5.4.1. Komunikacja od CPU do DPU	39
5.4.2. Komunikacja od DPU do CPU	39
5.5. Realizacja przerwań wewnętrznych. Opracowanie własne.	39
6. Opracowanie architektury procesora	41
6.1. Opracowanie modułu pobierania instrukcji	42
6.1.1. Pamięć programu	43
6.1.2. Realizacja przerwań	43
6.2. Opracowanie modułu dekodowania kodu operacyjnego	44
6.3. Opracowanie modułu dekodowania kodu operacyjnego	44
6.3.1. Działanie instrukcji	45
6.3.2. Transfer danych	45
6.3.3. Instrukcje zmiany wskaźnika programu	47
6.3.4. Instrukcje arytmetyczne i logiczne	49
6.3.5. Instrukcje manipulujące bitami	51
6.4. Opracowanie modułu wykonywania instrukcji	52
6.4.1. Opracowanie układu arytmetycznego	53
6.4.2. Opracowanie rejestrów	55
6.4.3. Realizacja pamięci danych	59
7. Symulacje procesora	61
7.1. Symulacja odczytywania pamięci programu	61
7.2. Symulacje modułu dekodowania instrukcji	61
7.3. Symulacje modułu wykonywania instrukcji	62
7.3.1. Symulacje działania rejestrów	63
7.3.2. Symulacje działania modułu ALU	64
7.3.3. Symulacje działania całego modułu	64
7.4. Symulacje całego procesora	67
7.4.1. Szczegółowa symulacja programu	67
7.5. Symulacja przerwań	70
8. Opracowanie interfejsów	73
8.1. Opracowanie wspólnych funkcjonalności interfejsów	73

8.1.1. Opracowanie pamięci interfejsów	74
8.2. Opracowanie interfejsu UART	75
8.2.1. Implementacja modułów transmisji	76
8.2.2. Implementacja modułów wysyłania	76
8.2.3. implementacja modułu odbioru	77
8.2.4. Opcje interfejsu	77
8.2.5. Symulacje interfejsu UART	78
8.3. Opracowanie interfejsu SPI	79
8.3.1. Opcje interfejsu	80
8.3.2. Implementacja modułu transmisji danych	81
8.3.3. Symulacje interfejsu	81
8.4. Opracowanie interfejsu I^2C	82
8.4.1. Opcje interfejsu	83
8.4.2. Implementacja modułu transmisji danych	84
8.4.3. Symulacje interfejsu I^2C	84
8.5. Opracowanie interfejsu PWM	85
8.5.1. Symulacje interfejsu PWM	86
9. Realizacja symulacji i testów	87
9.1. Biblioteka mikrokontrolera	87
9.2. Zalecane sposoby wykorzystania procesora przez programistę	89
9.2.1. Zalecenia dotyczące DPU	89
9.2.2. Zalecenia dotyczące CPU	90
9.2.3. Realizacja przesyłania oprogramowania do procesora	91
9.3. Realizacja projektu w języku HDL dla FPGA	91
9.3.1. Realizacja pliku głównego projektu	94
9.3.2. Realizacje testów	99
9.4. Zużycie zasobów układu FPGA	101
9.5. Częstotliwość zegara	101
10 Realizacja testów sprzętowych	104
10.1.Cel testu	104
10.2.Programy użyte podczas testu	105

10.2.1. Program CPU użyty podczas testu	105
10.2.2. Program DPU użyty podczas testu	107
10.3. Przeprowadzenie testu	109
10.3.1. Wgrywanie oprogramowania	109
10.3.2. Obsługa czujników	110
10.3.3. Weryfikacja działania sygnału PWM	111
10.3.4. Weryfikacja odczytu temperatury	111
11 Podsumowanie	114

Streszczenie Tematem pracy jest parametryzowalny wieloprocesorowy system FPGA służący do implementacji systemów zabezpieczeń. Głównym celem było opracowanie wieloprocesorowego systemu zdolnego do jednoczesnego odczytu danych z wielu różnych czujników używających protokoły komunikacyjne oraz do przetwarzania tych danych równolegle. System jest parametryzowalny, co pozwala użytkownikowi na definiowanie liczby i rodzaju interfejsów komunikacyjnych oraz obsługujących je procesorów. Układ jest w pełni programowalny, co pozwala na implementację algorytmów zabezpieczeń. Równoległe przetwarzanie danych jest istotne w zastosowaniach o potencjalnie złożonych algorytmach przetwarzania danych, tj. systemy kontrolno-pomiarowe.

Pracę zrealizowano wykorzystując język opisu sprzętu VHDL i zweryfikowano na prototypowej płytce z układem FPGA. Zaimplementowano procesor zgodny z architekturą Atmel AVR, na podstawie którego opracowano i zaimplementowano wielordzeniowy mikrokontroler. Jako przykładowe interfejsy zaimplementowano UART, I2C i SPI. Opracowano również moduły HDL służące do komunikacji z użytkownikiem oraz do programowania układu. W celu zaprogramowania układu napisano biblioteki programistyczne w języku C.

Projekt został przetestowany na platformie deweloperskiej Zybo Z7 z układami FPGA firmy Xilinx. Wyniki testów potwierdziły skuteczność rozwiązania w zakresie jednoczesnego przetwarzania danych z wielu źródeł. Zrealizowany system umożliwia implementację moduarnego mikrokontrolera z interfejsem programistycznym w języku C, obsługującego różne konfiguracje sprzętowe. Rozwiązanie ma zastosowanie w systemach pracujących z różnymi czujnikami, jakimi są m.in. systemy zabezpieczeń.

Słowa kluczowe: AVR, FPGA, HDL, Interfejsy, Mikrokontrolery, Procesory wielordzeniowe, Systemy zabezpieczeń.

Abstract The subject of this work is a parameterizable multiprocessor FPGA system for implementing protection systems. The main goal was to develop a multiprocessor system capable of simultaneously reading data from many different sensors using communication protocols and processing this data in parallel. The system is parameterizable, allowing users to define the number and type of communication interfaces and the processors that support them. The system is fully programmable, allowing the implementation of security algorithms. Parallel processing is important in applications with potentially complex algorithms, such as control and measurement systems.

The work was done using the VHDL hardware description language and verified on a prototype board with an FPGA chip. A processor conforming to the Atmel AVR architecture was designed and implemented, based on which a multi-core microcontroller was developed and implemented. UART, I2C, and SPI were implemented as exemplary interfaces. HDL modules for communication and programming were also developed. C programming libraries were written to program the chip.

The designed system was tested on the Zybo Z7 development platform with FPGAs from Xilinx. Test results confirmed the solution's effectiveness for simultaneous data processing from multiple sources. The realized system enables the implementation of a modular processor with a programming interface in C language, supporting various hardware configurations. The solution applies to systems working with different sensors, such as protection systems.

Keywords: Architecture, AVR, FPGA, HDL, Communication interfaces, Microcontrollers, Multicore processors, Protection systems.

Rozdział 1: Wstęp

W większości projektów systemów elektronicznych niezbędne jest zastosowanie jednostki sterującej. Najczęściej w tym celu wykorzystywane są mikrokontrolery o różnym stopniu złożoności. Częstym czynnikiem o zastosowaniu konkretnego modelu czy też serii układów są kryteria ekonomiczne, stanowiące kluczowy element w produkcji wielkoseryjnej. W przypadku złożonych systemów elektronicznych pracujących w trudnym środowisku niezbędne jest zapewnienie dodatkowego układu zabezpieczającego. Często wykorzystywane są w tym celu podobne układy jak w przypadku systemów sterujących.

Problem jest jednak bardziej złożony, gdy dotyczy systemów o dużym stopniu wyspecjalizowania, instalowanych w wyjątkowo trudnym środowisku pracy takich jak, wysoka temperatura, promieniowanie jonizacyjne, gdzie wszystkie elementy wspomagające jak i otaczające trzeba brać pod uwagę w trakcie pracy systemu.

Do takich zastosowań można zaliczyć systemy kontrolno-pomiarowe pracujące na reaktorach termojądrowych, które w sposób specjalny muszą być zabezpieczone. Przykładem tego typu systemów są systemy diagnostyki promieniowania miękkiego gorącej plazmy tokamakowej, instalowane bezpośrednio lub w otoczeniu reaktorów termojądrowych. Można wyróżnić systemy znajdujące się na tokamakach MAST (Wielka Brytania) i WEST (Francja)[1][2]. Na bazie analizy publikacji naukowych zwrócono uwagę na potrzebę opracowania systemu zabezpieczeń, którego głównym elementem byłby rekonfigurowalny, adaptacyjny mikrokontroler implementowany w układach FPGA. W tym przypadku istnieje często konieczność analizy danych z wielu czynników równocześnie. W związku z tymi wymogami, typowe mikrokontrolery mogą nie spełniać wszystkich założeń.

Tematem i celem pracy jest projekt i implementacja możliwie uniwersalnego parametryzowalnego wieloprocesorowego systemu FPGA do implementacji modułów zabezpieczeń, umożliwiającego jednoczesny odczyt danych z wielu różnych czujników wykorzystujących różne protokoły komunikacji. Musi on być programowalny w sposób, aby programista mógł zdefiniować, ile jakich interfejsów chce użyć oraz w jaki sposób chce z nich korzystać, mając przy tym możliwie jak największą swobodę w tworzeniu oprogramowania. Odczytane przez interfejsy informacje muszą być przetwarzane równolegle tak, aby możliwe było przetwarzanie wielu różnych danych jednocześnie oraz sprawdzenie, czy zaszyły zdefiniowane przez programistę warunki. Dzięki temu możliwa będzie szybka odpowiedź systemu. Jednocześnie system powinien zapewniać możliwość komunikacji z użytkownikiem w celu analizy pracy systemu.

W wielu krytycznych zastosowaniach wymagane jest stale natychmiastowe odbieranie i analizowanie danych z czujników. Jednakże mikrokontrolery, które są powszechnie stosowane do obsługi czujników, nie mają takiej możliwości. Mogą one odbierać dane z maksymalnie kilku źródeł naraz, a każda ich analiza wiąże się z zarezerwowaniem na wyłączność procesora, który może przerwać działanie tylko wtedy, gdy pojawi się sygnał zdefiniowany jako bardziej priorytetowy. Często niezbędną jest szybka reakcja na wykryte zjawisko, np. wyłączenie urządzenia

ze względu na zbyt wysoką temperaturę.

Systemy tego rodzaju pracują z reguły ze standardowymi protokołami komunikacyjnymi, dzięki czemu możliwe jest opracowanie modułarnego rozwiązania. Jednocześnie, w celu przyspieszenia pracy użytkownika, powinna być możliwość łatwego oprogramowywania realizowanych czynności mikrokontrolera w językach wysokopoziomowych, jak np. C. Tym samym, projekt może być wykorzystany przez osoby, które nie posiadają specjalistycznej wiedzy z zakresu programowania układów FPGA.

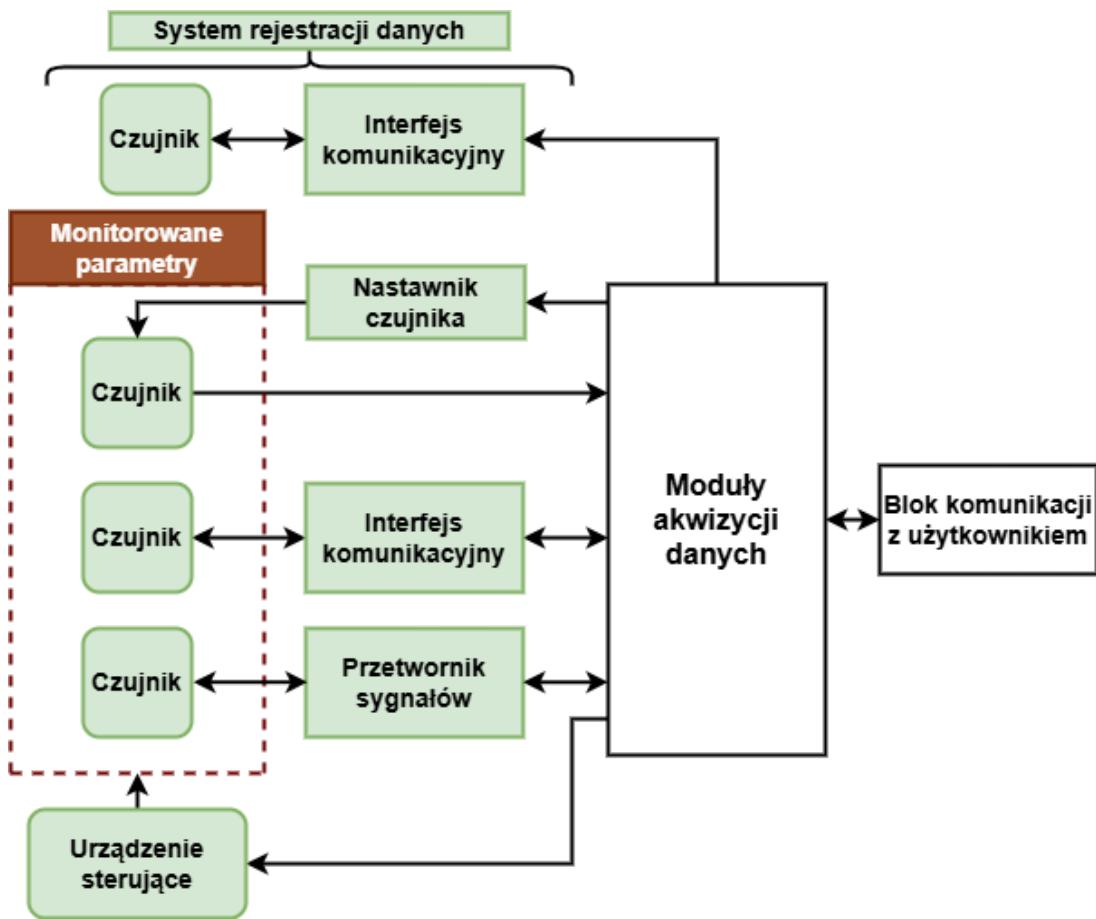
Rozdział 2: Systemy zabezpieczeń

Systemy zabezpieczeń to zespoły urządzeń, narzędzi oraz oprogramowania, które są stosowane do monitorowania, kontrolowania i analizowania parametrów procesów technologicznych lub środowiska. Ich celem jest zapewnienie optymalnej pracy systemów, identyfikowanie nieprawidłowości oraz utrzymanie jakości i bezpieczeństwa procesów. Znajdują zastosowanie w wielu dziedzinach, takich jak przemysł, medycyna, transport czy nauka. Pozwalają na automatyzację procesów technologicznych, ich optymalizację oraz zwiększenie bezpieczeństwa i wydajności[3]. Możemy wymienić 3 główne zadania systemu zabezpieczeń[4]:

- Rejestracja danych- Zmierzone przez czujnik informacje są odczytywane i gromadzone.
- Nadzór- System nadzoruje lub monitoruje stan systemu przy użyciu uzyskanych danych. Uzyskane dane są w tym celu przetwarzane.
- Sterowanie czujnikami- Polega ono na wysyłaniu poleceń do urządzeń w celu ich obsługi, inicjalizacji czujników oraz kontroli przesyłu danych.

Te zadania są najczęściej realizowane za pomocą programowalnych mikrokontrolerów połączonych z czujnikami. Na rysunku 1 przedstawiono podstawowe elementy systemu zabezpieczeń. Zaznaczono również lokalizację względem zabezpieczanego urządzenia. Przedstawiono 1 czujnik zewnętrzny dla urządzenia, służący do pomiaru otoczenia. Są to:

- Czujniki: Kluczowy komponent systemu, który wykrywa bodźce takie jak temperatura, ciśnienie, wilgotność, przemieszczenie czy stężenie gazów. W wielu przypadkach potrzebują one dodatkowego układu nastawczego, który kontrolowałby jego działanie.
- Przetworniki sygnałów: Konwertują dane z czujników na sygnały elektryczne odpowiednie do dalszego przetwarzania. Mogą obejmować wzmacniacze, przetworniki analogowo-cyfrowe (ADC), filtry i moduły kondycjonujące sygnał.
- Moduły akwizycji danych: Gromadzą i przetwarzają sygnały z różnych czujników. Często obejmują procesory lub mikrokontrolery, które analizują dane w czasie rzeczywistym lub je przechowują. Jest też odpowiedzialny za sterowanie czujnikami.
- Interfejsy komunikacyjne: Zapewniają transmisję danych między czujnikami, modułami akwizycji i użytkownikiem końcowym. Mogą to być przewodowe systemy, takie jak UART, SPI, I²C, lub bezprzewodowe standardy, np. Wi-Fi czy Bluetooth. We współczesnych urządzeniach zwykle są one połączone z czujnikiem w jeden układ scalony.
- Urządzenia sterujące: Zamieniają sygnały sterujące na działanie fizyczne, takie jak ruch, siła, zmiana położenia, czy kontrola przepływu. Dzięki nim możliwy jest wpływ systemu zabezpieczeń na zabezpieczone urządzenie.



Rysunek 1. Pozycja sensorów w systemie zabezpieczeń[5]

- Urządzenia komunikacji: Umożliwia komunikację systemu zabezpieczeń z użytkownikiem oraz innymi systemami elektronicznymi, dzięki czemu możliwe jest przekazanie wyników pracy systemu.

Systemy pomiarowe powinny ponadto spełniać szereg cech, takich jak:

- Wielowymiarowość pomiarów: Systemy te mogą integrować różnorodne czujniki, umożliwiając jednoczesny pomiar wielu parametrów (np. temperatury, wilgotności i ciśnienia w jednym urządzeniu).
- Automatyczna kalibracja i kompensacja błędów: Wiele systemów pomiarowych zawiera funkcje automatycznej kalibracji, które eliminują odchylenia wyników spowodowane wpływem środowiska, np. zmianami temperatury.
- Możliwość implementacji algorytmów: Oprogramowanie pomiarowe umożliwia analizę, wizualizację i interpretację danych pomiarowych. W zaawansowanych systemach może obejmować nawet algorytmy uczenia maszynowego do automatycznego wykrywania wzorców w danych. Wykonywanie algorytmów nie powinno uniemożliwić odczytu nowych danych.
- Dynamika i precyzja: Systemy te są w stanie dokładnie mierzyć zmieniające się w czasie wielkości, dostosowując się do różnych zakresów pracy.

2.1 Warunki pracy systemów zabezpieczeń

W celu określenia czynników niekorzystnych dla pracy systemu elektronicznego, a tym samym zagrożeń i sposobu ich detekcji, niezbędna jest krótka charakterystyka ich parametrów. Złożone systemy elektroniczne często wymagają ciągłego monitorowania parametrów zabezpieczanych systemów[6]. Do najczęściej monitorowanych należą:

- napięcie zasilania
- temperatura komponentów
- pobór mocy

Często systemy składają się z większej liczby modułów dodatkowych, które umożliwiają ich pracę. Jako przykład można podać:

- systemy zasilania gazowego
- systemy zasilania wysokiego napięcia
- systemy chłodzenia wodnego
- systemy pozycjonowania
- systemy serwerowni

Kolejnym elementem czynników wymagających obserwacji są elementy otoczenia, szczególnie przy instalacjach w trudnych warunkach środowiskowych. Przykładami parametrów które można monitorować są[7]:

- poziom promieniowania jonizującego
- poziom tlenu w otoczeniu
- poziom szkodliwych gazów w otoczeniu
- wilgotność
- ciśnienie
- temperatura otoczenia

Wszystkie te elementy wymagają często wymagają ciągłego monitorowania i reakcji w zależności od rodzaju zdarzenia. Tym samym konieczne jest implementowanie scenariuszy reakcji o różnym poziomie złożoności w systemach mikrokomputerowych.

W tabeli 1 omówiono wybrane kategorie czynników monitorowanych przez system zabezpieczeń wraz z czujnikiem mogącym zostać wykorzystanym do pomiaru[3][8]. Nie opisano wszystkich możliwych zagrożeń, a jedynie ich przykłady. Wiele z pomiarów wymaga złożonych obliczeń, które mógłby zablokować pracę jednordzeniowego mikrokontrolera uniemożliwiając wykonywanie pozostałych czynności, jak np. odbiór i analiza nowych danych

Tabela 1. Własności czynników monitorowanych przez systemy zabezpieczeń

Monitorowana właściwość	Przykładowe zagrożenie	Reakcja systemu	Sposób analizy	Używany czujnik
Napięcie zasilania	Przeciążenie	Wyłączenie	Porównanie	Przetwornik ADC
Napięcie zasilania	Poziom zakłóceń	Włączenie filtrów	Analiza szumowa	Przetwornik ADC
Temperatura urządzenia	Przegranie	Wyłączenie urządzenia	Porównanie	Termometr
Przepływ gazu	Zbyt niski przepływ	Zwiększenie mocy pompy	Równanie Bernoulliego	Czujnik przepływu
Hałas	Uszkodzenie mechaniczne	Wyłączenie urządzenia	Analiza Fouriera	Mikrofon
Poziom cieczy	Przelanie zbiornika	Odciecie przepływu	porównanie	Czujnik poziomu cieczy
Przyspieszenie	Zderzenie	Aktywacja poduszki powietrznej	Porównanie	Akcelerometr
Wykrycie ruchu	Włamanie	Aktywacja alarmu	Analiza wartości	Czujnik ruchu
Błędne przemieszczenie	Błędny ruch maszyny	Zatrzymanie ruchu	Analiza wektorowa	Czujnik MEMS
Poziom promieniowania	Radiacja urządzenia	Redukcja promieniowania	Analiza numeryczna	Czujnik promieniowania
Analiza chemiczna	Zatrucie	Ewakuacja	Interpolacja widma	Spektrometr

2.2 Czujniki

Czujniki to urządzenia służące do wykrywania i rejestrowania zmian różnych wielkości fizycznych, chemicznych lub biologicznych, a następnie konwersji tych zmian na sygnały elektryczne zrozumiałe dla systemów elektronicznych. Są one istotnym elementem nowoczesnych systemów akwizycji danych, które przekształcają informacje z otoczenia w dane możliwe do analizy i przetwarzania[5]. Ze względu na wykrywane zjawiska można je podzielić na kilka rodzajów[8]:

- Czujniki mechaniczne:
 - Wykrywają: siłę, ciśnienie, naprężenia, przemieszczenia, prędkość, przyspieszenie.
 - Przykłady: tensometry, akcelerometry, żyroskopy, czujniki ciśnienia.
- Czujniki elektryczne:
 - Wykrywają: zmiany rezystancji, pojemności, indukcyjności.
 - Przykłady: czujniki pojemnościowe, indukcyjne, rezystancyjne.
- Czujniki magnetyczne:
 - Wykrywają: pole magnetyczne, natężenie pola, kierunek.
 - Przykłady: czujniki Halla, magnetometry, czujniki magnetorezystancyjne.
- Czujniki optyczne:

- Wykrywają: światło, intensywność, długość fali, zmiany w transmisji lub odbiciu światła.
- Przykłady: fotodiody, fototranzystory, czujniki światłowodowe.
- Czujniki termiczne:
 - Wykrywają: temperaturę, zmiany ciepła.
 - Przykłady: termopary, termistory, pirometry.
- Czujniki akustyczne:
 - Wykrywają: fale dźwiękowe, wibracje.
 - Przykłady: mikrofony, hydrofony, czujniki ultradźwiękowe.
- Czujniki chemiczne:
 - Wykrywają: stężenie gazów, jonów, substancji chemicznych.
 - Przykłady: czujniki pH, elektrochemiczne, katalityczne.
- Czujniki biologiczne:
 - Wykrywają: biomolekuły, enzymy, antygeny.
 - Przykłady: glukometry, biosensory oparte na DNA.
- Czujniki radiacyjne:
 - Wykrywają: promieniowanie jonizujące, UV, podczerwień.
 - Przykłady: detektory Geigera-Müllera, czujniki podczerwieni, fotometry UV.

Pod względem konstrukcji czujniki można podzielić na bezpośrednie i złożone. Bezpośrednie, takie jak fotodioda czy termopara, wykorzystują efekty fizyczne do natychmiastowej konwersji bodźca w sygnał elektryczny. Czujniki złożone natomiast korzystają z transduktorów, które przekształcają energię jednego rodzaju w inną, zanim nastąpi przetwarzanie na sygnał elektryczny. Przykładem takiego systemu może być czujnik chemiczny, gdzie energia reakcji chemicznej przekształcana jest w ciepło, a następnie na sygnał elektryczny przez termoparę[5].

Dzięki swojej różnorodności i wszechstronności, czujniki odgrywają kluczową rolę w przemysłowych, naukowych i codziennych zastosowaniach, umożliwiając precyzyjną interakcję między światem fizycznym a cyfrowym [5].

2.3 Istniejące systemy zabezpieczeń

W większości przypadków jako system zabezpieczeń są wykorzystywane mikrokontrolery. To rozwiązanie nie pozwala na równoległość odczytywania i przetrwania danych. Rozwiązania, które to umożliwiają, to między innymi:

- Mikrokontrolery Propeller 1 [9] oraz Propeller 2[10] firmy Parallax. Są to 32-bitowe, 8-rdzeniowe procesory, każdy z 512B pamięci RAM, ograniczonym dostępem do głównej pamięci, oraz jednym znacznie większym centralnym modułem sterującym. Ich głównym problemem jest brak wbudowanych interfejsów komunikacji. Obsługują wyłącznie PWM, ADC i DAC. Natomiast inne interfejsy musi zaimplementować programista, co znacząco utrudnia korzystanie z urządzenia.

- Mikrokontrolery wyposażone w wielokanałowe układy DMA nie potrafią odczytywać lub przesyłać danych równolegle, a jedynie robić to bez przerwania pracy procesora. W większości przypadków jako system zabezpieczeń są wykorzystywane
- GA144 firmy GreenArrays [11] posiada aż 144 rdzenie, a XCORE-200 SERIES do 32, jednak nie mają one wbudowanych interfejsów komunikacji oraz mają niewielką liczbę pinów. Są one stworzone z myślą o równoległych obliczeniach wymaganych do szybkiego wykonywania niektórych algorytmów.
- W 2005 roku został złożony patent[12] na mikrokontroler o wielu identycznych rdzeniach, każdy z nich zdolny do równoległej analizy danych. Jednak nie ma on wbudowanych interfejsów, a jedynie umie je emulować sprzętowo oraz nie znalazłem informacji o realizacji sprzętowej patentu, a on sam jest nieaktywny. Jest to głównie zbiór wymagań potencjalnego projektu, którego cel był podobny do zagadnienia tej pracy.
- Wspomniane wcześniej układy FPGA potrafią przetwarzać dane w wymagany przez temat problemu, to wymagają nieporównywalnie większego nakładu pracy w celu wykorzystania ich do analizy danych z czujników niż napisanie kodu na mikrokontroler. Powszechnie stosuje się do rozwiązywania problemu pracy inżynierskiej, a moim zadaniem jest jego uproszczenie.

Rozdział 3: Jednostki wykonawcze

W ciągu ostatnich 70 lat technologia cyfrowa dokonała znaczących postępów. Dzisiaj za kilkanaście złotych można kupić przenośny komputer, który ma znacznie lepszą wydajność oraz więcej pamięci operacyjnej i dyskowej niż komputer sprzed 40 lat kosztujący miliony. Przykładowo Iphone 12 jest 5000 razy szybszy niż CRAY-2- superkomputer z lat 80.[13] Taki szybki rozwój wynika zarówno z udoskonaleń w technologii budowy komputerów, jak i z innowacyjnych rozwiązań w ich architekturze.

Choć rozwój technologii był dość regularny, postęp związany z nowymi architekturami komputerowymi był mniej przewidywalny. W pierwszych 25 latach istnienia komputerów elektronicznych oba te aspekty przyczyniły się do rocznej poprawy wydajności na poziomie około 25%. W końcu lat 70. wprowadzono mikroprocesor, który korzystając z nowoczesnych technologii układów scalonych, przyspieszył rozwój wydajności do około 35% rocznie[14].

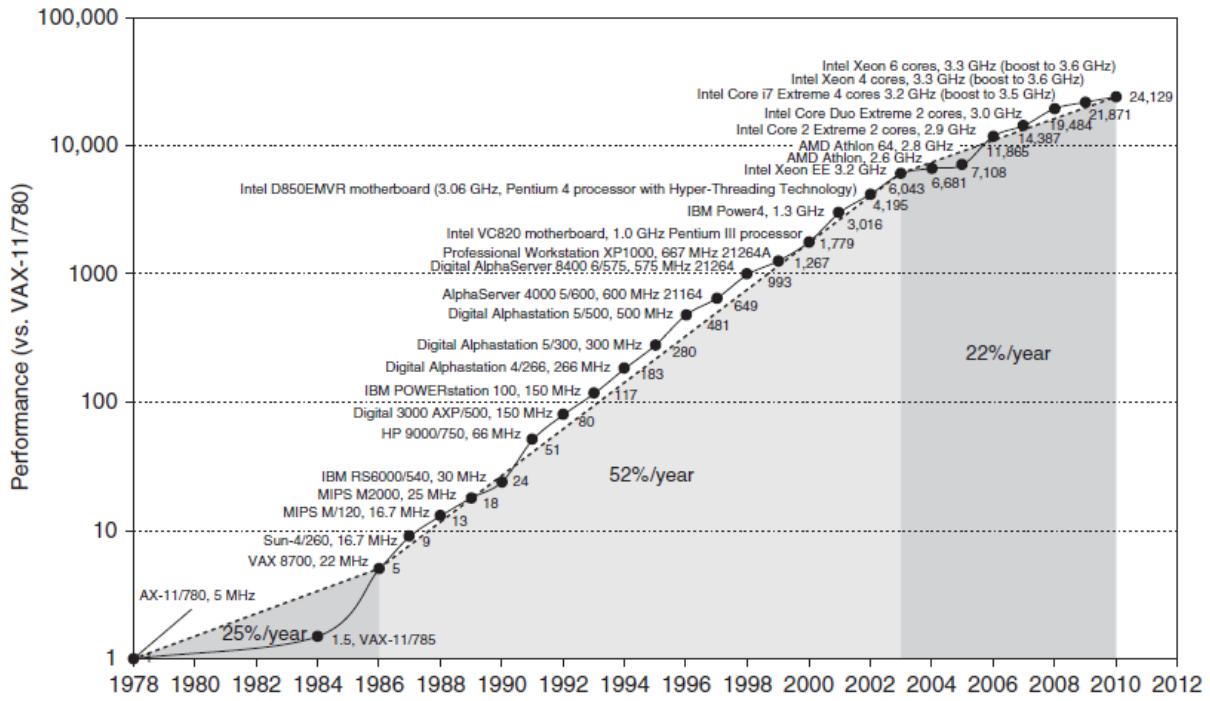
To przyspieszenie, w połączeniu z masową produkcją mikroprocesorów, spowodowało, że coraz częściej zaczęto polegać na tych komponentach. Pojawianie się wygodnego w użytkowaniu języka C w latach 70-tych, jak również powstanie nowoczesnych i darmowych kompilatorów, przyczyniły się do wyeliminowania języka ASM, który jest znacznie bardziej czasochłonny i skomplikowany w użyciu[15]. Dodatkowo powstanie standaryzowanych systemów operacyjnych, takich jak UNIX i Linux, który jest jego otwarto-źródłowym klonem, obniżyło koszty związane z implementacją nowych architektur.

Te zmiany umożliwiły na początku lat 80. rozwój architektury instrukcji RISC (ang. Reduced Instruction Set Computer)[16]. Komputery oparte na architekturze RISC skupiły się na dwóch kluczowych technikach wydajności: równoległym przetwarzaniu instrukcji oraz efektywnym wykorzystaniu pamięci podręcznej. Ta architektura podniosła standardy wydajności, zmuszając starsze do dostosowania się lub wycofania się z rynku.

Wykres 2 ilustruje, jak ulepszenia architektoniczne i organizacyjne doprowadziły do 17-letniego wzrostu wydajności przekraczającego 50% rocznie. W konsekwencji dostępność i możliwości komputerów znacznie się zwiększyły. W wielu dziedzinach dzisiejsze mikroprocesory o najwyższej wydajności przewyższają superkomputery sprzed zaledwie dziesięciu lat, a dodatkowo stały się na tyle tanie, że są dostępne dla każdego. Powstały też nowe zastosowania dla procesorów. Dzisiaj są one powszechnie wykorzystywane w telefonach osobistych czy tabletach, które coraz częściej zastępują komputery osobiste. Wzrasta też zapotrzebowanie na serwery internetowe wykorzystujące wielkie komputery składające się z setek procesorów. Dodatkowo nastąpiło znaczne zmniejszenie rozmiarów tranzystorów zgodnie z prawem Moore'a. Umożliwia to zmniejszenie rozmiarów oraz zużycia energii każdego elementu składowego komputera.

Jednak od 2003 roku na skutek różnych ograniczeń technologicznych wzrost wydajności procesorów uległ zahamowaniu. Problemem jest odprowadzanie ciepła, dalsze skalowanie tranzystorów oraz opóźnienia przesyłania sygnałów. Trudne staje się zwiększenie równolegle wykonywanych funkcjonalności przez daną instrukcję lub zmniejszenie wydzielanej przez procesor

mocy. Współcześnie skupia się przede wszystkim na rozwijaniu wielordzeniowych procesorów.



Rysunek 2. Wzrost wydajności procesorów[14]

3.1 Procesory

Procesor to układ cyfrowy, który wykonuje instrukcje na zewnętrznym źródle danych takim jak pamięć. Najczęściej przybiera on formę mikroprocesora, który może być zaimplementowany na jednym układzie scalonym. Często są używane jako centralna jednostka obliczeniowa (CPU), ale mają także inne zastosowania, od przetwarzania grafiki, do obsługi zewnętrznych urządzeń.

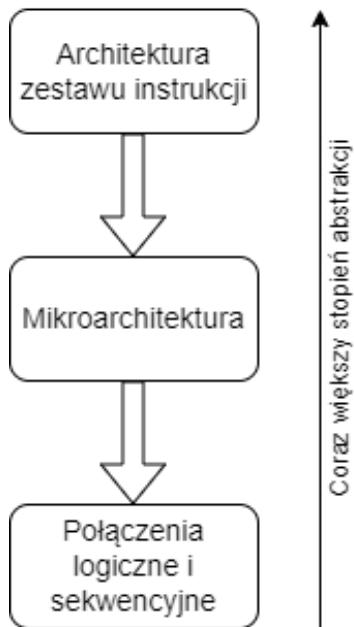
Architektura instrukcji określa zestaw wykonywalnych przez procesor instrukcji. To od architektury instrukcji zależą możliwe zastosowania procesora oraz jego wydajność wykonywania różnych operacji. Tylko ona ma znaczenie dla kompilatora, a co za tym idzie, dla programisty.

Mikroarchitektura to implementacja architektury instrukcji i musi spełniać wymagania niezbędne do jej działania, takie jak zapewnienie odpowiedniego zasobu rejestrów. Jest ona reprezentowana jako zbiór połączeń pomiędzy różnymi elementami procesora, ale nie określa implementacji obwodów logicznych. Dopiero te bloki są implementowane przez obwody logiczne. Przedstawiony go na rysunku 3 podział na stopnie abstrakcji jest niezbędny dla zaprojektowania procesora i należy je rozważyć przed implementacją procesora.

3.1.1 Mikroarchitektury procesorów

Procesor można podzielić na wiele drobniejszych wzajemnie połączonych elementów. Należą do nich między innymi:

- ALU (ang. Arithmetic logic unit) wykonuje operacje arytmetyczne i logiczne na liczbach całkowitych.



Rysunek 3. Podział procesora na poziomy abstrakcji. Opracowanie własne.

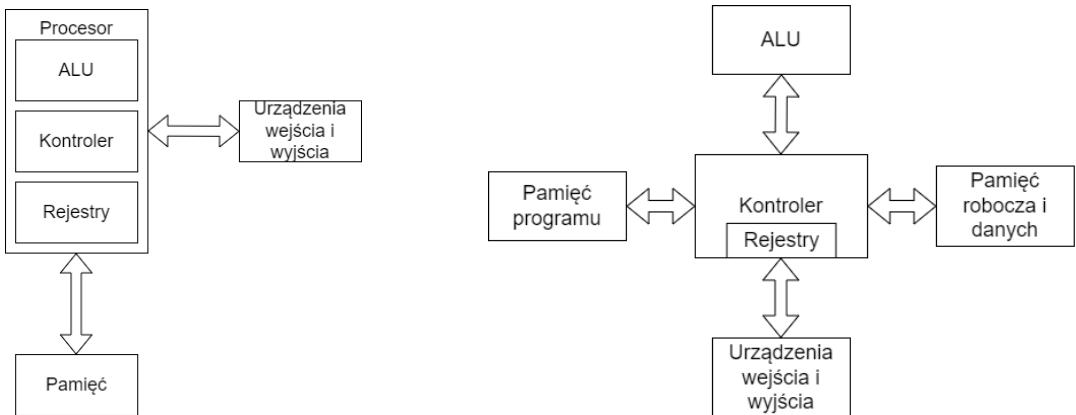
- Dekoder instrukcji zajmuje się odczytaniem instrukcji oraz wygenerowaniem na ich podstawie szeregu sygnałów sterujących.
- Pamięć pozwala na zapisywanie danych i ich odczytywanie.
- Rejestry to podręczne dostępną pamięć wykorzystywana do najczęściej używanych zadań, takich jak zapisywanie linijki aktualnie przetwarzanej instrukcji (PC- ang. Program Counter) czy zapisywanie wyniku ostatniej operacji arytmetycznej.
- Kontroler (ang. control unit) kieruje działaniem procesora. Instrukcje są dekodowane na sygnały sterujące, które kierują działaniem innych jednostek.

Od połączenia ze sobą tych bloków zależy działanie procesora. Możemy wyróżnić podział na architekturę Architektura von Neumanna i Architekturę Harwadzką.

Architektura von Neumanna odnosi się do każdego komputera z zapisanym programem, w którym pobieranie instrukcji i operacja na danych nie mogą wystąpić w tym samym czasie ze względu na wspólną magistralę. Jest to określone jako wąskie gardło von Neumanna, które często ogranicza wydajność tych procesorów.

Architektura Harwadzka zawiera oddzielne połączenia sygnałów dla instrukcji i danych. Zdecydowana większość współczesnych komputerów wykorzystuje ten sam mechanizm sprzętowy do kodowania i przechowywania danych i instrukcji programu. Procesor właśnie z tą architekturą jest przedstawiony w tej pracy.

Kod operacyjny (ang. OpCode, skrót od Operational Code) to podstawowy składnik instrukcji komputerowych. Reprezentuje on konkretną operację lub instrukcję, którą centralna jednostka przetwarzania (CPU) może wykonać. Określa rodzaj operacji do wykonania, taki jak operacje



Rysunek 4. Architektura von Neumanna. Opracowanie Rysunek 5. Architektura Harwadzka. Opracowanie własne.

arytmetyczne, logiczne lub transferu danych. Te kody są dekodowane przez CPU w celu wykonania odpowiadającej operacji, ułatwiając manipulację danymi i kontrolę przepływu w programie komputerowym.

Stos to specjalna pamięć, która przechowuje adresy powrotu, zmienne lokalne i inne dane tymczasowe używane przez funkcje i podprogramy. Stos jest strukturą danych o charakterze LIFO (Last In, First Out), co oznacza, że ostatnio dodane dane są pierwsze do usunięcia. W mikroprocesorze stos jest zwykle używany do przechowywania adresów powrotu z podprogramów, wartości rejestrów oraz innych danych tymczasowych podczas wykonywania programu. Jest to ważny element architektury mikroprocesora, który umożliwia efektywne zarządzanie pamięcią i kontrolę przepływu danych w trakcie wykonywania programu.

3.1.2 Zestawy instrukcji procesorów

Jest wiele różnych zestawów instrukcji, które są bardziej lub mniej wyspecjalizowane do konkretnych zadań. Różnią się one ich ilością, długością, złożonością i funkcjonalnością. W pierwszej kolejności należy rozróżnić dwa najpopularniejsze rodzaje architektury: Architektury RISC (Reduced Instruction Set Computer) i CISC (Complex Instruction Set Computer)[16]. Różnią się one podejściem do projektowania procesorów. RISC opiera się na prostych, jednorodnych instrukcjach, które wykonują się w jednym cyklu zegara, co sprzyja efektywności energetycznej, prostocie sprzętu i łatwości implementacji potokowania. Z kolei CISC wykorzystuje złożone instrukcje, które mogą wykonywać wiele operacji w ramach jednej komendy, co zmniejsza rozmiar kodu, ale wymaga bardziej skomplikowanego sprzętu i większego zużycia energii.

Architektury typu RISC obecnie dominują w rynku mobilnych systemów[16]. Zaczęły być one powszechnie używane w latach 90. Poniżej znajduje się kilka najpowszechniejszych architektur tego typu. Inne architektury dzisiaj nie cieszą się popularnością i zostały wyparte przez rynek.

- ARM

dominuje rynek urządzeń energooszczędnnych. Powszechnie używane przez smartfony, tablety, smartwatche, konsole przenośne, Raspberry Pi itd. Jednak dzięki stałemu szybkościu rozwojowi tej architektury, urządzenia bazujące na ARM zaczynają dośćcigać procesory CISC, zachowując przy tym znacznie mniejsze zużycie energii. Przykładem jest najnowszy MacBook ze swoim słynnym procesorem M1. [17]

- MIPS,
mimo że dzisiaj nie jest już powszechnie używany, wciąż jest istotny ze względów historycznych. Wywarł on spory wpływ na inne architektury, a dzięki swojej prostocie jest nadal często wykładowaną architekturą, która łatwo wprowadza do wielu zaawansowanych pojęć. Dzisiaj jego komercyjne zastosowania ograniczają się do urządzeń wbudowanych[18].

- Atmel AVR
to 8-bitowa architektura, która współcześnie jest stosowana głównie przez Arduino, czyli hobbystyczną platformę do tworzenia urządzeń elektronicznych. Jego instrukcje są mniejsze od innych współcześnie produkowanych urządzeń, ale przez to ma mniejsze możliwości. W przeciwieństwie do pozostałych zestawów instrukcji może operować maksymalnie na 16-bitowych liczbach, które są za małe dla większości współczesnych zastosowań. Brakuje mu też wsparcia dla liczb zmiennoprzecinkowych[19].

- RISC-V
to otwartoźródłowa architektura procesora o modularnym i skalowalnym projekcie, zaprojektowana z myślą o elastyczności i szerokim zastosowaniu. Podstawowy zestaw instrukcji jest prosty, a rozszerzenia takie jak obsługa mnożenia, operacji atomowych, czy wektorowych umożliwiają dostosowanie architektury do specjalistycznych potrzeb. RISC-V wykorzystuje stałą długość instrukcji (32 bity), co upraszcza dekodowanie i implementację potokowania, a opcjonalne instrukcje skompresowane zmniejszają rozmiar kodu. Dzięki otwartości licencyjnej, RISC-V wspiera rozwój niestandardowych układów, od prostych mikrokontrolerów po superskalowe procesory[20].

Typ architektur CISC ze względu na brak jednoznacznej definicji, zawierają wszystkie zestawy instrukcji, w których instrukcje różnią się od siebie długością. Do lat 90. dominowały one rynek. Współcześnie stosowane architektury tego rodzaju to:

- x86 opracowana przez firmę Intel, która zadebiutowała w 1978 roku z procesorem 8086 i od tego czasu przeszła wiele ewolucji. Charakteryzuje się złożonym zestawem instrukcji (CISC), który umożliwia wykonywanie skomplikowanych operacji w jednym cyklu zegara, a także różnorodnymi typami rejestrów, w tym rejestrami ogólnego przeznaczenia i segmentowymi. Architektura ta obsługuje model pamięci segmentowej oraz różne tryby pracy, w tym tryb rzeczywisty i tryb chroniony, co pozwala na lepsze zarządzanie pamięcią i wielozadaniowością. Dzięki dużej kompatybilności wstępnej nowsze procesory x86 mogą uruchamiać starsze oprogramowanie, co czyni je popularnymi w komputerach osobistych, serwerach i stacjach roboczych, a ich wsparcie dla wielowątkowości i procesorów wielordzeniowych sprawia, że są one odpowiednie do współczesnych zastosowań obliczeniowych[21].
- MCS-51
popularnie nazywanym 8051, to produkowana przez Intela architektura procesorów mająca swoje zastosowanie w systemach wbudowanych. Oryginalne procesory Intela cieszyły się dużą popularnością w latach 80. i na początku lat 90. a ich udoskonalone, binarne kompatybilne wersje są nadal w użyciu. Mają złożony zestaw instrukcji oraz oddzielne przestrzenie pamięci dla programów i danych. Poza fizycznymi urządzeniami, kilka firm oferuje również pochodne MCS-51 jako rdzenie IP do wykorzystania w układach FPGA [22].

Istnieją jeszcze architektury rodzaju VLIW (ang. Very long instruction word). Zawierają one jeszcze bardziej skomplikowane instrukcje.[23] Architektury rodzaju VLIW nigdy nie zdobyły znaczącej popularności, a ich jedną współczesną realizacją jest rosyjski mikroprocesor Elbrus 2000, jednak jest on wykorzystywany tylko w Rosji i jest tutaj wspominany w ramach omówienia teoretycznych architektur procesorów

W tabeli nr 2 są przedstawione poszczególne architektury procesorów z podziałem na szczegółowo istotne cechy.

	ARM	MIPS	Atmel AVR	RISC-V	x86	MCS-51
Typ	RISC	RISC	RISC	RISC	CISC	CISC
Liczba bitów	32	32-64	8	32	16-64	8-32
Liczba rejestrów	15	4-32	32	32	8-32	4-32
Zastosowanie	Mobilne, IoT, serwery	Systemy wbudowane, IoT	Mikrokontrolery	Ogólne	Komputery osobiste, serwery	Proste systemy wbudowane
Złożoność	Średnia	Średnia	Niska	Zależna od implementacji	Wysoka	Niska
Wielowątkowość	Tak	Tak	Nie	Tak	Tak	Nie

Tabela 2. Porównanie architektur procesorów

Ze względu na konieczność zmieszczenia wielu procesorów na pojedynczym układzie FPGA musi on być jak najprostszy, najlepiej 8 bitowy, ale powinien też być kompatybilny ze współczesnymi narzędziami programistycznymi. Dlatego idealnym kandydatem jest Atmel AVR. Jest on wykorzystywany głównie w mikrokontrolerach dzięki czemu jest szeroko znanym zestawem instrukcji wśród projektantów systemów zabezpieczeń. Jest zaledwie 8-bitowy i stosunkowo prosty w implementacji.

3.1.3 Klasy architektur równoległych

Równolegle wykonywanie instrukcji pozwala na zwiększenie wydajności procesora bez przyspieszania działania jego zegara. Możemy wyróżnić 2 rodzaje równoległości w procesorach:

- równoległość przetwarzania danych
- równoległość wykonywania instrukcji

W przypadku równoległości na poziomie instrukcji, wykorzystuje się ją aby wykonać większą liczbę instrukcji na sekundę. Z kolei równoległość na poziomie procesora polega na współpracy wielu procesorów nad tym samym problemem. Oba podejścia mają swoje unikalne zalety.

W zależności od kombinacji tych dwóch technik można wyróżnić 4 główne rodzaje procesów[24]:

- Pojedynczy strumień instrukcji, pojedynczy strumień danych (SISD) to kategoria systemów jednordzeniowych. Programista postrzega ją jako standardowy komputer sekwencyjny, ale istnieje możliwość wykorzystania równoległości na poziomie instrukcji. Jest to najprostszy i najbardziej rozpowszechniony model procesora

- Pojedynczy strumień instrukcji, wiele strumieni danych (SIMD) to model, w którym ta sama instrukcja jest wykonywana przez wiele procesorów za pomocą różnych strumieni danych. Komputery SIMD wykorzystują równoległość na poziomie danych, wykonując te same operacje równolegle na różnych danych. Każdy procesor ma swoją własną pamięć danych, jednak istnieje wspólna pamięć instrukcji oraz procesor sterujący, który pobiera i rozsyła instrukcje.
- Wiele strumieni instrukcji, pojedynczy strumień danych (MISD) to kategoria, która dotychczas nie doczekała się komercyjnych multiprocesorów, ale uzupełnia klasyfikację architektur komputerowych.
- Wiele strumieni instrukcji, wiele strumieni danych (MIMD) to model, w którym każdy procesor pobiera własne instrukcje i operuje na własnych danych, koncentrując się na równoległości na poziomie zadań. MIMD jest generalnie bardziej elastyczny niż SIMD, co sprawia, że ma szersze zastosowanie, ale jest również droższy w implementacji. Komputery MIMD mogą korzystać z równoległości na poziomie danych, choć wiąże się to z większym narzutem niż w systemach SIMD. Wysoki narzut oznacza, że ziarno równoległości musi być wystarczająco duże, aby efektywnie wykorzystać dostępne zasoby.

Przedstawiony w tej pracy mikrokontroler należy do klasy procesorów MIMD. Zawiera on wiele niezależnych pamięci danych zawierających różne programy wykonywane niezależnie przez poszczególne procesory. Są to tylko komunikujące się ze sobą procesory SISD. Ze względu na brak wspólnej pamięci, procesory komunikują się ze sobą zewnętrznymi adresami. Pamięć jest rozdysytrybuowana. Klasycznie MIMD zawiera jedną pamięć programu. Tutaj ona też jest rozzielona tak, że każdy program jest w pełni niezależny. Przykładami procesorów o zbliżonej budowie są MPP (ang. massively parallel processors), COW (ang. clusters of workstations) i NUMA (ang. non-uniform memory access)[25]. Służą one jednak do wielkoskalowych obliczeń, a taka architektura nie została dotychczas komercyjnie zastosowana w mikrokontrolerach.

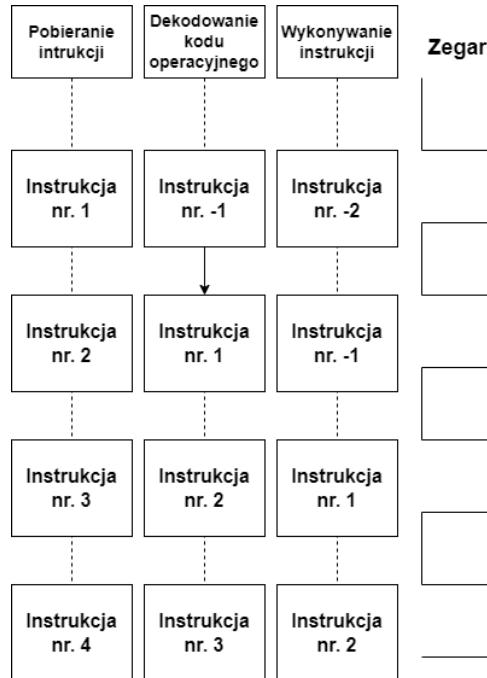
3.1.4 Potokowanie

Pobieranie instrukcji z pamięci stanowi istotne wąskie gardło w szybkości ich wykonywania. Aby złagodzić ten problem, komputery, począwszy od IBM Stretch (1959)[26], wprowadziły możliwość wstępnego pobierania instrukcji, co pozwala na ich wcześniejsze załadowanie do pamięci, aby były gotowe, gdy zajdzie potrzeba ich użycia. Instrukcje te przechowywane są w specjalnym zestawie rejestrów zwanym buforem pobierania wstępnego. Dzięki temu, gdy instrukcja jest potrzebna, można ją zazwyczaj pobrać z bufora, unikając opóźnień związanych z odczytem z pamięci.

Pobieranie wstępne dzieli wykonywanie instrukcji na dwie fazy: pobieranie i właściwe wykonywanie. Koncepcja potoku rozwija tę strategię, dzieląc wykonanie instrukcji na wiele etapów. Często jest ich kilkanaście lub więcej. Każdy obsługiwany jest przez dedykowany element sprzętowy, co umożliwia równoległe działanie [27].

Rysunek 6 przedstawia potok składający się z trzech modułów, znanych również jako etapy. W pierwszym cyklu zegara etap S1 pobiera instrukcję 1 z pamięci. W drugim cyku etap S2 dekoduje instrukcję 1, podczas gdy S1 zajmuje się pobieraniem instrukcji 2. W trzecim cyku

S3 wykonuje instrukcję 1, S2 dekoduje instrukcję 2, a S1 pobiera instrukcję 3. Dzięki temu następuje 3 krotne zwiększenie liczby wykonywanych instrukcji na sekundę bez zwiększania częstotliwości zegara. Potokowanie wprowadza kompromis między czasem wykonania instrukcji a przepustowością procesora. Właśnie takie potokowanie jest wykorzystywane przez procesory z serii AVR.



Rysunek 6. Etapy wykonywania instrukcji. Opracowanie własne.

3.2 Mikrokontrolery

Mikrokontroler to mały komputer na pojedynczym układzie scalonym. Zawiera on jeden lub więcej procesorów wraz z pamięcią i programowalnymi urządzeniami peryferyjnymi. Mikrokontrolery są przeznaczone do zastosowań wbudowanych, w przeciwieństwie do mikroprocesorów, które są pozbawione urządzeń wejścia/wyjścia, a są używane w komputerach osobistych lub innych aplikacjach ogólnego przeznaczenia.

Zmniejszając rozmiar i koszt, w porównaniu z konstrukcją wykorzystującą oddzielny mikroprocesor, pamięć i urządzenia wejścia/wyjścia, mikrokontrolery umożliwiają tanie cyfrowe sterowanie większą liczbą urządzeń i procesów. Mikrokontrolery sygnałów mieszanych są powszechnie, integrując komponenty analogowe potrzebne do sterowania cyfrowymi systemami elektronicznymi. W kontekście internetu rzeczy mikrokontrolery są ekonomicznym i popularnym sposobem zbierania danych o otaczającym je świecie.

Mikrokontrolery są zróżnicowane. Na ogół mają one zdolność do zachowania funkcjonalności podczas oczekiwania na zdarzenie, takie jak naciśnięcie przycisku lub inne przerwanie. Pobór mocy podczas uśpienia może wynosić zaledwie nanowaty[28], dzięki czemu wiele z nich dobrze nadaje się do długotrwałych zastosowań baterijnych. Inne mikrokontrolery mogą pełnić funkcje wymagające znacznej wydajności, w których mogą działać jak procesor sygnałowy, z wyższymi prędkościami zegara i zużyciem energii.

Są one szeroko stosowane w wielu dziedzinach, takich jak elektronika konsumencka, kontrolery przemysłowe, sprzęt medyczny, kontrolery pojazdów itp. Mogą one współpracować z różnymi urządzeniami peryferyjnymi w zależności od obszarów zastosowań, a one z kolei mogą działać w ramach różnych protokołów, takich jak I^2C , UART itd. Służą one ustandaryzowaniu sposobów transmisji danych. Mikrokontroler współpracuje z różnymi urządzeniami poprzez interfejsy komunikacyjne zgodnie ze standardowymi protokołami.

3.2.1 Architektury interfejsów

Komunikacja z podzespołami jest ustandaryzowana do wielu różnych protokołów. Różnią się one między innymi zastosowaniami, przepustowością i ilością wymaganych złączy. Każdy mikrokontroler zawiera wybrane interfejsy, które są obsługiwane przez procesor. Układ FPGA może obsługiwać dowolną liczbę dowolnych protokołów komunikacji. Najpopularniejsze z nich to:

- I^2C została zaprojektowana przez firmę Philips na początku lat 80-tych, aby umożliwić łatwą komunikację między wieloma komponentami znajdującymi się na tej samej płytce drukowanej. Ma bardzo szerokie zastosowania, do których można zaliczyć między innymi obsługę sterowników, LCD i LED, portów I/O, pamięci RAM i EEPROM, zegarów czasu rzeczywistego i przetworników analogowo-cyfrowych, ale też wszelkiego rodzaju czujników[29].
- SPI Magistrala szeregowego interfejsu peryferyjnego to synchroniczny szeregowy interfejs komunikacyjny używany do komunikacji na krótkich dystansach, głównie w systemach wbudowanych. Interfejs został opracowany przez firmę Motorola pod koniec lat 80-tych. Typowe zastosowania obejmują karty SD i wyświetlacze LCD[30].
- UART to jeden z najwcześniejnych sposobów komunikacji elektronicznej, używany do kontroli pierwszych komputerów. Dzięki swojej prostej budowie nie stracił na znaczeniu, a dzisiaj jest niezwykle popularny, używany między innymi do komunikacji z sensorami, modułami sieci bezprzewodowej oraz pomiędzy mikrokontrolerami[31].
- 1-wire to system magistrali komunikacyjnej zaprojektowany przez Dallas Semiconductor Corp., który zapewnia wolną komunikację i zasilanie przez pojedynczy przewód. Jest podobny w koncepcji do I^2C , ale z niższą szybkością transmisji danych i większym zasięgiem. Zwykle jest używany do komunikacji z małymi i niedrogimi urządzeniami, takimi jak cyfrowe termometry i przyrządy pogodowe[32].
- PWM (ang. Pulse Width Modulation) to metoda generowania sygnału analogowego przy użyciu źródła cyfrowego. Pozwala na sterowanie prostymi urządzeniami[33].

3.2.2 Zastosowania mikrokontrolerów

Współcześnie procesory są wykorzystywane w każdym aspekcie życia codziennego. Jednak w zależności od docelowego zastosowania, będą one miały inne wymagania. Można wyróżnić 5 głównych zastosowań, których specyfikacje są przedstawione w tabeli numer 3. W ramach tej pracy bardziej rozwinięte są jedynie zastosowania wbudowane, ponieważ tylko one są stosowane w systemach zabezpieczeń.

Cecha	Systemy wbudowane	Smartfony, tablety	Komputery osobiste	Serwery	Makro-komputery
Cena systemu [zł]	10 – 500 000	100 – 10 000	1000 – 10 000	od 10 000	od 500 000
Cena procesora [zł]	0,05 – 500	10 - 1000	100 - 3000	1000 – 10 000	200 – 1000
Cechy konstrukcyjne	Cena, zużycie energii, specyficzne zastosowania	Koszt, zużycie energii, wydajność sieciowa, responsywność graficzna,	Koszt, wydajność, zużycie energii, wydajność,	Skalowalność, stosunek wydajności do ceny i mocy.	Stosunek wydajności do ceny i mocy.

Tabela 3. Wymagania procesorów w najważniejszych zastosowaniach[34]

3.2.2.1 Systemy wbudowane

Urządzenie wbudowane to wyspecjalizowany system komputerowy zawierający procesor, pamięć i urządzenia zewnętrzne. Są wbudowane jako część większego urządzenia. Często musi działać w czasie rzeczywistym, na bieżąco przetwarzając dane. W 2023 roku sprzedano łącznie 2,66 mld mikroprocesorów, z pośród których 98% zostało wykorzystanych w systemach wbudowanych[35].

Ze względu na wyspecjalizowanie urządzenia do wykonywania konkretnego zadania, mogą one być zoptymalizowane pod kątem ceny i wydajności. Większość zastosowań nie potrzebuje wysoce wydajnych, nowoczesnych procesorów, a równie dobrze mogą wykorzystywać modele i architektury sprzed wielu lat. Nowoczesne systemy wbudowane są najczęściej oparte na mikrokontrolerze, ale w zaawansowanych rozwiązańach spotyka się też procesory, które zwykle są wyspecjalizowane do wykonywania konkretnego zadania. Takie procesory przetwarzające sygnały nazywa się DSP (ang. Digital Signal Processor)[36].

Urządzenia wykorzystujące systemy wbudowane to szeroka gama produktów zawierająca prawie każde urządzenie wykorzystujące technikę cyfrową. Można wymienić między innymi cyfrowe zegarki, urządzenia AGD, roboty, linie produkcyjne, samochody czy urządzenia medyczne[36].

3.2.2.2 Procesory w systemach wbudowanych

Koszt i zużycie energii są często równie istotne, co wydajność w branży urządzeń wbudowanych. Oprócz ceny modułu procesora, która obejmuje wszystkie niezbędne układy interfejsowe, pamięć często stanowi drugi najdroższy element systemu wbudowanego. W przeciwieństwie do systemów stacjonarnych i serwerowych, wiele systemów wbudowanych nie ma dodatkowej pamięci masowej, a całe oprogramowanie musi być przechowywane w pamięci FLASH lub DRAM. W przypadku urządzeń przenośnych które są ograniczone zarówno pod względem kosztów, jak i rozmiaru, ilość pamięci potrzebnej dla aplikacji jest kluczowa. Zużycie energii staje się także decydującym czynnikiem przy wyborze procesora, szczególnie w systemach zasilanych baterijnie.

W aplikacjach wbudowanych często występują wymagania związane z czasem rzeczywistym. Taki wymóg oznacza, że dany segment aplikacji ma maksymalny czas wykonania, który nie może być przekroczony. Na przykład w cyfrowych dekoderach czas przetwarzania każdej klatki wideo jest ograniczony, ponieważ procesor musi przetworzyć jedną klatkę przed nadaniem następnej.

Systemy wbudowane obejmują bardzo szeroką gamę urządzeń komputerowych. Na przykład procesor TI 320C55 DSP, zaprojektowany do zastosowań wbudowanych, jest procesorem przypominającym architekturę RISC, z dokładnie dostosowanymi możliwościami. Z kolei TI 320C64x to wydajny, ośmiowątkowy procesor VLIW, stworzony do bardziej wymagających zadań. Niektóre procesory muszą działać tylko na zasilaniu baterijnym, podczas gdy inne mogą korzystać z zasilania sieciowego. Wszystkie te procesory łączy potrzeba przetwarzania sygnałów dla aplikacji wbudowanych.

Decyzje architektoniczne, które są praktyczne dla aplikacji ogólnego przeznaczenia, takie jak podział na wiele poziomów buforowania, nie są tak pożądane w systemach wbudowanych. Wynika to z ograniczeń dotyczących obszaru układu scalonego, kosztów, zużycia energii oraz wymagań czasowych. Model programowania w tych systemach stawia większe wymagania przed programistami i kompilatorami, aby skutecznie wykorzystywać równoległość.

3.2.2.3 Multiprocesory w systemach wbudowanych

Wielordzeniowe procesory są obecnie powszechnie używane w komputerach stacjonarnych i serwerach. W kontekście systemów wbudowanych wiele specjalistycznych projektów wykorzystywało niestandardowe multiprocesory. Często w takich projektach znajduje się procesor ogólnego przeznaczenia lub DSP (ang. Digital Signal Processor), współpracujący z urządzeniami zewnętrznymi. Tego typu architektura multiprocesorowa jest już powszechna w różnych zastosowaniach, od grafiki komputerowej i przetwarzania multimedialnych po telekomunikację. Choć interakcje między procesorami są zorganizowane i stosunkowo proste, głównie polegające na podstawowych kanałach komunikacyjnych, kluczowym wyzwaniem jest zapewnienie poprawności protokołów komunikacyjnych między procesorami wejścia/wyjścia a procesorem ogólnego przeznaczenia.

W systemach wbudowanych coraz częściej wykorzystuje się multiprocesory składające się z kilku procesorów ogólnego przeznaczenia, które są wykorzystywane głównie w zaawansowanych modułach telekomunikacyjnych i sieciowych, gdzie kluczowa jest skalowalność. Wiele aplikacji w przestrzeni wbudowanej naturalnie wykorzystuje równoległość zapewnianą przez mnogość rdzeni, szczególnie w zaawansowanych zastosowaniach. Przykładem może być procesor MXP zaprojektowany przez empowerTel Networks do systemów głosowych działających w sieci IP. Procesor MXP składa się z czterech głównych komponentów:

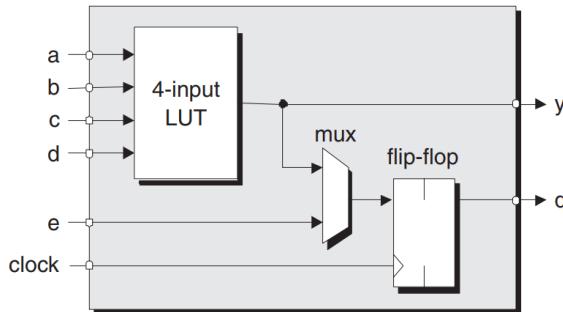
- interfejs do szeregowych strumieni głosowych
- obsługa szybkiego trasowania i wyszukiwania kanałów
- interfejs Ethernet
- cztery procesory MIPS32 klasy R4000, każdy z własną pamięcią podręczną

Procesory MIPS są wykorzystywane do obsługi kanałów głosowych, zapewniając eliminację echa, prostą kompresję i kodowanie pakietów. Aby uruchomić jak najwięcej niezależnych strumieni głosowych, zastosowanie wieloprocesorowości okazuje się idealnym rozwiązaniem[36].

3.3 Układy FPGA

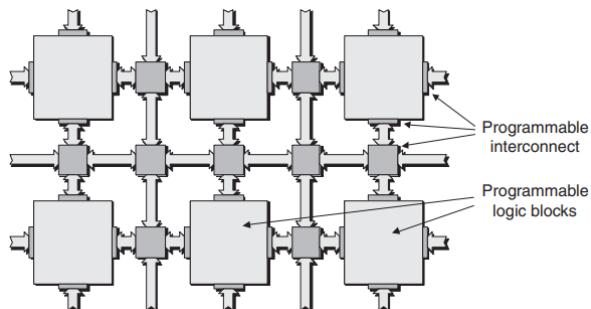
Układy FPGA(ang. Field-Programmable Gate Array) umożliwiają projektantom implementację cyfrowych obwodów logicznych. Składają się z kilku kluczowych elementów:

- Programowalne bloki logiczne (ang. Logic Blocks) to podstawowe elementy funkcjonalne układu FPGA. Składają się z LUT-ów (ang. Look-Up Tables), które realizują funkcje logiczne poprzez przechowywanie wyników dla wszystkich możliwych kombinacji wejść. Zawierają również przerzutniki (ang. flip-flops) lub rejesty do przechowywania stanów logicznych. Są odpowiedzialne za wykonywanie operacji logicznych. Na rysunku 7 przedstawiono składowe prostego, 4-wejściowego bloku LUT. Funkcje logiczne są wykonywane w pierwszym bloku wejściowym.



Rysunek 7. Widok przedstawiający składowe prostego bloku LUT.[37]

- 2. Matryca połączeń programowalnych (ang. Programmable Interconnects) to sieć przewodów i przełączników, które umożliwiają połączenie bloków logicznych oraz innych komponentów FPGA. Na rysunku 8 przedstawiono połączenia blogów logicznych wewnętrz układów FPGA. Umożliwiają one dostosowania topologii połączeń do zaprogramowanej architektury obwodu.

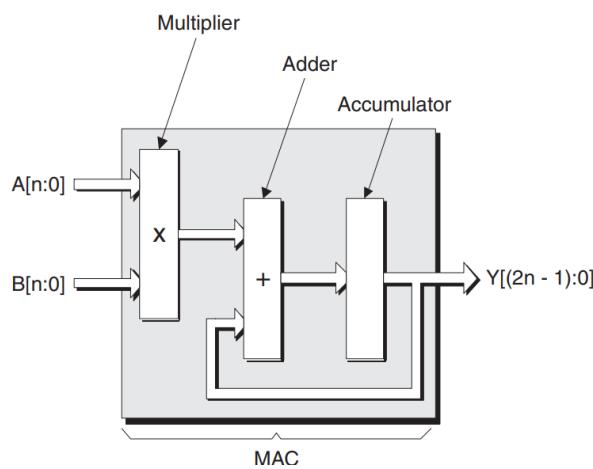


Rysunek 8. Schemat ilustrujący działanie układu FPGA[37]

- Bloki wejścia/wyjścia (ang. I/O Blocks) Służą do komunikacji FPGA ze światem zewnętrznym. Obsługują różnorodne standardy sygnałowe.

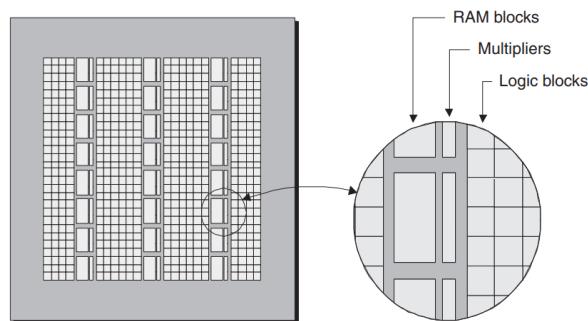
- Bloki dedykowane (ang. Dedicated Blocks)- specjalistyczne elementy, które zwiększą wydajność FPGA w określonych zastosowaniach:

– Bloki DSP (ang. Digital Signal Processing) są zoptymalizowanymi jednostkami dedykowanymi do wykonywania operacji matematycznych, takich jak mnożenie, sumowanie czy przesunięcia bitowe. Są kluczowe w aplikacjach wymagających intensywnego przetwarzania sygnałów, np. w systemach komunikacyjnych, obrazowaniu czy sztucznej inteligencji, zapewniając wysoką wydajność przy niskim zużyciu zasobów FPGA. Składa się on z układu mnożącego oraz sumującego. Na rysunku 9 przedstawiono budowę takiego bloku, a na rysunku 10 pokazano ich lokację pomiędzy blokami logicznymi umożliwiającą ich pełną integrację z pozostałymi elementami logicznym układu FPGA.



Rysunek 9. Budowa elementu bloku DSP[37]

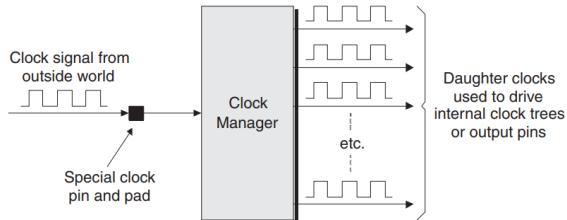
- Pamięci RAM/ROM to specjalne bloki pamięci do przechowywania danych. Występują często w postaci pamięci BRAM (Block RAM). Służą do dynamicznego przechowywania danych, natomiast ROM przechowuje stałe wartości, takie jak tablice LUT. Są one wykorzystywane w różnych zastosowaniach, od buforowania po implementację algorytmów. Na rysunku 10 pokazano ich lokację pomiędzy blokami logicznymi zapewniającą szybki transfer danych z pamięci do logiki.



Rysunek 10. Schemat przedstawiający bloki RAM i DSP (tutaj nazwane Multipliers) w układzie FPGA [37]

- Procesory znajdują się w niektórych układach FPGA. Takie rdzenie procesorowe umożliwiające wykonywanie oprogramowania.

- Matryca zegarowa (ang. Clock Management) zawiera obwody zarządzające zegarem. Umożliwia stabilne taktowanie układu oraz różne częstotliwości pracy dla poszczególnych części FPGA. Na rysunku 11 przedstawiono działanie matrycy zegarowej, która z jednego zewnętrznego zegara generuje kilka sygnałów zegarowych o zmienionej częstotliwości.



Rysunek 11. Matryca zegarowa generująca sygnały zegarowe[37]

Programowanie lub konfiguracja FPGA polega na wprowadzeniu do pamięci konfiguracyjnej strumienia bitów, który zawiera wartości kontrolujące funkcje logiczne i punkty przełączania połączeń. Główną zaletą FPGA w porównaniu z układami wykonanymi na zamówienie jest to, że projektant nie musi zajmować się procesem projektowania układów scalonych[38].

FPGAs, obok procesorów komputerów osobistych, są pionierami w zastosowaniu zaawansowanej technologii. Współczesne FPGAs produkowane są w technologii 12 nm, podczas gdy większość projektantów ASIC wciąż pracuje w oparciu o starsze technologie.[39] Najważniejszą zaletą FPGA jest ich możliwość rekonfiguracji. Wadą FPGA jest to, że wiążą się z niepotrzebnymi kosztami, prędkością i zużyciem energii w porównaniu z układami wykonanymi na zamówienie. Niemniej jednak, głównie z powodu wysokich kosztów produkcji układów scalonych, FPGA mogą być stosowane w coraz większej liczbie zastosowań, nie tylko jako prototypy, ale także jako część finalnego produktu. Jedynie produkty o bardzo wysokim wolumeniu i wysokiej wrażliwości na zużycie energii pozostają poza zasięgiem FPGA[38]. Obecnie notują wzrost sprzedaży w wysokości 10% rocznie[40].

3.3.1 Implementacje procesorów na układach FPGA

We współczesnych układach FPGA procesory znajdują się obok układu, towarzysząc mu w wykonywaniu różnych obliczeń[37]. Taki układ nazywa się SOC (ang. system on chip). Jednak w celu eksploracji i rozwijania architektury samych procesorów można je zaimplementować na układzie FPGA. Takie implementacje nazywają się soft core w odróżnieniu od hard core, które nie podlegają konfiguracji.

Firma OpenCores[41] oferuje szereg rdzeni i innych komponentów, które można pobrać i nagrać na układ FPGA, który będzie działał jak prawdziwy procesor. ARM oferuje niektóre ze swoich mikrokontrolerów za darmo do użytku na układach FPGA[42].

Jednym z czynników, które przyczyniły się do ostatnich trendów w projektowaniu procesorów, jest dojrzałość narzędzi programowych do modyfikacji architektury procesorów, projektowania zestawów instrukcji oraz rozwoju narzędzi programowych. Powstały różne metodologie projektowania, które podchodzą do projektowania procesorów z różnych perspektyw, w tym podejście oparte na istniejącej architekturze podstawowej, eksploracja architektury oparta na kompilatorach oraz projektowanie w języku opisu procesora[38].

Rozdział 4: Cel i założenia pracy

Celem pracy było opracowanie i implementacja parametryzowalnego wieloprocesorowego systemu FPGA do implementacji modułów zabezpieczeń. Przyjęto następujące założenia pracy:

- implementacja rozwiązania na układzie FPGA
- opracowanie wieloprocesorowej implementacji, z niezależnie i równolegle pracującymi blokami
- opracowanie systemu umożliwiającego jednoczesny odczyt danych z wielu różnych czujników wykorzystujących różne protokoły komunikacji
- parametryzowalność rozwiązania: definiowania liczby i rodzajów interfejsów komunikacyjnych
- możliwość zaprogramowania algorytmu użytkownika
- komunikacja z użytkownikiem, rozumiana jako odczyt danych z czujników, pracy algorytmu itp.

W ramach pracy wykonano szereg zadań niezbędnych do osiągnięcia celu:

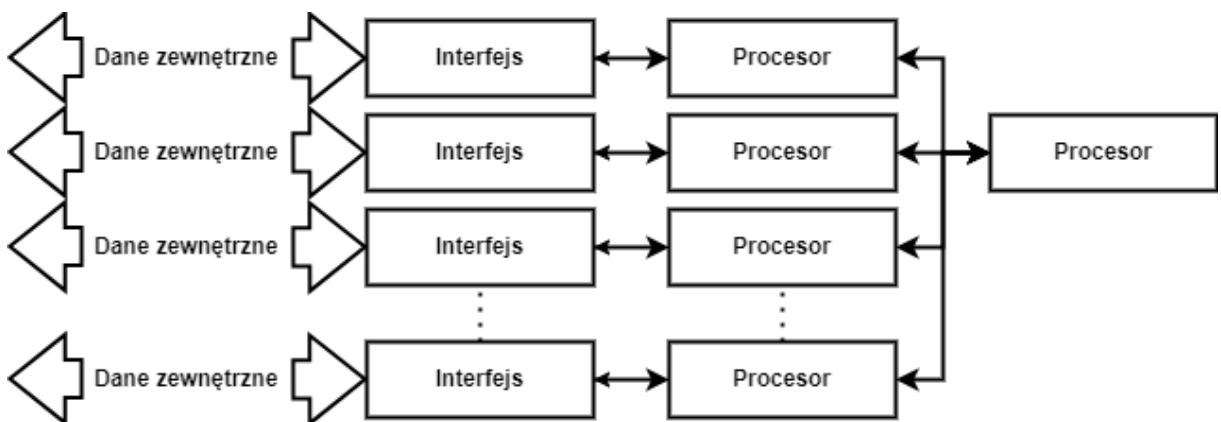
- zaprojektowano mikroprocesor implementujący zestaw instrukcji Atmel AVR
- zaprojektowano wielordzeniowy mikrokontroler
- zaimplementowano interfejsy UART, I2C i SPI
- opracowano moduł komunikacji z użytkownikiem
- napisano biblioteki programistyczne w celu ułatwienia programowania
- przetestowano działanie układu na układzie FPGA

Przyjęto również, iż praktyczna realizacja będzie z wykorzystaniem:

- układów FPGA firmy Xilinx
- plafromy deweloperskiej Zybo Z7
- przykładowych interfejsów komunikacyjnych i czujników

Rozdział 5: Opracowanie architektury systemu zabezpieczeń

W celu spełnienia wymagań pracy niezbędne jest zaprojektowanie wielordzeniowego procesora, którego rdzenie będą niezależnie otrzymywać dane poprzez interfejsy, ale ich działania będą mogły być koordynowane. Propozycje takiego rozwiązania przedstawiono na rysunku 12. Pozwala ono na wyodrębnienie przetwarzania i analizowania różnych danych przy zachowaniu komunikacji pomiędzy procesami, pozwalając na analizę porównawczą danych z różnych urządzeń. Dodatkowy procesor, pełniący funkcję kontrolera pozostałych procesorów zapewnia, że żaden z biorących udział w przetwarzaniu danych procesorów nie będzie dodatkowo obciążony koordynowaniem działań.

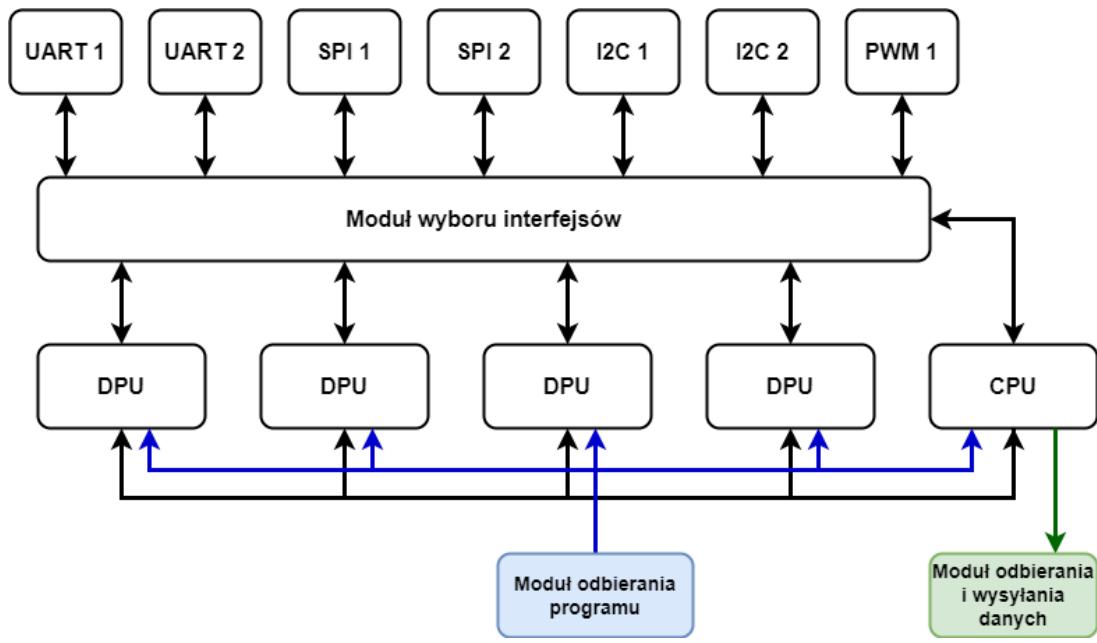


Rysunek 12. Schemat przedstawiający ogólną zasadę działania urządzenia. Opracowanie własne.

Do komunikacji z większością urządzeń wystarczy implementacja kilku najpopularniejszych interfejsów, będących w stanie porozumiewać się z prawie każdym czujnikiem. W tym celu należy zaimplementować przedstawione wcześniej protokoły komunikacji. To, jakie interfejsy są używane jest elastyczne, ponieważ w razie potrzeby można dodać inny interfejs na układ FPGA.

Użytkownik musi mieć możliwość łatwego zaprogramowania urządzenia, bez konieczności nauki nowego języka programowania. Bez tego nie będzie ono przydatne dla szerszego grona odbiorców. Najpopularniejszym językiem programowania mikrokontrolerów jest język C i z tego względu został wybrany do zastosowania w projekcie. Są dostępne jego kompilatory na wszystkie współcześnie używane zestawy instrukcji. Do komplikacji na zestaw instrukcji AVR zostanie wykorzystany kompilator AVR-GCC.

Na rysunku 13 przedstawiono ogólny schemat działania architektury RSZ. Jego poszczególne elementy są przedstawione dalej w rozdziale. Różnymi kolorami przedstawiono komunikację z poszczególnymi modułami w celu czytelniejszego zobrazowania tego, które moduły z czym są w stanie się komunikować. Przedstawiono na nim jednostki procesora, które nazywają się DPU i CPU.



Rysunek 13. Schemat działania RSZ. Opracowanie własne.

5.1 Procesor

Procesory są odpowiedzialne za większość funkcji mikrokontrolera. Powinny wykonywać podstawowe operacje arytmetyczne, manipułować pamięcią i zajmować jak najmniej zasobów sprzętowych, to znaczy być jak najmniejsze. Dzięki temu na układzie FPGA będzie możliwość zmieszczania wielu z nich, co zwiększy możliwości urządzenia.

W celu wykorzystania gotowego kompilatora należy wybrać gotową architekturę zestawu instrukcji. Powinna być ona 8-bitowa, a jednocześnie wspierać współczesne narzędzia programistyczne, pozwalać na obsługę przerwań oraz obsługiwać wystarczającą pamięć dla długich programów. Powinna być ona popularnie stosowana przez mikrokontrolery. W celu uproszczenia implementacji powinna ona być rodzaju RISC.

Biorąc te czynniki pod uwagę, najlepszą opcją jest architektura Atmel AVR. Dzięki temu, że działa na 8-bitowych liczbach, zajmuje mało zasobów w porównaniu do innych mikroprocesorów. Jest powszechnie stosowany przez mikrokontrolery Arduino. Najbardziej użyteczne dla tego zastosowania są procesory ATmega. Realizacja tego procesora jest omówiona w osobnym rozdziale. Większość instrukcji ma 16 bitów, ale występuje też kilka 32-bitowych. W dalszej części pracy 16 bitów będzie oznaczane jako jedno słowo, a 32 jako dwa słowa.

Procesory komunikują się z innymi modułami poprzez 5 sygnałów:

- Adresu - jest 5 bitowy i przekazuje to, jaki adres procesor chce odczytać lub do jakiego adresu chce coś zapisać.
- Danych wejściowy - zawiera 8-bitowe dane odczytywane przez procesor.
- Danych wyjściowy - zawiera 8-bitowe dane zapisywane przez procesor do innego rejestru.
- Zapisu - 1 bitowy sygnał, który jest ustawiany kiedy procesor chce zapisać rejestr.
- Odczytu - 1 bitowy sygnał, który jest ustawiany kiedy procesor chce odczytać rejestr.

Widoczne na rysunku 13 procesory DPU (ang. Data Processing Unit), dla odróżnienia od CPU zajmują się czym innym, ale są zaimplementowane tak, jak jednostka nadzorująca CPU (ang. Central Processing Unit). Te procesory obsługują inne rejestrze zewnętrzne, a przez to mają inne funkcje, ale mogą wykonywać te same programy.

DPU jest odpowiedzialne za komunikację z interfejsem oraz CPU. Oprócz tego, to DPU sprawdza, czy odczytane przez czujnik wartości są odpowiednie i może powiadomić o tym procesor.

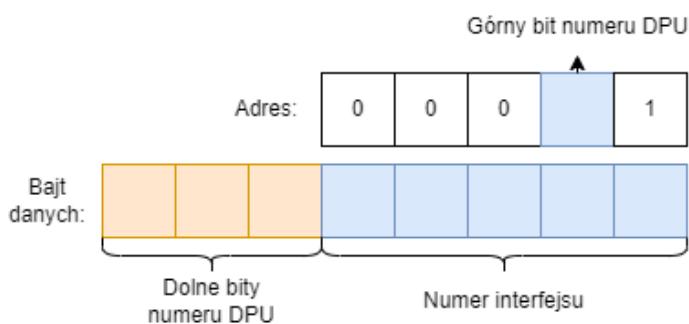
CPU nadzoruje pracę RSZ. Zarządza pracą DPU, wysyła mu informację o podpięciu do danego interfejsu lub porównuje ze sobą dane odebrane z różnych DPU. Jest odpowiedzialny za szereg czynności:

- komunikację z użytkownikiem
- komunikację z przyciskami
- komunikację z diodami
- komunikacje z DPU
- komunikacje z modułem wyboru interfejsów

5.2 Realizacja modułu wyboru interfejsów

Zgodnie z opracowaną architekturą przedstawioną na rysunku 12 każdy z CPU obsługuje pojedynczy interfejs. Realizowane to jest za pomocą modułu wyboru interfejsów. Moduł jest połączony z DPU poprzez sygnały adresu, danych, przerwań z interfejsów, zapisu i odczytu. Te same sygnały są transmitowane do interfejsu. Układ definiuje to, do którego interfejsu trafią sygnały z procesora, oraz z którego interfejsu trafią sygnały do danego procesora.

CPU jest odpowiedzialne za dobieranie DPU do interfejsów. Realizuje to poprzez zapisanie danych do specjalnego rejestru tak, jak przedstawiono na schemacie 14.



Rysunek 14. Wybieranie interfejsu przez CPU. Opracowanie własne.

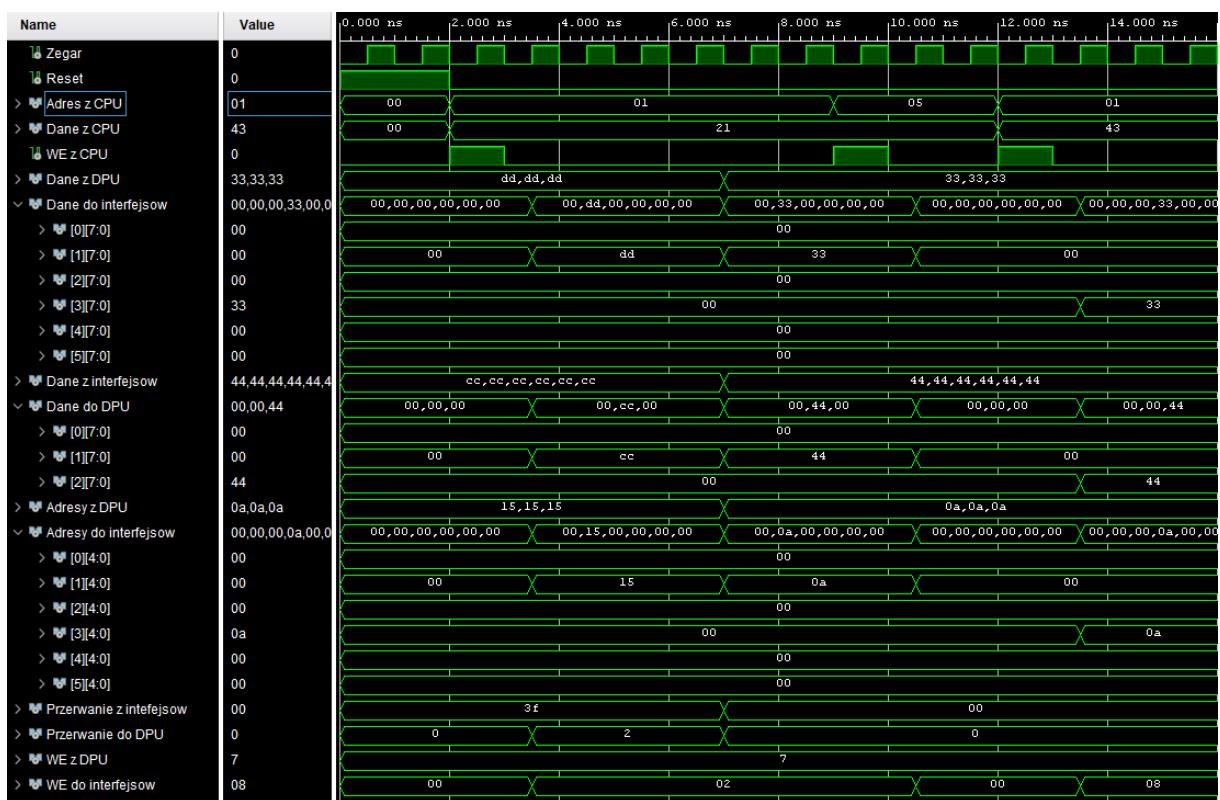
To, który interfejs zostanie wybrany zależy od tego, za pomocą jakiego adresu moduł jest wskazywany. Ogranicza to maksymalną liczbę interfejsów do 32. Takie rozwiązanie ułatwia i przyspiesza wybranie interfejsu oraz wprowadza odporność na błędy w przypisaniu.

Po wybraniu danego połączenia procesora i interfejsu, muszą one zostać odłączone przez CPU, zanim zostaną połączone z innym modułem. Dzieje się to w analogicznym sposobie, jak przy ich wybraniu. Zapobiega to ewentualnym konfliktom, w których więcej niż jeden sygnał miałby trafić do tego modułu.

5.2.1 Symulacje modułu wyboru interfejsów

W celu weryfikacji poprawności działania modułu niezbędnie było wykonanie jego symulacji. Jej wybrane sygnały są przedstawione na rysunku 15.

Najpierw CPU wysłało informacje o przypisaniu pierwszego interfejsu do pierwszego DPU. Po jednym cyklu zegara dane do pierwszego interfejsu zaczęły zawierać dane z DPU, podobnie jak inne sygnały. Sygnały mające trafić za pośrednictwem modułu do interfejsu lub DPU zmieniły się jednokrotnie, aby sprawdzić, czy sygnały wyjściowe natychmiastowo zmienią swoją wartość. Można to zaobserwować w 7. ns symulacji w której zgodnie z oczekiwaniem, sygnały wyjściowe natychmiast przybrały nową wartość pomimo spadającego zbocza zegara. Następnie, w 9. ns symulacji, CPU odłączył interfejs od DPU. Widać, że na wyjściach z modułu zaczęły być same zera sygnalizujące brak transmisji. W celu sprawdzenia możliwości ponownego połączenia, czwarty interfejs został połączony z drugim DPU. Skutek tego był taki sam jak wcześniejszego połączenia, a wszystkie sygnały mają przebieg zgodny z oczekiwany.



Rysunek 15. Przebieg wybranych sygnałów symulacji modułu wyboru interfejsów. Opracowanie własne.

5.3 Realizacja komunikacji z użytkownikiem

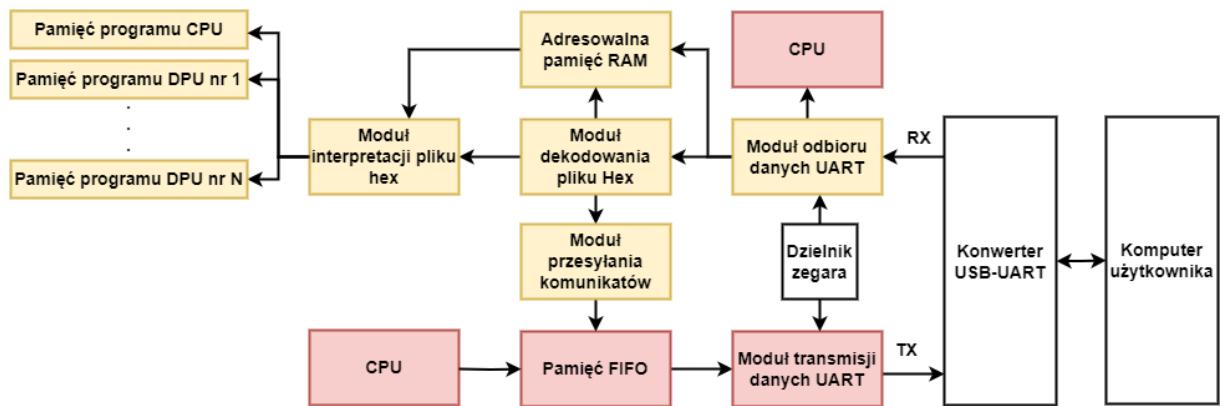
Użytkownik mikrokontrolera musi mieć możliwość odczytywania danych wysyłanych przez urządzenie, w tym podglądu jego działania. Aby mikrokontroler był możliwy do użytku, niezbędne jest umożliwienie zmiany wgranych przez użytkownika programów. W tym celu wymagane jest stworzenie modułu umożliwiającego przesyłanie przez użytkownika danych, przetwarzanie ich i wpisywanie do wybranych pamięci programu.

Na rysunku 16 jest przedstawiona realizacja tego modułu. Na czerwono są zaznaczone bloki wysyłania danych, a na żółto bloki odbioru danych pamięci programu. Na rysunku 13 przedsta-

wiono, z jakimi modułami możliwa jest bezpośrednia komunikacja, a z jakimi tylko wgrywanie programu.

Komunikacja odbywa się za pomocą protokołu komunikacji UART, w którym pośredniczą wcześniej omówione moduły transmisji i odbioru danych. Jego ustawienia są definiowane przed wgraniem mikrokontrolera na układ FPGA i nie podlegają zmianie. W trakcie testów ma on następujące ustawienia:

- $bod = 115200$
- bity zatrzymania = 2
- brak bitu parzystości
- 8 bitów danych



Rysunek 16. Schemat przedstawiający moduł komunikacji z użytkownikiem. Opracowanie własne.

5.3.1 Moduł transmisji danych do użytkownika

Jedynie CPU ma możliwość komunikacji z użytkownikiem. Może otrzymywać przerwania w przypadku wysłania przez użytkownika danych, po czym je odczytać. W celu wysłania danych może zapisać je do pamięci FIFO, z której zostaną wysłane. Umożliwia to płynne wysyłanie danych. DPU nie mają możliwości komunikacji z użytkownikiem, ze względu na trudne do rozwiązania konflikty między poszczególnymi procesorami oraz wysokie zużycie zasobów, w przypadku równoległego odbierania danych od poszczególnych DPU. Lepiej wykorzystać gotowe mechanizmy rozwiązywania konfliktów w transmisji danych między DPU a CPU. Transmisja odbywa się za pomocą wpisywania danych przez procesor do pamięci FIFO. Są one następnie wysyłane za pomocą interfejsu UART.

Dodatkowo istnieje możliwość odebrania wysłanych przez użytkownika danych. Jeżeli zostaną odebrane dane które nie są częścią transmitowanego programu, zostają one wysłane do CPU za pomocą przerwania.

5.3.2 Moduł wgrywania programu CPU/DPU

Przesyłanie danych do poszczególnych pamięci programu odbywa się za pomocą transmitowania danych uzyskanych na podstawie plików Intel Hex[43], które są uzyskiwane z kompilatora. Został on wybrany ponieważ umożliwia łatwy zapis danych do dowolnego adresu pamięci oraz

jest powszechnie stosowany przez kompilatory. Każda pamięć programu ma zapisany osobny program który musi zostać zapisany.

5.3.2.1 Budowa pliku Intel HEX

Plik Intel HEX zawiera dane, które mają zostać zapisane do pamięci programu. Są one reprezentowane przez znaki ASCII. Liczby są zapisywane w kodzie szesnastkowym. Każda jego linijka ma strukturę przedstawioną na rysunku 17. Jest ona podzielona na kilka elementów:

- Znak ": pokazuje rozpoczęcie nowej linijki.
- Liczba bajtów danych- może być ich między 0 a 255, ale zwykle nie używa się wartości większych niż 16.
- Adres pamięci- do niego mają zostać zapisane dane. 4 znaki pozwalają na zapisanie 16 bitowego adresu. Fizyczny adres jest obliczany poprzez dodanie do niego adresu bazowego.
- Typ rekordu- określa znaczenie bajtów danych. Jest możliwych 6 różnych typów rekordu. Są to:
 - 00 - dane mające zostać zapisane do pamięci
 - 01 - koniec pliku. Jako jedyny nie używa bajtów danych.
 - 02 - rozszerzony adres segmentu. Nie jest używany, ale został zaimplementowany w celu zachowania kompatybilności z różnymi kompilatorami. Pozwala na 20 bitowe adresowanie.
 - 03 - rozpoczęcie adresu segmentu. Definiuje wartość wskaźnika pamięci programu (PC). Ta funkcjonalność nie jest zaimplementowana ze względu na niezrealizowanie debugowania mikrokontrolera.
 - 04 - adres bazowy. Jest używany w celu adresowania 32 bitowego i zawiera jego górne 2 bajty.
 - 05 - początek adresu funkcji main. Ta funkcjonalność nie jest zaimplementowana ponieważ nie jest przydatna[43].
- Dane
- Suma kontrolna. Jest ona dolnymi 8 bitami sumy wszystkich pozostałych wartości liczbowych z danej linijki.

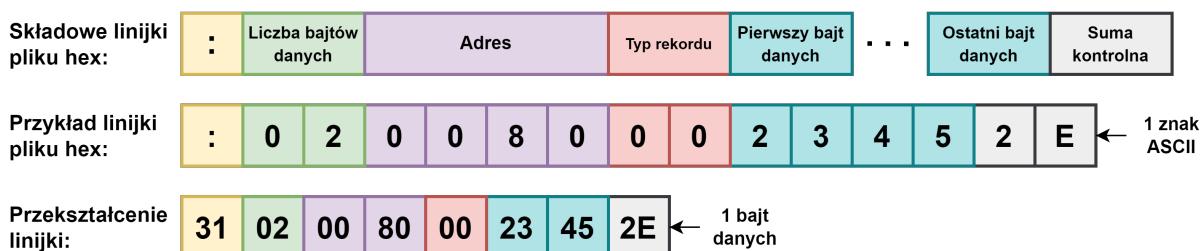
Zgodnie z tym, przykładowa linijka na rysunku 17 przedstawia zapisanie danych 23 do adresu 0x0080 i 24 do adresu 0x0081 8 bitowej pamięci. W przypadku używania 16 bitowej pamięci programu tak, jak projektowany mikrokontroler, dane 0x4523 są zapisywane do adresu 0x0080. W związku z tym wymagane jest zamienienie odczytanych bajtów kolejnością.

5.3.2.2 Budowa wysyłanych danych

Ze względu na specyficzną budowę mikrokontrolera, który zawiera wiele niezależnych pamięci programów, niezbędne było wprowadzenie pewnych modyfikacji do wysyłania danych za pomocą pliku hex.

Poszczególne pliki hex są łączone ze sobą tak, że tylko na końcu wysyłanych danych zawarty jest kod kończący plik. Pomiędzy nimi znajdują się kody rozszerzonego adresowania, wskazujące to, do której pamięci programu są wpisywane dane.

W celu szybszego przesyłania danych plik jest zamieniany do postaci liczbowej, co pozwala na dwukrotne zmniejszenie czasu transmisji. Oznacza to, że zamiast wysyłania znaków reprezentujących daną liczbę, wysyłany jest bajt zawierający tą liczbę. Znaki ":" są zamieniane na jego reprezentację w kodzie ASCII. Na rysunku 17 przedstawiona jest przekształcona przykładowa linijka danych.



Rysunek 17. Ramka komunikacji pliku hex wraz z przykładową linijką. Opracowanie własne.

Ten wysyłany plik jest realizowany poprzez specjalny skrypt napisany w języku Python. Łączy on różne pliki typu Intel hex tworząc nowy, który dla odróżnienia ma format - .hexn.

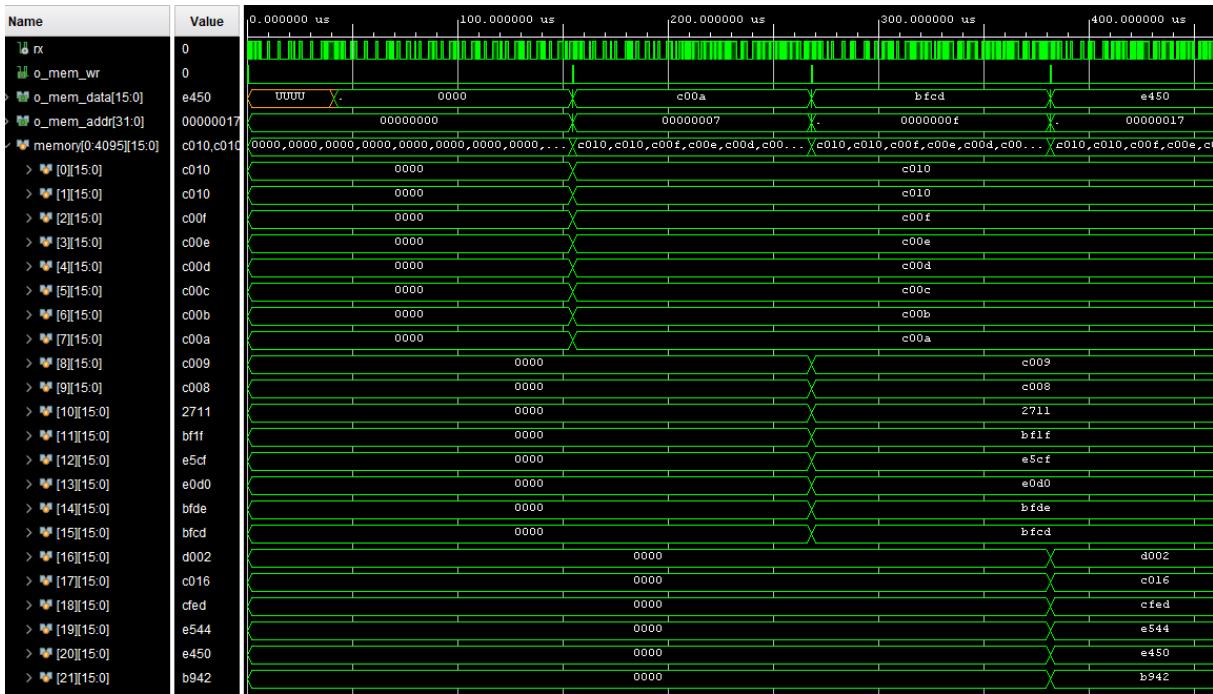
5.3.2.3 Realizacja wgrywania programu

Moduł dekodowania pliku hex jest odpowiedzialny za odczytanie ciągu danych i ich podział na poszczególne fragmenty każdej linijki odbieranego pliku. Jest on zrealizowany jako maszyna stanów której stan zmienia się w zależności od liczby odczytywanych bajtów. Gdy moduł wykryje, że odbiera bajty danych, są one zapisywane bezpośrednio do 8-bitowej adresowej pamięci RAM. Następnie sprawdzane jest, czy odczytana suma kontrolna zgadza się z docelową. Jeżeli się zgadza, wcześniej odczytywany adres, typ rekordu i dane są przesyłane do modułu wpisywania danych do pamięci programu. W przeciwnym przypadku, pamięć jest czyszczona i wysyłany jest komunikat o błędzie.

Po rozpoznaniu rozpoczęcia lub zakończenia transmisji pliku, wysyłane są do użytkownika komunikaty, które mają za zadanie poinformować go o tych wydarzeniach. Są to, zapisane w kodzie ASCII[44] komunikaty odpowiednio: „hex received” i „hex finished”.

Zadaniem modułu wpisywania danych do pamięci programu jest zapisanie danych do pamięci na podstawie typu rekordu. Jedynym rekordem, który nie jest przez niego obsługiwany, jest koniec pliku, ponieważ zmienia on stan modułu dekodowania. Odczytuje on 8-bitowe dane zapisane do pamięci RAM, łączy je w 16-bitowe dane, po czym zapisuje je do odpowiedniego adresu w pamięci danych.

Do modułu przesyłania komunikatów wiadomość trafia w postaci jednej długiej wiadomości. To ten moduł jest odpowiedzialny za podział wysyłanego komunikatu na pojedyncze bajty i ich



Rysunek 18. Przebieg wybranych sygnałów symulacji zapisu pamięci programu. Opracowanie własne.

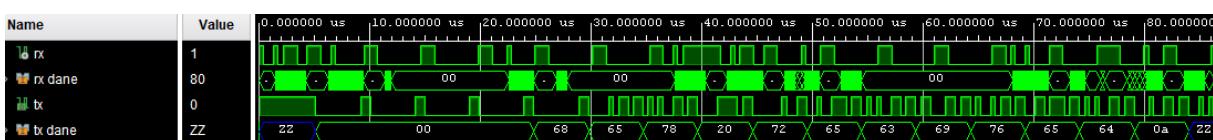
wysłanie do pamięci FIFO. W przypadku próby przesyłania danych zarówno przez ten moduł, jak i procesory, procesory mają pierwszeństwo, a komunikat jest ignorowany i nie zostaje wysłany.

5.3.2.4 Symulacje wgrywania programu

W celu przeprowadzenia pełnej symulacji wgrywania programu cały plik formatu .hexn został przesłany za pomocą symulowanego sygnału protokołu komunikacji UART. Jego dane powinny zostać zapisane do odpowiednich pamięci programu reprezentując poszczególne instrukcje programu. Aby sprawdzić poprawność zapisu odbieranych przez procesor danych, do pamięci programu wpisywane są one za pośrednictwem procesora. Angażuje to cały mikrokontroler i jest jego pierwszą pełną symulacją.

Na rysunku 18 jest przedstawiony przebieg wybranych sygnałów tej symulacji. Dane otrzymywane jako sygnał RX odczytywane przez moduł są analizowane, a część z nich jest wysyłana do pamięci. Widać również, że proces zapisywania danych do pamięci jest znacznie szybszy od ich odczytywania z sygnału RX.

Osobno, na rysunku 19 przedstawiono komunikat zwrotny modułu na wykrycie rozpoczęcia transmisji. Po odebraniu pierwszego bajtu za pomocą sygnału RX widać, że sygnał TX zaczął wysyłać komunikat. Sygnał "TX dane" przedstawia wysyłane bajty. Są to kolejno 0x68, 0x65, 0x78, 0x20, 0x72, 0x65, 0x63, 0x69, 0x76, 0x65, 0x65 i 0x0A. W kodzie ASCII te liczby reprezentują litery „hex received”[45]. Jest to zgodne z oczekiwaniami.



Rysunek 19. Ramka danych wysyłana przez mikrokontroler do użytkownika. Opracowanie własne.

5.4 Realizacja komunikacji między CPU a DPU

Komunikacja między procesorami jest niezbędnym elementem działania RSZ. Odbywa się ona inaczej w zależności od tego, który procesor wysyła dane. Nie ma możliwości transmisji danych bezpośrednio między dwoma DPU.

5.4.1 Komunikacja od CPU do DPU

CPU może wybrać DPU do których będzie jednocześnie wysyłać dane. Służy do tego specjalna lista zawierająca te DPU, które otrzymają dane po ich wysłaniu. W tym celu używa 3. adresów do których wysyłany jest numeru DPU:

- DPU_WRITE_SEL - Dodaje jedno DPU do listy
- DPU_WRITE_REMOVE - Usuwa jedno DPU z listy
- DPU_WRITE_CLEAR - Usuwa wszystkie DPU z listy

Po zdefiniowaniu tej listy można wysłać dane zapisując je do adresu DPU_WRITE_DATA. DPU może odebrać te dane po otrzymaniu przerwania INTER_MESS_FROM_CPU. Robi to za pomocą odczytania konkretnego adresu. Dane czekają w pamięci FIFO. Oznacza to, że zdefiniowana liczba bajtów może oczekiwać na odczytanie przez DPU tak długo, aż nie zostaną one odczytane. Zalecanym sposobem jest wysłanie w pierwszym wysyłanym bajcie długość wiadomości, a w kolejnych przesyłane dane, ale zdefiniowanie tej ramki jest zostawione programistę.

5.4.2 Komunikacja od DPU do CPU

Aby CPU mogło otrzymać dane od CPU musi wpierw ustawić rejestr CPU_ALLOW_MES_FROM_DPUS. Informuje on o gotowości do odbioru danych i jest czyszczony po ustawieniu nowego wektora przerwania. Dlatego nieużywane wektory przerwania też muszą być zaimplementowane przez programistę.

Aby wysłać dane z DPU wystarczy, że zapisze ono dane do rejestru send_data_to_cpu. Wtedy, jeżeli CPU nie obsługuje innego przerwania, otrzymuje te dane i może je odebrać po przez odczytanie adresu tego DPU. W przeciwnym przypadku DPU otrzymuje przerwanie INTER_MESS_TO_CPU_NOT_SENT, dzięki czemu wie, że dane nie zostały jeszcze odebrane przez CPU. W przypadku ponownego ustawienia rejestru CPU_ALLOW_MES_FROM_DPUS, wszystkie DPU które wcześniej chciały wysłać dane, otrzymują przerwanie INTER_MESS_ABLE_TO_SENT. Jeżeli kilka DPU na raz wyśle dane, pierwszeństwo ma DPU o najniższym numerze. CPU nie ma kolejki wchodzących danych, ale przechowywany jest jeden ostatnio wysłany bajt od każdego DPU.

5.5 Realizacja przerwań wewnętrznych. Opracowanie własne.

Przerwania są wykorzystywane do komunikacji pomiędzy DPU a CPU i interfejsami. Inne wektory przerwań są przeznaczone dla DPU, a inne dla CPU. DPU może odebrać szereg wektorów przerwań:

- INTER_READ_INTERFACE - Odczyt interfejsu. Interfejs informuje w ten sposób DPU, że odebrał nowe dane. Następuje to po każdym odebranym bajcie.

- INTER_MESS_FROM_CPU - Wiadomość od CPU. Oznacza to, że jest wiadomość, którą DPU może odczytać. Aby to zrobić wystarczy, że wybierze odpowiedni adres.
- INTER_MESS_TO_CPU_NOT_SENT - Nie odebranie wiadomości przez CPU.
- INTER_MESS_ABLE_TO_SENT - Ponowne umożliwienie wysyłania wiadomości przez CPU.

Jeżeli zdarzyłoby się jednocześnie wysłanie przerwań przez interfejs i CPU, interfejs ma pierwszeństwo. Jednak nie ma potrzeby informowania o tym CPU, ponieważ dane przez niego wysłane i tak zostaną odebrane nawet, jeśli wyśle kolejne przerwanie przed odebraniem poprzedniego, ze względu na użycie pamięci FIFO dla danych przesyłanych od CPU do DPU.

W przypadku CPU, pierwszeństwo mają wektory przerwań wysyłane przez moduł komunikacji z użytkownikiem, a po nim DPU o niższych numerach. Wektory przerwań obsługiwane przez CPU to:

- INTER_MESS_FROM_DPU_N - Oznacza przychodząą wiadomość od DPU o określonym numerze.
- INTER_MESS_FROM_USER - Oznacza oczekującą na odczytanie wiadomość od użytkownika.

Rozdział 6: Opracowanie architektury procesora

Istotnym elementem realizacji układu procesorowego w pracy jest wybór odpowiedniej architektury, dopasowanej do układów FPGA. Został on w całości samodzielnie zaimplementowany w języku opisu sprzętu, co oznacza realizację procesora wykonującego gotowy zestaw instrukcji, ale sposób ich wykonywania nie został dokładnie opisany przez producenta. Producent opisał jedynie efekty danej instrukcji, działanie rejestrów oraz wyróżnił kilka podstawowych bloków.

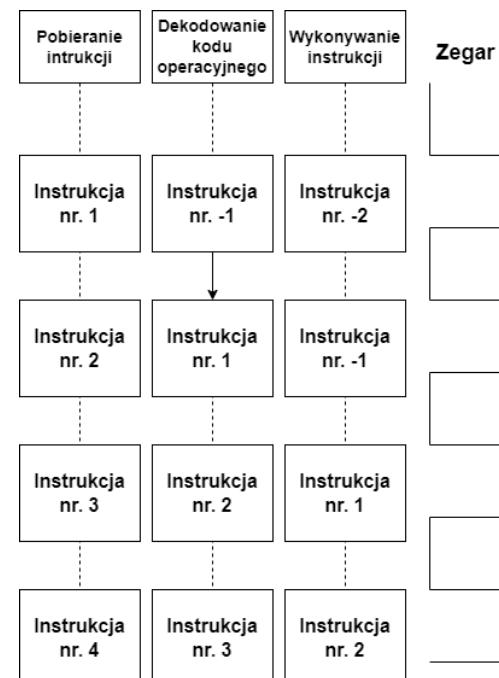
Mimo, że zestaw instrukcji nie jest twórczy, to przedstawiona tutaj mikroarchitektura jest twórcza i w jakimś stopniu unikatowa, ponieważ dwie implementacje tego samego procesora nigdy nie będą identyczne. W ramach tego rozdziału przedstawiono zarówno opisane przez producenta działanie procesora AVR, jak i jego realizacja w języku opisu sprzętu. Procesor jest w pełni kompatybilny ze kompilatorami AVR takimi jak AVR-GCC i XC8.

W trakcie pisania procesora pomocne okazały się jego inne implementacje [46][47].

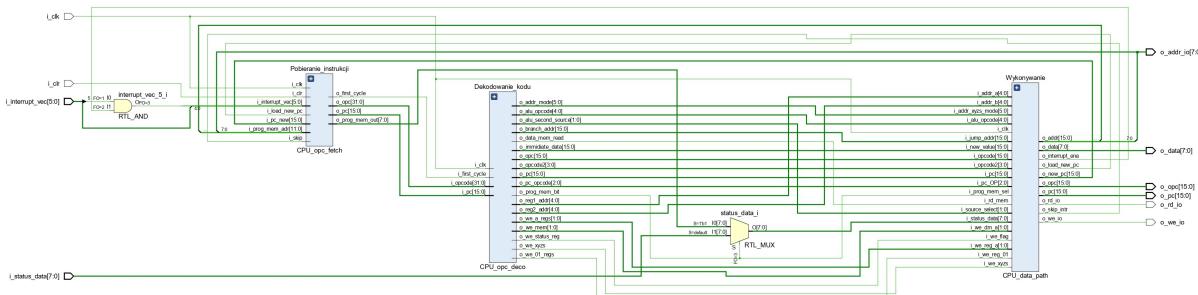
Architektura harwardzka pasuje do układów FPGA ze względu na możliwość wykorzystywania z szybkich wewnętrznych modułów pamięci. Oznacza to, że pamięć programu i danych są różne, co umożliwia większą elastyczność w projektowaniu. Korzystając z zalet architektury harwardzkiej jest możliwe wykorzystanie potokowości w celu zwiększenia wydajności procesora. Procesor można podzielić na 3 kolejno wykonywane etapy[48]:

- Pobieranie instrukcji odpowiada za jej odczytanie z pamięci programu.
 - Dekodowanie kodu operacyjnego oblicza sygnały sterujących, które będą potrzebne w kolejnym etapie.
 - Wykonanie instrukcji wykonuje wcześniej policzony kod operacyjny. Może przeprowadzać operacje matematyczne i obsługuje pamięć.

Ich przebieg w odniesieniu do cykli zegara jest przedstawiono na rysunku 20. Szczegółowa implementacja tego rozwiązania, a jednocześnie całego procesora, jest przedstawiona na rysunku 21. W kolejnych rozdziałach zostanie omówiona implementacja tych trzech modułów.



Rysunek 20. Etapy wykonywania instrukcji.
Opracowanie własne.



Rysunek 21. Implementacja potokowości procesora. Opracowanie własne.

6.1 Opracowanie modułu pobierania instrukcji

Sekwencer CPU, będący tematem niniejszej dyskusji, pełni kluczową funkcję w uruchamianiu rdzenia. Jest on odpowiedzialny za generowanie kodów operacyjnych, które następnie są dekodowane i wykonane.

W kontekście wykorzystywanej w pracy architektury harwardzkiej, charakteryzującej się osobnymi pamięciami dla programu i danych, można zainicjować pamięć programu w ramach tego etapu. Wewnętrzna pamięć współczesnych układów FPGA jest wystarczająca dla pełnej pamięci omawianego mikrokontrolera, dzięki czemu nie ma potrzeby dołączania pamięci zewnętrznej.

Ten etap głównie manipuluje licznikiem programu PC i generuje kody programu. PC jest wewnętrznym rejestrem tego modułu i podlega aktualizacji w każdym cyklu zegara oraz wskazuje adres pobieranej instrukcji.

Następna wartość PC w zależności od sygnałów wejściowych przybiera następujące wartości:

- 0 jeżeli zachodzi reset urządzenia.
 - Nie zmienia się, jeżeli układ oczekuje na drugi cykl zegara w celu wykonania instrukcji.
 - Przybiera wartość obliczaną na etapie wykonania, jeżeli sygnał `i_load_new_pc` pochodzący od modułu wykonywania jest równe ustawiony.
 - Zwiększa się o 2, gdy moduł wykryje jedną z instrukcji o podwójnej długości.

W większości przypadków nie zostaje spełniony żaden z tych warunków, a PC się wówczas inkrementuje, czyli zwiększa wartość o 1. Ze względu na użycie 16-bitowej pamięci programu, powoduje to odczytanie kolejnego adresu zawierającego nową instrukcję.

Sygnal o_skip_instruction pochodzi z etapu wykonawczego i unieważnia segmenty potoku. Używane jest to po podjęciu decyzji o skoku warunkowym. Z modułu wychodzi opóźniony o jeden cykl zegara sygnal o_pc, który jest zsynchronizowany z odczytaną i przesyłaną dalej instrukcją, dzięki czemu dalsze moduły mają dostep do prawidłowej wartości tego rejestru.

Większość instrukcji jest wykonywana w jeden cykl zegara, ale kilka z nich, tych które wymagają odczytywania danych z pamięci programu, wykonuje się w dwa cykle zegara. W celu oznaczenia tego, ile cykli zegara będzie potrzebnych, używany jest sygnał wait, a do oznaczania który cykl instrukcji jest aktualnie wykonywany, używany jest sygnał first_cycle.

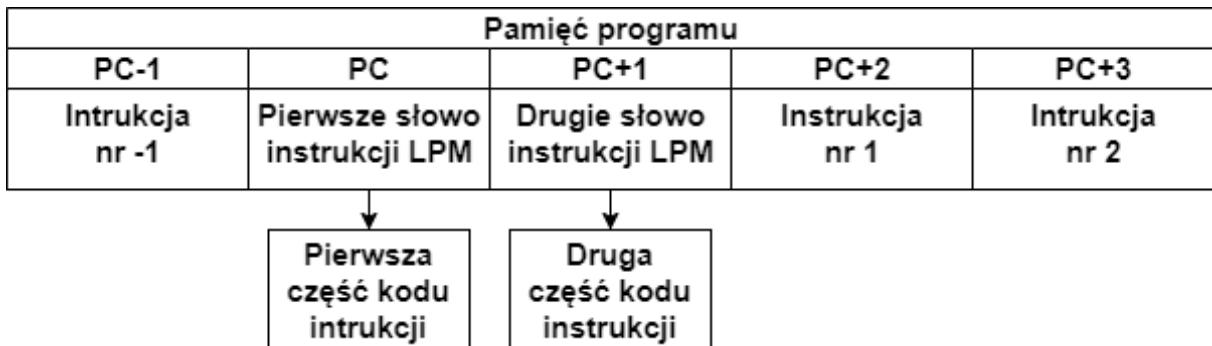
Ten etap jest używany do obsługiwanego przerwań generowanych w zewnętrznych urządzeniach. Przerwanie jest brane pod uwagę wtedy, gdy globalny bit aktywacji przerwań został wcześniej ustawiony. Najwyższy bit wskazuje, czy niższe bity zawierają poprawny numer przerwania. W takim przypadku kod instrukcji jest zastępowany przez kod przerwania.

6.1.1 Pamięć programu

Ten moduł ma wyłączny dostęp do pamięci programu. Działa on jako pamięć dwuportowa, umożliwiając jednocześnie odczytywanie i zapisywanie dwóch różnych lokalizacji pamięci. Do wykonywania instrukcji LPM potrzebne jest jednocześnie odczytywanie dwóch adresów pamięci. Ta instrukcja pobiera dane z pamięci programu podczas kontynuowania pobierania kolejnych instrukcji. Dzięki użyciu dwuportowej pamięci nie trzeba przerywać potoku, ponieważ dane z kolejnego adresu pamięci, są pobierane równolegle z odczytywaną instrukcją.

Instrukcja została zaimplementowana jako udostępniony przez producenta generator pamięci blokowej RAM. Pobiera lub zapisuje dane do dwóch niezależnie adresowanych 16-bitowych komórek pamięci. Ma długość 32kB, ale można ją powiększyć. Dane są zwracane po jednym cyklu zegara.

Na rysunku 22 przedstawione jest odczytywanie zawartości pamięci programu dla jednej z długich instrukcji. Zwykle dodatkowa zawartość kodu instrukcji jest ignorowana przez dalsze moduły, ale zawsze jest ona im wysyłana, ponieważ to one zajmują się jej interpretacją. Dzięki temu nie trzeba czekać dodatkowego cyklu zegara na niektóre instrukcje.



Rysunek 22. Odczytywanie czterobitowej instrukcji. Opracowanie własne.

Możliwy jest reset pamięci programu poprzez ustawienie sygnału `i_rst`. Wskaźnik PC się wówczas zeruje, a zawartość pamięci się wymazuje aż do ponownego zapisania. W ten sposób przeprowadzany jest reset całego procesora.

6.1.2 Realizacja przerwań

Przerwania są mechanizmem pozwalającym na przerwanie normalnego działania programu w celu obsługi zdarzeń zewnętrznych lub wewnętrznych. Początek każdego programu zawiera wektory przerwań, które są wskaźnikami do zaimplementowanych przez programistę funkcji. Przerwanie wpierw zapisuje aktualną wartość wskaźnika programu do stosu, po czym zmienia ją na wartość zawartą w sygnale przerwania. Dzięki temu może zostać wykonana instrukcja skoku do odpowiedniej funkcji.[48] Czas wykonywania przerwania od jego otrzymania do wykonania pierwszej instrukcji to cztery cykle instrukcji ze względu na dwukrotne przeprowadzenie instrukcji skoku. [48]

Specjalny sygnał wchodzący do modułu przekazuje wartość 7. bitu rejestru ustawień. Jeżeli jest on ustawiony, a wchodzący do modułu wektor przerwań jest różny od zera, to kod operacyjny zmienia swoją wartość. Dolne 5 bitów kodu operacyjnego to wektor przerwania, a górne oznaczają instrukcje wykonania przerwania. Aktualnie wykonywane instrukcje zostają wykonane do końca, a inkrementacja PC zostaje zatrzymana. Dzięki temu wartość PC z momentu dostania przerwania może zostać zapisana do stosu. Umożliwia to późniejszy powrót z przerwania do wcześniejszej wykonywanej funkcji.

6.2 Opracowanie modułu dekodowania kodu operacyjnego

Zadaniem tego modułu jest przekształcenie odczytanej instrukcji na szereg sygnałów, które dokładnie określają akcje wykonywane przez moduł wykonywania. Instrukcja jest przekazywana przez sygnał `i_opcode`, a jej ewentualny drugi bajt za pomocą `i_opc_h`.

Moduł generuje wiele sygnałów wyjściowych, ale większość instrukcji używa zaledwie kilka z nich. Ich działanie na moduł wykonywania jest opisane w rozdziale section 6.4. Nie używany przez daną instrukcję wyjście przybiera wartości domyślne, które dla każdego sygnału wynoszą 0. Oznacza to, że:

- Domyślny tryb adresowania jest ustalony na adresowanie natychmiastowe i jest podawane przez sygnał wyjściowy `o_addr_mode`.
- W prawie każdej instrukcji adres rejestrów, a są one w niej zdefiniowane, zawiera się w tych samych bitach instrukcji. Z tego względu nie ma potrzeby definiowania ich za każdym razem, kiedy piszemy do portów `o_reg1_addr` i `o_reg2_addr`. Czasami jest on skrócony, w takim przypadku wystarczy zaznaczyć nadpisanie najważniejszego bitu adresu.
- Domyślnie drugie źródło danych dla modułu ALU to drugie wyjście rejestru, co jest oznaczone poprzez dwubitowe wyjście `o_alu_second_source`.

6.3 Opracowanie modułu dekodowania kodu operacyjnego

Istotnym dla działania instrukcji jest opis sposobów adresowania, który ten moduł definiuje. Decyduje on o wartości wskaźników danych. Informuje o nich sygnał wyjściowy `o_addr_mode`. Instrukcja wybiera źródło adresu i operację wykonywaną na nim. Jest możliwe 5 wskaźników danych: rejesty X, Y, Z, SP oraz sygnał `o_addr_imm`, czyli adres zawarty w drugim słowie instrukcji. Na adresie może zostać wykonana jedna z sześciu poniższych operacji:

- brak zmiany wartości wskaźnika,
- zwiększenie jego wartości o 1 lub 2 po wykonaniu instrukcji,
- zmniejszenie jego wartości o 1, lub 2 przed wykonaniem instrukcji,
- dodanie do wskaźnika liczby od 0 do 64 zawartej w wykonywanej instrukcji,

Sposób wykonania tych operacji jest omówiony w rozdziale 6.4.2.1. Nie wszystkie kombinacje źródła danych i operacji na nich mogą wystąpić. Z tego względu niepotrzebne są wszystkie możliwe kody adresowania wchodzące do modułu wykonywania, a wystarczy uwzględnić wykonywane przez procesory AVR instrukcje. Są możliwe do użycia następujące kombinacje adresów danych:

- niezmieniona wartość natychmiastowa przekazywana przez instrukcję
- niezmieniona wartość rejestrów X, Y lub Z
- postinkrementacja wartości rejestrów X, Y lub Z
- predekrementacja wartości rejestrów X, Y lub Z
- dodanie liczby zawartej w instrukcji do wartości rejestrów X, Y lub Z
- zmienienie wartości wskaźnika stosu o 1 lub 2, przed lub po wykonaniu instrukcji

6.3.1 Działanie instrukcji

Większość instrukcji działa poprzez ustawienie odpowiadającego im kodu operacji ALU na porcie o_alu_opcode. Zapis rejestrów odbywa się poprzez sygnał o_we_a_regs, a zapis rejestru ustawień poprzez o_we_status. Instrukcje można podzielić na kilka kategorii na podstawie podobieństwa wykonania. Nie jest to podział wiążący, ale ułatwia ich zrozumienie i implementację.

6.3.2 Transfer danych

Prawidłowo zrealizowany transfer danych pomiędzy adresami pamięci i rejestrami jest niezbędny dla sprawnego działania procesora. Używany do programowania kompilator GCC w zdecydowanej większości używa adresowania bezpośredniego lub z przemieszczeniem, a rzadziej adresowania przez rejesty. Ewentualne dodatkowe sposoby adresowania nie są przez niego praktycznie wykorzystywane [49]. Z tego powodu to te instrukcje są używane przez architekturę AVR a ze względu na ich gęstość w kodzie programu, muszą działać w ciągu jednego cyklu zegara.

Sposób uzyskania informacji na temat funkcjonalności poszczególnych bitów jest pokazany na przykładzie instrukcji transferu danych. Poniżej znajdują się instrukcje, których pierwsze 6 bitów to 100100, a 7. bajt równy 0.

Większość z nich to warianty instrukcji LD i LDD. Ładują one dane z przestrzeni danych wskazywanych przez wskaźnik do rejestrów. Wskaźnik może przybierać różne wartości oparte o rejesty X, Y i Z. Instrukcja LPM wczytuje bajt wskazywany przez rejestr Z, a POP odczytuje bajt ze stosu do rejestrów.

- LD X,r 1001 00-0d dddd 1100
- LD X+,r 1001 00-0d dddd 1101
- LD -X,r 1001 00-0d dddd 1110
- LD Y+,r 1001 00-0d dddd 1001
- LD -Y,r 1001 00-0d dddd 1010
- LD Z+,r 1001 00-0d dddd 0001
- LD -Z,r 1001 00-0d dddd 0010
- LPM r, Z 1001 00-0d dddd 0100
- LPM r, Z+ 1001 00-0d dddd 0101

- POP r 1001 00-0d dddd 1111

Poniższe instrukcje również mają pierwsze 6 bitów równe 100100, ale 7. bajt jest równy 1. Służą one do zapisania danych.

- ST X,r 1001 00-1r rrrr 1100
- ST X+,r 1001 00-1r rrrr 1101
- ST -X,r 1001 00-1r rrrr 1110
- ST Y+,r 1001 00-1r rrrr 1001
- ST -Y,r 1001 00-1r rrrr 1010
- ST Z+,r 1001 00-1r rrrr 0001
- ST -Z,r 1001 00-1r rrrr 0010
- PUSH r 1001 00-1r rrrr 1111

Instrukcje ST i STD zapisują dane z rejestru do przestrzeni danych wskazywanych przez wskaźnik. Wskaźnik może przybierać takie wartości, jak w przypadku instrukcji LD, stąd, podobnie jak poprzednio, mnogość opcji wykonania instrukcji. Z kolei instrukcja PUSH zapisuje bajt do stosu.

Po takim przedstawieniu tych instrukcji łatwo zauważać ich części wspólne oraz to, jakie bity są używane w celu wyboru adresowania. Wszystkie powyższe instrukcje natychmiastowego przesunięcia zaczynają się od 100100. To właśnie pierwsze 6 bitów jest najczęściej używane do identyfikacji instrukcji. Na podstawie kolejnego bitu układ decyduje, czy ma do czynienia z instrukcjami odczytania, czy zapisania. To od tego zależy wartość portu o_alu_opcode oraz kilku innych. Ostatnie 4 bity definiują, jaki ma być źródło rejestru. Różnią się więc wartościami na wyjściu o_addr_mode, które decyduje o sposobie adresowania oraz wyjściem o_prog_mem_sel służącym do wyboru źródła danych.

Pomimo, że poniższe instrukcje różnią się jedynie adresowaniem od wyżej wymienionych, należy je osobno wykryć, po czym zastosować operacje podobne do poprzednich operacji. Pokazuje to, że nie zawsze zbliżone instrukcje mają podobny kod operacyjny.

- LD Z,r 1000 000d dddd 0000
- LD Y,r 1000 000d dddd 1000
- LDD Y+q,r 10q0 qq0d dddd 1qqq
- LDD Z+q,r 10q0 qq0d dddd 0qqq
- ST Y,r 1000 001r rrrr 1000
- ST Z,r 1000 001r rrrr 0000
- STD Y+q,r 10q0 qq1r rrrr 1qqq
- STD Z+q,r 10q0 qq1r rrrr 0qqq

- LPM 1001 0101 1100 1000
- MOV r,r 0010 11rd dddd rrrr - Kopiuje dane pomiędzy dwoma rejestrami.
- LDI 1110 KKKK dddd KKKK - Ładuje do rejestru wartość podaną w instrukcji.
- MOVW r, r 0000 0001 dddd rrrr - Kopiuje dane między dwiema parami rejestrów.
- OUT P,r 1011 1PPr rrrr PPPP - Umożliwia zapis rejestrów zewnętrznych.
- IN r, P 1011 OPPd dddd PPPP - Umożliwia odczyt rejestrów zewnętrznych.

Te instrukcje są zupełnie inaczej reprezentowane, ale kod instrukcji docierający do modułu wykonania może być dla nich taki sam ze względu na podaną funkcjonalność. Dlatego ten etap jest niezbędny. Gdyby każda instrukcja była odróżnialna od wszystkich innych wyłącznie na podstawie pierwszych kilku bitów, musiałyby być one znacznie dłuższe. Zmieszczenie tak wielu różnych instrukcji w 16 bitów w sposób, który optymalizuje działanie procesora, jest na tyle trudne, że procesor używa gotowej architektury zestawu instrukcji.

6.3.3 Instrukcje zmiany wskaźnika programu

Manipulacja wskaźnikiem programu PC jest niezbędna do powtarzania jakiejś części programu lub jego warunkowego wykonania. Dlatego jest wiele instrukcji umożliwiających to na wiele różnych sposobów. O tym, w jaki sposób ma się zmienić PC w module wykonywania, decyduje wyjście `o_pc_opcode`, które określa kod manipulacji wskaźnikiem programu, jednak większość z tych instrukcji używają `o_alu_opcode` w celu obliczenia nowego adresu. Można je podzielić na dodatkowe podkategorie w zależności od ich funkcjonalności.

6.3.3.1 Instrukcje skoku

Instrukcje skoku powodują bezwarunkowy skok do nowego adresu pamięci programu. Nie zapisują w żaden sposób aktualnej wartości adresu programu. Pozwalają one na skok, ale bez prostej możliwości powrotu, co utrudnia wykorzystanie tych instrukcji przy wykonywaniu funkcji programu. Różnią się one sposobami adresowania. Są to:

- JMP 1001 010h hhhh 110h - Adres skoku jest podawany w kolejnym słowie instrukcji. Jako jedyna z nich ma dwa słowa
- RJMP 1100 LLLL LLLL LLLL - Nowy wskaźnik programu jest większy od aktualnego o podawaną przez instrukcję wartość.
- IJMP 1001 0100 0000 1001 - Adres skoku jest podany w rejestrze Z. Pozwala na skok do 64 tysięcy słów instrukcji.

Do realizacji skoków umożliwiających późniejszy powrót do kolejnej instrukcji używa się funkcji CALL. Mają one następujące, analogiczne do instrukcji JMP warianty:

- RCALL 1101 LLLL LLLL LLLL
- ICALL 1001 0101 0000 1001

Każda z tych instrukcji zapisuje do stosu aktualną 16-bitową wartość adresu programu. W tym celu musi odjąć od wartości wskaźnika stosu 2, ponieważ adres zajmuje dwa bajty.

W celu powrotu z wywołania używane są instrukcje powrotu:

- RET 1001 0101 0000 1000
- RETI 1001 0101 0001 1000

Obie pobierają wartość ze stosu, po czym ustawiają ją jako adres programu, jednak RETI dodatkowo ustawia globalny bit przerwania.

6.3.3.2 Instrukcje skoku warunkowego

Dla ocenienia, czy zaszło jakieś zdarzenie, co jest niezbędnym elementem każdego programu, niezbędne są funkcje umożliwiające skok warunkowy. Służą one do warunkowej zmiany wskaźnika programu. Są one bardzo pomocne zważywszy na to, że następna instrukcja może być skokiem do wywołania. Są to:

- CPSE 0001 00rd dddd rrrr
Pomija kolejną instrukcję, jeżeli dwa rejestrów są sobie równe.
- SBIC r, b 1001 1001 pppp psss
Pomija kolejną instrukcję, jeżeli określony bit w rejestrze zewnętrznym jest czysty.
- SBIS r, b 1001 1011 pppp psss
Pomija kolejną instrukcję, jeżeli określony bit w rejestrze zewnętrznym jest ustawiony.
- SBRC r, b 1111 110r rrrr 0sss
Pomija kolejną instrukcję, jeżeli określony bit w rejestrze jest czysty.
- SBRS r, b 1111 111r rrrr 0sss
Pomija kolejną instrukcję, jeżeli określony bit w rejestrze jest ustawiony.
- BRBS s,k 1111 00kk kkkk ksss
Jeżeli określony przez nią bit rejestrów ustawień jest ustawiony, to zachodzi skok o określana przez nią wartość przyjmującą wartości z zakresu od -64 do 63.
- BRBC s,k 1111 01kk kkkk ksss
Jeżeli określony przez nią bit rejestrów ustawień jest czysty, to zachodzi skok o określana przez nią wartość przyjmującą wartości z zakresu od -64 do 63.

Instrukcje BRBS i BRBC są wyjątkowe, ponieważ umożliwiają skok warunkowy o większą wartość niż jedna instrukcja. Mają one wiele wariantów. Wszystkie mają identyczną funkcjonalność, a różnią się tylko tym, który bit biorą pod uwagę. Mają one różne nazwy, ale każda z nich jest zaimplementowana, jeżeli zaimplementuje się tylko BRBC i BRBS.

6.3.3.3 Instrukcje porównawcze

Instrukcje porównawcze służą do zapisu rejestrów ustawień bez zapisywania wyniku operacji. ALU dostaje ten sam kod operacji, co w przypadku operacji odejmowania, ale nie zmienia się wartości żadnego rejestrów z wyjątkiem rejestrów ustawień, do którego zwarcane są określone bity.

Są one przydatne w połączeniu z instrukcjami skoku warunkowego, które decydują o wykonaniu skoku na podstawie wcześniej ustawionych bitów rejestru.

- CP r,r 0001 01rd dddd rrrr Porównuje ze sobą dwa rejesty.
- CPI d,M 0011 KKKK dddd KKKK Porównuje ze sobą dwa rejesty odejmując bit reszty.
- CPC r,r 0000 01rd dddd rrrr Porównuje dwa rejesty i resztę.

6.3.4 Instrukcje arytmetyczne i logiczne

Większość instrukcji arytmetycznych działa w ten sam sposób. Zmienia się jedynie kilka sygnałów, a pozostałe są nieużywane. Są to:

- Sygnał o_alu_opcode określający wykonywaną przez ALU operację.
- Sygnał o_we_status_reg umożliwiający zapis bitów rejestru ustawień poprzez ustawienie bitu.
- Sygnał o_we_a_regs umożliwia zapis do wyniku do rejestru.
- Sygnały o_addr_a i o_addr_b pozwalają na wybranie rejestrów używanych przez ALU.
- Sygnał o_alu_second_source pozwala na wybranie przez ALU wartości z sygnału o_immediate_data

Te instrukcje można podzielić na kilka mniejszych kategorii:

Suma lub różnica wartości natychmiastowej i rejestru

- SBIW 1001 0111 KKdd KKKK
- ADIW 1001 0110 KKdd KKKK

Te instrukcje mogą adresować tylko 4 górne pary rejestrów. Przekazują one wartość od 0 do 63. Jako jedyne wykonują operacje arytmetyczne na parach rejestrów, a przez to ustawiają oba bity sygnału o_we_a_regs.

Instrukcje arytmetyczne z adresowaniem bezpośrednim:

- SUBI r, M 0101 KKKK dddd KKKK Odejmuje przekazywaną wartość od rejestru.
- SBCI r, M 0100 KKKK dddd KKKK Odejmuje od rejestru przekazywaną wartość i resztę.

Pozostałe warianty dodawania i odejmowania:

- ADD r,r 0000 11rd dddd rrrr Sumuje dwa rejesty.
- ADC r,r 0001 11rd dddd rrrr Sumuje dwa rejesty i bit reszty.
- SUB r, r 0001 10rd dddd rrrr Odejmuje od siebie dwa rejesty.
- SBC r,r 0000 10rd dddd rrrr Odejmuje od siebie dwa rejesty i resztę.

- COM r 1001 010r rrrr 0000 - Odejmuje zawartość rejestru od 255.
- NEG r 1001 010r rrrr 0001 - Odejmuje zawartość rejestru od 0.
- DEC r 1001 010r rrrr 1010 - Dekrementuje wartość rejestru.
- INC r 1001 010d dddd 0011 - Inkrementuje wartość rejestru.

Operacje mnożenia Jest wiele różnych możliwości mnożenia. Ze względu na zasady mnożenia, iloczyn dwóch 8-bitowych liczb jest 16-bitowy. Wszystkie te instrukcje zwracają wynik do dwóch pierwszych rejestrów R0 i R1.

- MUL r, r 1001 11rd dddd rrrr
Mnoży dwie liczby naturalne umieszczone w rejestrach od R0 do R31.
- MULS r, r 0000 0010 dddd rrrr
Mnoży dwie liczby całkowite, które mogą być umieszczone w rejestrach od R16 do R31.
- MULSU r, r 0000 0011 0ddd Orrr
Mnoży liczbę naturalną i całkowitą, które znajdują się w rejestrach od R16 do R23.
- FMUL r, r 0000 0011 0ddd 1rrr
Mnoży dodatni ułamek i liczbę naturalną. Liczby mogą być umieszczone tylko w rejestrach od R16 do R23.
- FMULS r, r 0000 0011 1ddd 0rrr
Mnoży ułamek i liczbę całkowitą. Liczby mogą być umieszczone tylko w rejestrach od R16 do R23.
- FMULSU r, r 0000 0011 1ddd 1rrr
Mnoży dodatni ułamek i liczbę całkowitą. Liczby mogą być umieszczone tylko w rejestrach od R16 do R23.

W powyższych instrukcjach ułamki są zdefiniowane jako liczby z jedną cyfrą binarną przed przecinkiem i siedmioma po przecinku. Taki format cyfr jest często używany w przetwarzaniu sygnałów. Ze względu na właściwości mnożenia, wynik ma dwie liczby przed przecinkiem i 14 po przecinku, a do jego obliczenia wymagane jest zastosowanie przesunięcia binarnego w lewo.

Tylko do nich jest używany sygnał o_we_01_regs. Pozwala on na zapisanie wyników mnożenia do pierwszej pary rejestrów.

Operacje logiczne Są trzy operacje logiczne zaimplementowane sprzętowo: koniunkcja, alternatywa i alternatywa rozłączna. Ze względu na dwie możliwość wykonywania operacji logicznej na niezapisanej wcześniej do pamięci liczbie, jest 5 instrukcji:

- AND r, r 0010 00rd dddd rrrr
Przeprowadza koniunkcję na dwóch rejestrach.
- ANDI r, K 0111 KKKK dddd KKKK
Przeprowadza koniunkcję na rejestrze i natychmiastowej wartości.

- EOR r,r 0010 01rd dddd rrrr
Przeprowadza alternatywą rozłączną.
- OR r, r 0010 10rd dddd rrrr
Przeprowadza alternatywę pomiędzy dwoma rejestrami.
- ORI r, K 0110 KKKK dddd KKKK
Przeprowadza alternatywę na rejestrze i natychmiastowej wartości.

Ustawianie i czyszczenie rejestrów:

- SBR r, M 0110 KKKK dddd KKKK
Ustawia wybrane bity w jednym z górnych 16 rejestrów.
- SER d 1110 1111 dddd 1111
Ustawia wszystkie bity w jednym z górnych 16 rejestrów.
- CLR r 0010 01rd dddd rrrr
Czyści wszystkie bity w rejestrze. Jest równoznaczne z instrukcją EOR wykonaną dla dwóch tych samych rejestrów.
- CBR 0111 KKKK dddd KKKK
Czyści wybrane bity w rejestrze.
- TST 0010 00rd dddd rrrr
Sprawdza, czy w rejestrze jest zero lub liczba ujemna. Jest równoznaczne z koniunkcją bitową wykonaną dla dwóch tych samych rejestrów.

6.3.5 Instrukcje manipulujące bitami

Te instrukcje służą do działań na poszczególnych bitach.

- SBI p, b 1001 1010 pppp psss - Ustawia bit w rejestrze zewnętrznym
- CBI p, b 1001 1000 pppp psss - Czyści bit w rejestrze zewnętrznym.
- LSL r 0000 11rd dddd rrrr- Przeprowadza przesunięcie bitowe zawartości dowolnego rejestrów w lewo, a najbardziej prawy bit jest czyszczony. Może być używana do pomnożenia liczby przez 2.
- LSR r 1001 010r rrrr 0110 - Przeprowadza przesunięcie bitowe zawartości do wolnego rejestrów w prawo, a najbardziej lewy bit jest czyszczony. Może być używana do podzielenia wartości rejestrów przez 2.
- ROL r 0001 11rd dddd rrrr - Rotacja binarną w lewo przez bit reszty. Przesuwa bity rejestrów w lewo. Bit najbardziej na prawo przyjmuje wartość bitu reszty, po czym reszta przyjmuje wartość najbardziej znaczącego bitu pierwotnego rejestrów.
- ROR r 1001 010r rrrr 0111 - Rotacja binarną w prawo przez bit reszty. Polega na tym samym, co instrukcja ROL, ale w prawo.
- ASR r 1001 010r rrrr 0101 - Przesuwa bity rejestrów o jeden bit w prawo.

- SWAP r 1001 010r rrrr 0010 - Zamienia ze sobą górne 4 bity z dolnymi 4 bitami wybranego rejestru.
- BSET s 1001 0100 0SSS 1000 - Ustawia wybrany bit rejestru ustawień.
- BCLR s 1001 0100 1SSS 1000 - Czyści wybrany bit rejestru ustawień.
- BST r,b 1111 101d dddd 0sss - Kopiuje wybrany bit z rejestru ogólnego do rejestru ustawień.
- BLD r, b 1111 100d dddd 0sss - Zapisuje bit z rejestru ustawień do wybranej pozycji rejestru ogólnego.

Dodatkowo są osobne instrukcje ustawiania i czyszczenia konkretnych bitów w rejestrze ustawień. Ze względu na to, że jest to 8 bitów, istnieje 16 dedykowanych instrukcji, których przytoczenie tutaj mija się z celem, ze względu na to, że sprzętowo są jedynie wariantami instrukcji BSET i BCLR.

6.4 Opracowanie modułu wykonywania instrukcji

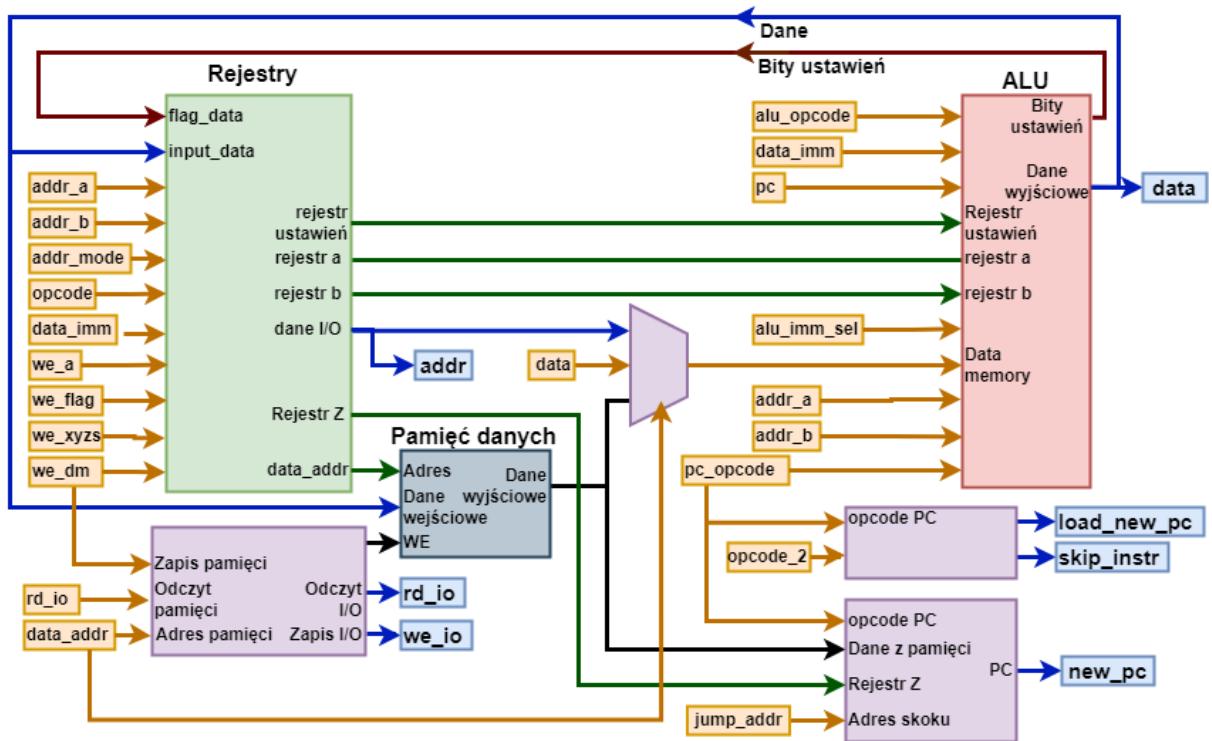
Zadaniem modułu wykonywania instrukcji jest wykonanie instrukcji, która wcześniej została podzielona na konkretne etapy. Składa się on z 3 głównych elementów:

- Rejestrów- to w tym module znajdują się wszystkie 32 rejesty ogólne procesora, które są głównym źródłem danych odczytywanych przez ALU. Zawiera on też rejestr ustawień i wskaźnik stosu.
- Układ arytmetyczny ALU. Odpowiada za wykonywanie wszystkich obliczeń i operacji logicznych.
- Pamięć danych. Stosunkowo duża, adresowana pamięć zawierająca rzadziej używane dane oraz stos.

Schemat tego modułu znajduje się na rysunku 23. Dla zwiększenia czytelności pomija on kilka rzadko używanych sygnałów. Na żółto zaznaczone są sygnały wejściowe, na niebiesko wyjściowe, a na fioletowo jest logika modułu. Dane mogą być wysyłane do rejestrów i pamięci danych tylko za pośrednictwem ALU, ale nie ma on wpływu na to, czy zostaną one zapisane ani na to, do jakiego adresu trafią.

Logika modułu jest odpowiedzialna za dwie funkcje:

- Obsługę wartości zależnych od adresu danych. Może on wskazywać rejesty, dane zewnętrzne lub pamięci. Dlatego w zależności od wartości sygnału wejściowego *i_data_addr*, dane wchodzące do ALU mogą pochodzić z rejestrów, danych przychodzących do procesora lub pamięci danych, a zapis danych może następować albo do portu zewnętrznego, albo do pamięci danych. Analogicznie działa odczyt danych.
- Decydowaniu o wykonaniu skoku. O tym, czy kolejna instrukcja ma zostać pominięta, a wskaźnik PC ma zmienić swoją wartość, decydują sygnały wejściowe *i_PC_opcode* i *i_opcode_2*. Wartość nowego wskaźnika PC może przybrać jedną z czterech opcji:
 - Wartość pobraną z pamięci danych



Rysunek 23. Opracowana architektura modułu wykonywania instrukcji. Opracowanie własne.

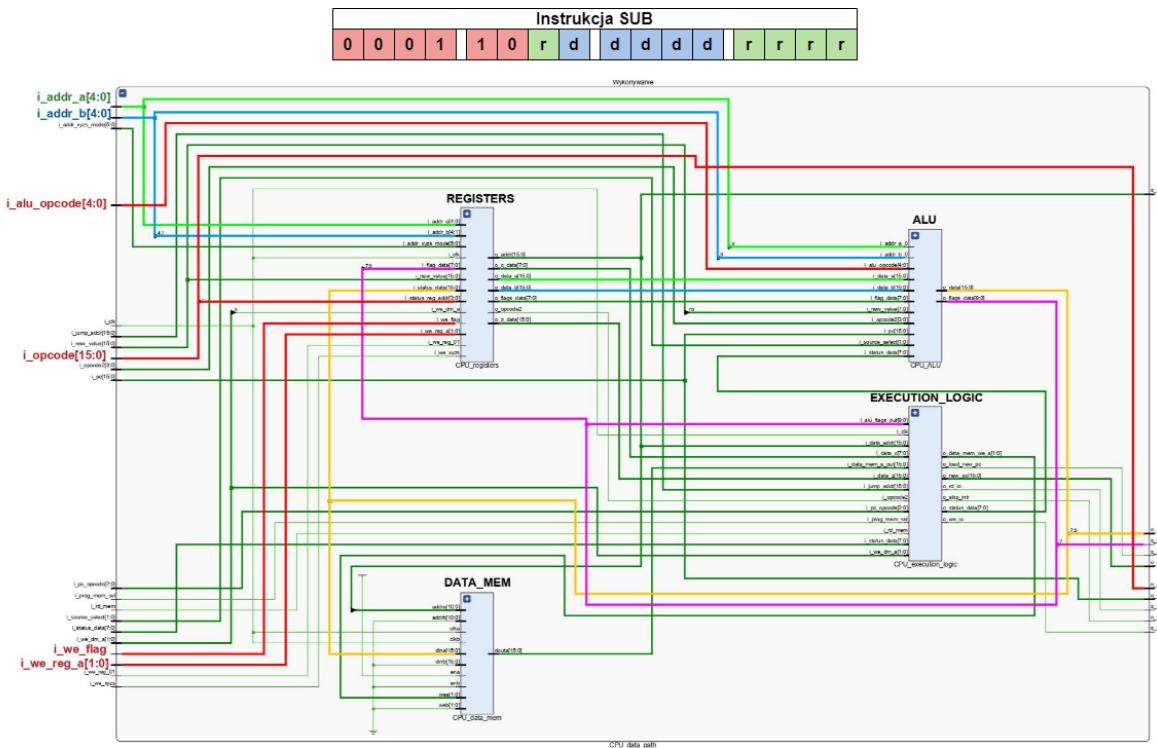
- Wartość rejestru Z
- Wartość skoku obliczaną przez dekoder instrukcji i przekazywaną przez sygnał wejściowy `i_jump_addr`

Na rysunku 24 przedstawiony jest schemat implementacji układu. Kolorami zaznaczono przebieg danych wejściowych oraz ich znaczenie. Moduł dekodowania kodu operacyjnego rozdziela instrukcje na mniejsze, zaznaczone na rysunku elementy. Są one następnie przesyłane do odpowiednich modułów. Rysunek to dokładna implementacja schematu 23, między którymi jedyną różnicą to umieszczenie logiki układu w jednym module, a nie tak jak wcześniej, w kilku fioletowych modułach.

Na czerwono zaznaczono sygnały sterujące określające rodzaj wykonywanych operacji, wraz z bitami instrukcji na podstawie których je określono. Na zielono przedstawiono adres pierwszego rejestru R, a na niebiesko drugiego rejestru D, a odpowiednimi kolorami zaznaczono bity instrukcji zawierające te adresy. Na żółto zaznaczono dane wyjściowe, a na różowo dane rejestrów ustawień.

6.4.1 Opracowanie układu arytmetycznego

Układ arytmetyczny jest najprostszym pod względem implementacji urządzeniem. Ma za zadanie wykonywać operacje arytmetyczne, zwracać ich wynik oraz aktualizować rejestr ustawień. Nie posiada zegara wejściowego, ponieważ wszystkie operacje wykonuje po pojawienniu się nowych danych na jego sygnały wejściowe, a wynik jest zapisywany do pamięci i rejestrów w kolejnym cyklu zegara. Schemat modułu jest przedstawiony na rysunku 25. Składa się on z trzech kolejnych etapów:



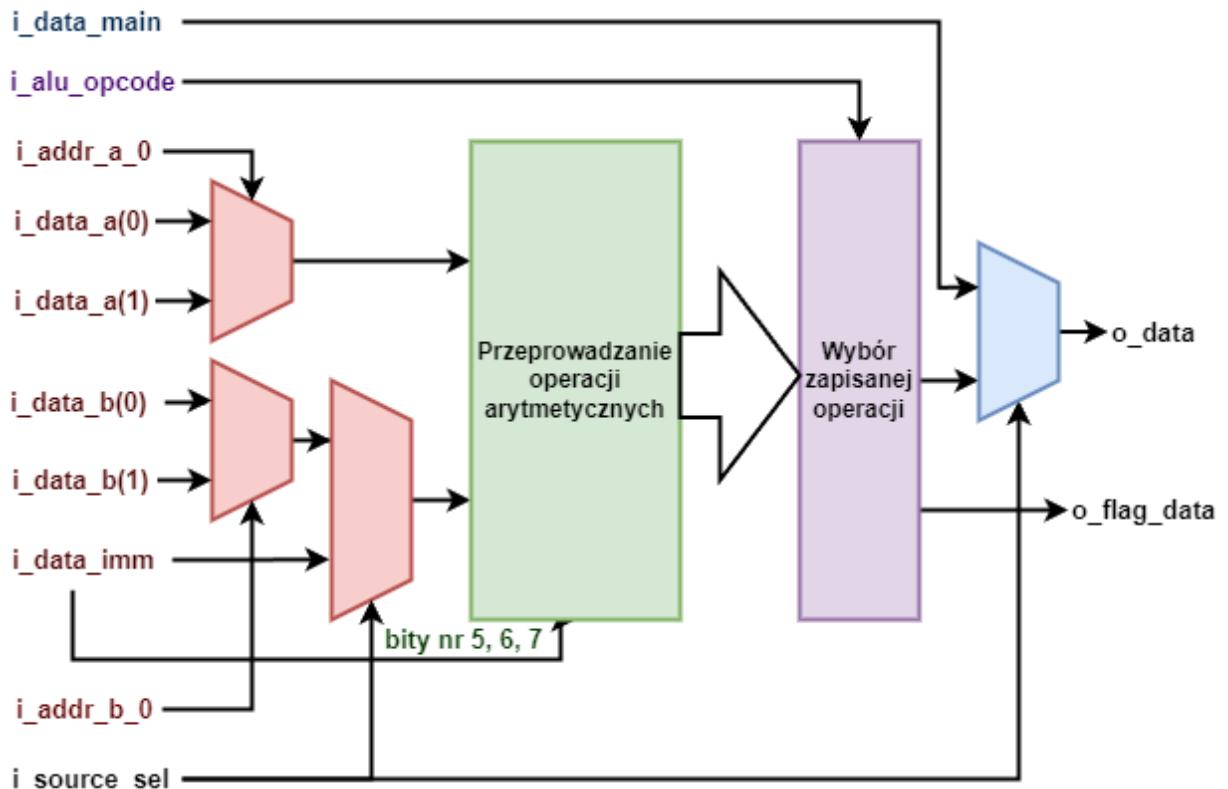
Rysunek 24. Wykonywanie instrukcji na przykładzie instrukcji SUB. Opracowanie własne.

- Multipleksery danych wejściowych. Pierwszym źródłem danych jest zawsze rejestr adresowany przez sygnał *i_addr_a*. Drugim źródłem danych może być rejestr adresowany przez *i_addr_b* lub Wybierany jest górny lub dolny bajt rejestrów wejściowych. Może on też wybrać dane z sygnału wejściowego *i_new_value* jeżeli *i_source_select* ma odpowiednią wartość. W efekcie z pięciu bajtów wejściowych wybiera 2, które są wykorzystywane w dalszych operacjach.
- Przeprowadzanie operacji arytmetycznych. To tu obliczane są wszystkie możliwe do przeprowadzania operacje arytmetyczne i logiczne. 3 górne bity sygnału *i_new_value* są wykorzystywane podczas wykonywania operacji mnożenia i określają, która dokładnie operacja mnożenia jest wykonywana.
- Wybór zapisanej operacji. W zależności od kodu operacji ALU, wybierana jest jedna operacja której wynik może zostać zapisany do danych wyjściowych. To na tym etapie liczone są nowe wartości bitów ustawień.
- Multiplekser danych wyjściowych. Instrukcje obsługujące pamięć danych wymagają, aby ALU tylko przepisywał dane wejściowe bez ich modyfikowania. Dlatego przy wybraniu odpowiedniego kodu źródła danych przez dekoder instrukcji, sygnał wyjściowy ALU przepisuje sygnał wejściowy *i_data_mem*, pozwalając na szybkie odczytanie danych pamięci.

Wyjście danych jest 16-bitowe, a większość operacji wykorzystuje jedynie 8 bitów. Dlatego dolny bajt jest przepisywany, co ułatwia późniejsze zapisanie danych. Od tego przebiegu są dwa wyjątki:

- Instrukcje ADIW i SUBI wykonują działania na parze rejestrów. Dlatego używają one danych z *i_data_a* bezpośrednio, z pominięciem multipleksera.

- Instrukcje SBIC, SBIS i CPSE zmieniają wartość PC wtedy, gdy spełniony jest dany warunek wartości rejestrów. W pierwszym cyklu zegara, po odczytaniu instrukcji, zachodzi sprawdzenie warunku. Informacja na temat tego, czy został on spełniony, jest przekazywana przez sygnał skip_SKIS i zapisywana. Dzięki temu w kolejnym cyklu zegara wiadomo, jaką wartość ma przybrać sygnał o_skip_intr.



Rysunek 25. Schemat architektury modułu ALU. Opracowanie własne.

6.4.2 Opracowanie rejestrów

Rejestry służą do przechowywania pojedynczych łatwo dostępnych bajtów danych przez procesor. Można wyróżnić rejesty ogólne, które mogą przechowywać dowolne dane, oraz rejesty specjalne, czyli rejesty wskaźnika stosu i ustawień.

6.4.2.1 Rejestry ogólne

Architektura zestawu instrukcji AVR[48] definiuje 32 8-bitowe rejesty przedstawione na rysunku 26. Są one adresowane od zera do 31 zgodnie z ich nazwami. W celu szybszego zapisywania lub odczytywania 16-bitowych danych są one podzielone na pary tak, jak na rysunku. Jest możliwość zapisywania lub odczytywania pojedynczego rejestru lub jego pary.

6 ostatnich rejestrów mają dodatkowe funkcjonalności, są one używane jako 16-bitowe wskaźniki pamięci. Nazywają

Rejestry			
R0	R1		
R2	R3		
...			
R14	R15		
R16	R17		
...			
R24	R25		
R26	X	R27	5
R28	Y	R29	
R30	Z	R31	

się X, Y i Z[48]. Dzięki temu, że część instrukcji używa tylko ich, a nie wszystkich rejestrów, mogą one być krótsze i zmieścić informację o postinkrementacji, predekrementacji lub dodaniu liczby do wskaźników bez wykonywania dodatkowych instrukcji. Pozwala to na szybkie odczytywanie danych. Należy wyróżnić cztery możliwe kombinacje działań na rejestrach [48]:

- Jedno 8-bitowe odczytanie i jedno 8-bitowe zapisanie.
- Dwa 8-bitowe odczytanie i jedno 8-bitowe zapisanie.
- Dwa 8-bitowe odczytanie i jedno 16-bitowe zapisanie.
- Jedno 16-bitowe odczytanie i jedno 8-bitowe zapisanie.

Te działania są wykonywane za pomocą różnych mechanizmów adresowania[50]. Każdy z nich może służyć do samego odczytu lub też zapisu danych. Zawsze odczytywana jest para rejestrów a to, czy zostanie użyty tylko jeden, czy dwa bajty zależy od kodu operacji ALU. Wykorzystywane jest:

- adresowanie pojedyncze
- adresowanie podwójne
- adresowanie bezpośrednie za pomocą adresu rejestru przechowywanego w instrukcji
- adresowanie bezpośrednie za pomocą adresu danych przechowywanego w drugim słowie instrukcji
- adresowanie za pomocą wskaźnika X, Y, lub Z

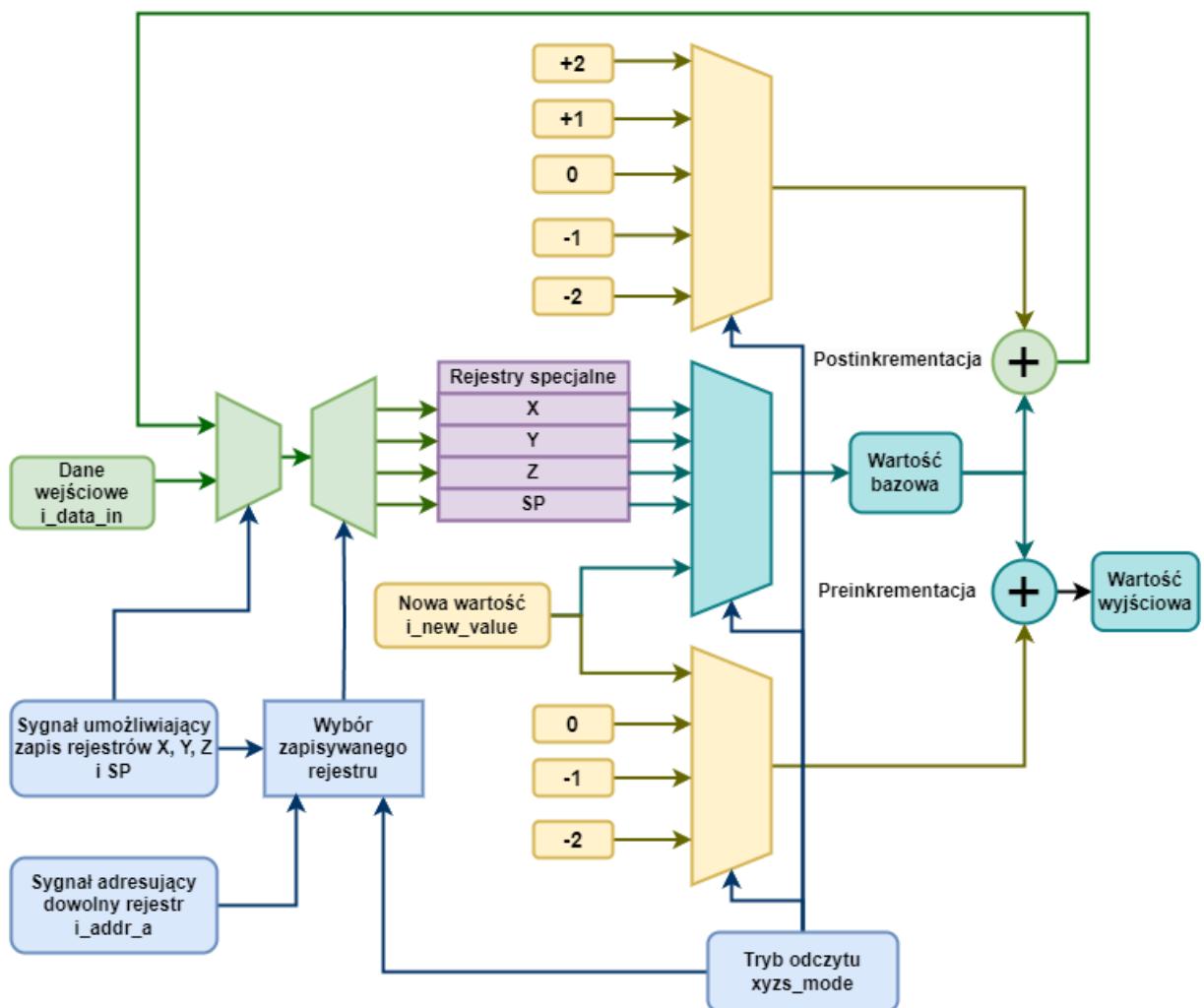
Implementacja adresowania pojedynczego Adres jest wybierany za pomocą sygnału wejściowego i_addr_a. Możliwe jest zapisanie danych do pojedynczego rejestru lub jego pary. Decyduje o tym dwubitowy sygnał i_we_reg_a.

Implementacja adresowania podwójnego Pierwszy adres wybierany jest tak, jak przy adresowaniu pojedynczym. Drugi natomiast wybierany jest przez sygnał i_addr_b. Nie jest możliwe jego zapisanie.

Adresowanie bezpośrednie za pomocą adresu danych przechowywanego w drugim słowie instrukcji Adresowanie zewnętrzne rejestrów polega na zapisaniu rejestru adresowanego przez sygnał addr_io_reg danymi przekazywanymi przez data_io_reg, jeżeli jest ustalony sygnał i_we_io. Adres musi wynosić mniej niż 32.

Implementacja adresowania bezpośredniego za pomocą adresu rejestru przechowywanego w instrukcji oraz wskaźników Istnieje specjalny system odczytywania danych z rejestrów X, Y, Z i SP[50]. Służy on do pobierania adresów rejestrów zewnętrznych. Ich wartość bazowa jest wybierana na podstawie sygnału *i_xyzs_mode*. Może być równa zawartości SPstrów SP, X, Y, Z lub sygnału *i_new_value*, który jest używany w przypadku adresowania bezpośredniego. Następne ich wartość może zostać postinkrementowana lub predekrementowana w zależności od instrukcji. Na wyjściu *o_addr* znajduje się uzyskany w ten sposób adres. O tym, które rejesty zostaną zapisane, decyduje sygnał *i_we_xyzs*, którego wartość zależy od sposobu adresowania.

Na rysunku 27 znajduje się uproszczony schemat opisujący ten system adresowania. Pomija on odczytywanie wartości rejestrów X, Y, Z i SP za pomocą innych sposobów pobierania danych.



Rysunek 27. Implementacja adresowania rejestrów. Opracowanie własne.

Implementacja pamięci Do uproszczenia zapisu wyników mnożenia używany jest dodatkowy sygnał *i_we_reg_01*, który służy do uproszczenia zapisu *i_input_data* pierwszych dwóch rejestrów. Dodatkowo na wyjściu *o_z_data* zawsze znajduje się wartość wskaźnika Z.

Ze względu na mnogość możliwych kombinacji adresowania, nie jest możliwa implementacja rejestrów za pomocą podwójnie adresowanej pamięci. Wymagane jest umożliwienie jedno-

czesnego odczytu lub zapisu aż do trzech różnych rejestrów. Dlatego moduł zaimplementowano za pomocą przerzutników. To rozwiązanie używa około połowy zasobów używanych przez cały procesor, a nie udało się wydajniej zaimplementować tego modułu.

6.4.2.2 Rejestr ustawień

Oprócz wcześniej wspomnianych 32 rejestrów niezbędny do działania procesora jest rejestr ustawień[48]. Zawiera on informacje na temat ostatnio wykonywanej operacji arytmetycznej. Mogą one być używane przez program w celu wykrycia określonych zdarzeń, jednak podczas skoku programu do funkcji należy rejestr zapisać, po czym go wczytać. Zgodnie z rysunkiem 28 zawiera 8 bitów o następujących oznaczeniach:

- Bit 7. - I - Globalne umożliwienie przerwań. Jeżeli jest równy 0, to układ ignoruje wszystkie wektory przerwań, niezależnie od ich wartości. Jest zerowany po każdym przerwaniu i programista musi zadbać o jego ustawienie.
- Bit 6. - T - Magazyn kopiowanego bitu. Instrukcje BLD (wczytanie bitu) i BST (załadowanie bitu) używają tego bitu jako schowka wartości.
- Bit 5. - H - Pół reszty (ang. half-carry flag)[51] jest używane do oznaczania konieczności korekty operacji wykonywanych w kodzie BCD (ang. binary-coded decimal)[52].
- Bit 4. - S - Jego ustawienie oznacza, że wartość ostatniej operacji jest ujemna.
- Bit 3. - V - Przelanie operacji arytmetycznej (ang. overflow). Jego ustawienie oznacza, że wynik operacji matematycznej nie mieści się w reprezentacji liczbowej.
- bit 2. - N - Liczba ujemna. Jego ustawienie oznacza, że wynik operacji arytmetycznej jest ujemny.
- bit 1. - Z - Zero. Jego ustawienie oznacza, że wynik operacji arytmetycznej jest zerowy.
- bit 0. - C - Reszta z operacji.

Rejestr ustawień							
I	T	H	S	V	N	Z	C

Rysunek 28. Bity rejestrów ustawień. Opracowanie własne.

Jest on zaimplementowany w taki sposób, aby ustawienie wartości było możliwe tylko dla czystych bitów, dzięki czemu nie dojdzie do zapisania sąsiednich bitów przez przepelnienie.

Zapisywane są do niego dane z `i_flag_data`, jeżeli ustawiony jest bit `i_we_flag`. Zawartość rejestrów jest stale widoczna na wyjściu na `o_flags_data`. Dodatkowo może on być adresowany jako rejestr ogólny o adresie `0x5F`.

6.4.2.3 Rejestr wskaźnika stosu

Stos jest zaimplementowany jako fragment wewnętrznej pamięci RAM rosnący od wyższych adresów do niższych, a jego wskaźnik stosu jako dwa 8-bitowe rejestrów[48].

Rejestr wskaźnika stosu SP (ang. Stack Pointer) wskazuje na ostatnio dodaną do niego wartość. Polecenie PUSH zapisujące dane do stosu zmniejsza wskaźnik stosu, a polecenie POP odczytujące dane zwiększa go. Może on być adresowany poprzez adresy 0x5D i 0x5E.

Dane w stosie muszą zostać wpierw zapisane, a wskaźnik stosu powinien mieć zapisaną wartość na samym początku każdego programu. Musi mieć wartość powyżej 0x60. Zmiana wskaźnika programu za pomocą instrukcji call spowoduje zapisanie 2-bajtowego adresu aktualnie wykonywanej instrukcji do stosu, po czym zmniejszy SP o 2. Analogicznie, jest zwiększany o 2 w przypadku powrotu z podprogramu (RET) lub przerwania (RETI).

Cała pamięć mikrokontrolera ma przypisane unikalne adresy, co ujednolica zapisywanie danych do pamięci[48]. Kontakt z modułami zewnętrznymi jest realizowany poprzez wysłanie danych do odpowiedniego adresu z zakresu od 0x20 do 0x5E. Umożliwia to adresowanie 62 zewnętrznych rejestrów. Sygnał data_addr używany przez logikę modułu wykonywania instrukcji używa właśnie tych adresów do decydowania o źródle danych.

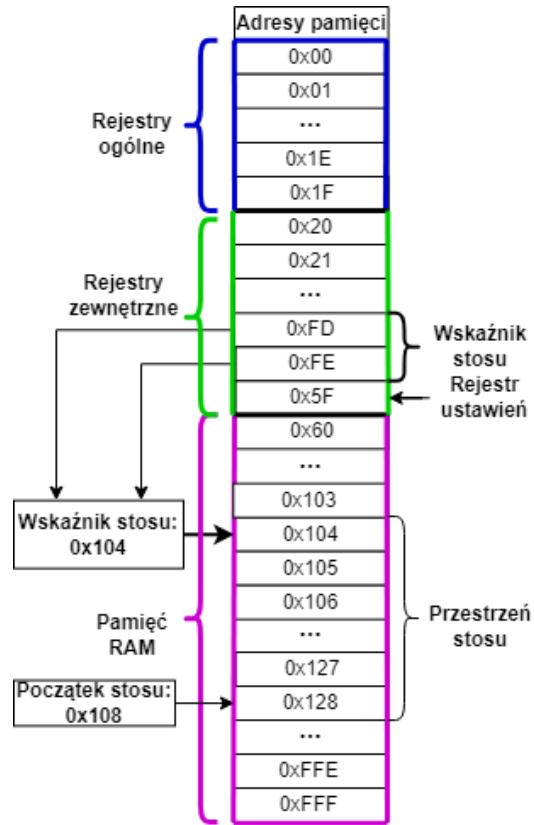
Na rysunku 29 przedstawiono podział adresów pamięci, pamięć zajmowaną przez stos i jego działanie w przypadku, w którym programista zdefiniuje jego początek jako 130, po czym program wpisze do niego kilka informacji.

6.4.2.4 Obsługa wejść i wyjść

Moduł rejestrów obsługuje porty wejść i wyjść. W celu obsługi rejestrów zewnętrznych używane są instrukcje IN i OUT, odpowiednio do odczytu i zapisu danych. Aby sygnały zewnętrzne mogły być ustalone, wymagane jest przekazanie przez sygnał data_addr odpowiedniego adresu z zakresu od 0x20 do 0x5E[53]. Oznacza to, że rejestr zewnętrzne mogą być tylko z tego zakresu. Przykładowo, instrukcja OUT 0x00 zostanie wykonana przez procesor z adresem 0x20, ale zapisze dane do adresu zewnętrznego 0x00. Pozwala to na obsługę pamięci danych za pomocą tych samych sygnałów, które obsługują rejestr wewnętrzne i zewnętrzne. Różni się tylko adres.

6.4.3 Realizacja pamięci danych

Pamięć danych jest 8-bitowa, ale niektóre instrukcje wymagają wpisywania danych do dwóch kolejnych adresów pamięci. Aby to umożliwić, pamięć jest zaimplementowana jako 16-bitowa,



Rysunek 29. Mapa adresów pamięci.
Opracowanie własne.

ale z możliwością wyboru, który bajt z aktualnie wykorzystywanych chcemy zapisać. Takie rozwiązanie znacznie ułatwia implementację pamięci i jest wspierane przez producenta za pomocą dostarczonej przez niego parametryzowej implementacji blokowej pamięci RAM. W przypadku, w którym adresujemy nieparzysty bajt, należy zamienić ze sobą dwa bajty danych wyjściowych i zrobić to samo dla dwóch bajtów danych wejściowych. Taka implementacja wiąże się z zaimplementowaniem dodatkowej logiki, ale jest prostsze, niż wykorzystanie podwójnej pamięci RAM, które wymagałoby inkrementacji adresu danych, spowalniając tym samym dostęp do niej.

Została ona zaimplementowana za pomocą bloku IP jako jedno-portowa pamięć o pojemności 8192 bajtów z portem WE [54]. Odczytywanie danych z pamięci danych 1 cykl zegara, ale wpierw muszą zostać ustawione poprawne sygnały wejściowe. W efekcie cała operacja odczytania zajmuje dwa cykle zegara, co zgadza się ze specyfikacją procesora[48].

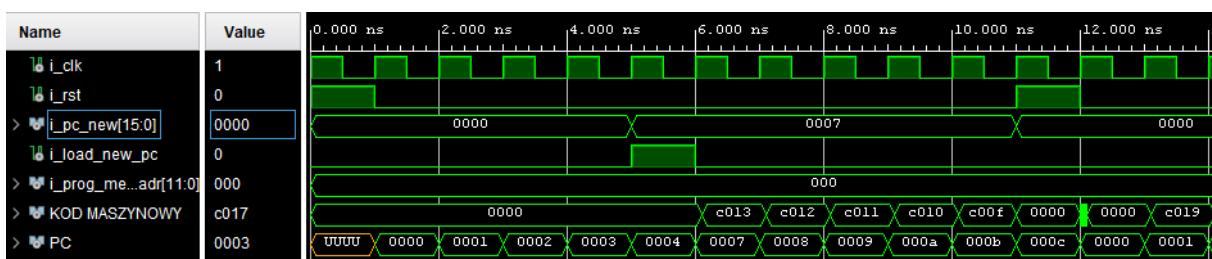
Rozdział 7: Symulacje procesora

Przeprowadzono wiele różnych symulacji procesora, z których część jest dokładnie przedstawiona w tym rozdziale. Symulowany był zarówno cały procesor, jak i jego poszczególne moduły, dzięki czemu było możliwe zweryfikowanie działania procesora oraz poprawa jego funkcjonalności.

7.1 Symulacja odczytywania pamięci programu

Moduł pobierania instrukcji powinien prawidłowo zmieniać wartość wskaźnika programu w zależności od odczytywanej instrukcji. W tym celu do pamięci programu został wgrany prosty program.

Na przeprowadzonej symulacji do pamięci programu wgrano instrukcje programu. W ciągu każdego kolejnego cyklu zegara powinna być odczytywana kolejna instrukcja, a wskaźnik PC inkrementowany. W przypadku, w którym sygnał `i_load_new_pc` przybierze wartość 1, wskaźnik PC powinien przybrać wartość z sygnału `i_pc_new`, po czym dalej ulegać inkrementacji. W przypadku wystąpienia resetu PC zeruje się i zatrzymuje inkrementację. Na rysunku 30 przedstawiono przebieg kluczowych sygnałów tej symulacji.



Rysunek 30. Przebieg wybranych sygnałów symulacji odczytywania danych z pamięci programu. Opracowanie własne.

Po ustaniu resetu PC zaczyna się inkrementować aż do ustawienia sygnału `i_load_new_pc`. Wtedy PC przybiera wartość 7 czyli dokładnie tyle, ile przekazuje sygnał `i_pc_new`. Po ustawieniu się sygnału `i_rst` wskaźnik PC się zeruje. Cała symulacja przebiega zgodnie z oczekiwaniemi.

Pamięć programu nie była symulowana w osobnej symulacji ze względu na jej niską złożoność. Jej działanie można sprawdzić za pomocą symulacji modułu odczytywania. Ma on też inne funkcjonalności, które były badane podczas symulacji całego mikrokontrolera. Wynika to z ich wysokiej integracji z innymi modułami.

7.2 Symulacje modułu dekodowania instrukcji

Moduł dekodowania jest odpowiedzialny za przetworzenie wcześniejszej odczytanego kodu operacyjnego do szeregu różnych sygnałów sterujących moduł ALU. W celu symulacji jego działania

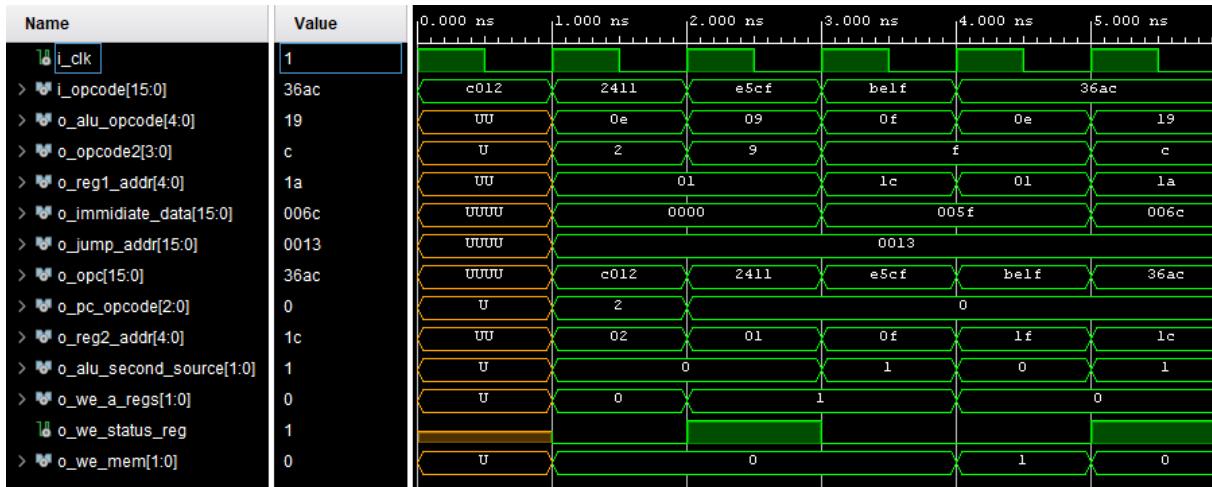
zostało wgrane 5, zmieniających się co cykl zegara instrukcji. Wszystkie wartości wyjściowe zostają ustawione 1 cykl zegara później, niż otrzymano kod operacyjny instrukcji, a każda z poniższych instrukcji zajmuje jeden cykl zegara. W przypadku braku zdefiniowanej przez instrukcję zmiany wartości sygnału, przybiera on wartość 0. Na rysunku 31 przedstawiono przebieg tej symulacji.

Nie jest potrzebne symulowanie wszystkich instrukcji ze względu na ich podobne działanie z punktu widzenia dekodera. Ewentualne nieprawidłowości łatwiej wykryć podczas symulacji całego procesora. Do symulacji wybrano tylko kilka, z których każda ma inne spodziewane działanie. Jest ono opisane poniżej.

1. RJMP .+36 - Zmienia wartość wskaźnika PC o wartość zawartą w instrukcji. Sygnał `o_pc_opcode` przybiera wartość 2 informującą o tym, że PC ma zmienić swoją wartość na tą przekazywaną przez sygnał wyjściowy `o_jump_addr`, na którym z kolei powinna pojawić się nowa wartość 0x13.
2. EOR r1, r1 - Na wyjściu `o_alu_opcode` ma się znaleźć znalezć się unikalny kod 01001. Informuje on o tym, jakie działanie ma wykonać moduł ALU. Powinien zostać zapisany jeden rejestr, dlatego sygnał `o_we_a_regs` powinien przyjąć wartość 01. Jednocześnie rejestr ustawień powinien zostać zapisany, dlatego bit wyjściowy `o_we_a_regs` powinnien zostać ustawiony.
3. LDI r28, 0x5F - Na wyjściu `o_alu_opcode` powinno pojawić się sygnał 01111. Sygnał wyjściowy `o_alu_second_source` przybiera wartość 01, `o_reg1_addr` adresuje wybrany przez instrukcję rejestr o numerach od 16 do 31. W tym przypadku jest to 0x1c, co odpowiada rejestrowi 28. Sygnał `o_immediate_data` przybiera wartość zawartą w instrukcji czyli 0x5F, a `o_we_a_regs` powinno przyjąć wartość 1 ze względu na zapisanie danych do jednego rejestrów.
4. OUT 0x3f, r1 - `o_alu_opcode` powinno przyjąć wartość 0x0E. Sygnał `o_immediate_data` powinien być równy adresowi przekazanemu przez instrukcję powiększonemu o 0x20, czyli 0x5F. Dolny bit sygnału `o_we_mem` zostaje ustawiony oznaczając umożliwienie zapisania sygnału wyjściowego procesora WE.
5. CPI r26, 0x6C - porównuje zawartość rejestrów do stałej. W tym celu `o_alu_opcode` jest ustawiony na 0x19 oznaczając wykonanie przez ALU operacji odejmowania. Sygnał `o_immediate_data` przyjmuje przekazywaną przez instrukcję wartość 0x6c. W celu jej wykorzystania przez ALU sygnał `o_alu_second_source` już już przyjmuje wartość 1. Sygnał wyjściowy `o_we_status_reg` przyjmuje wartość 1 umożliwiając zapisanie nowego rejestrów ustawień, podczas gdy pozostałe sygnały WE pozostają wyzerowane dzięki czemu rezultat działania nie zostaje zapisany do rejestrów. Jest to główna różnica między operacją CPI a SUBI.

7.3 Symulacje modułu wykonywania instrukcji

Aby przetestować działanie modułu symulowane jest wykonanie prostego, zmyślonego programu składającego się z 4 różnych instrukcji. Nie testuje on wszystkich funkcji modułu, a jedynie sprawdza, czy jest on w stanie wykonać kilka podstawowych funkcjonalności. Dalsze testowanie modułu przeprowadzono po połączeniu go z resztą procesora.



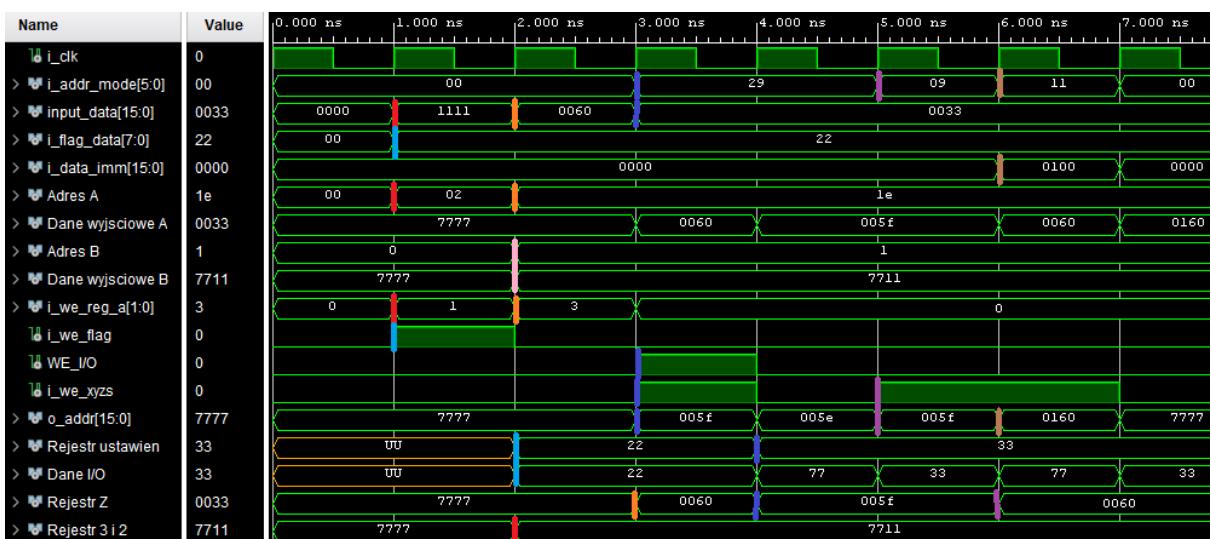
Rysunek 31. Przebieg wybranych sygnałów symulacji dekodowania instrukcji. Opracowanie własne.

7.3.1 Symulacje działania rejestrów

W celu przetestowania działania modułu rejestrów przeprowadzono symulację testującą kilka wybranych funkcjonalności. Dzieli się ona na poszczególne etapy które są zaznaczone na rysunku 32 różnymi kolorami.

1. Najpierw adres A przybiera wartość 2, a i_we_reg_a wartość 01. Umożliwia to zapisanie dolnego bitu sygnału input_data do rejestru numer 2. Następuje to po jednym cyklu zegara co pokazuje poprawne zapisywanie danych do rejestrów ogólnych.
2. Bit i_we_flag zostaje ustawiony, dzięki czemu po jednym cyklu zegara rejestr ustawień przybiera wartość z i_flag_data.
3. Adres A przybiera wartość 1e co oznacza, że wskazuje rejestr Z. Sygnał i_we_reg_a pozwala na zapisanie jego dwóch bajtów danymi z i_input_data, co następuje po jednym cyklu zegara- rejestr Z ma teraz wartość równą 0x60. W trakcie całej symulacji dane wyjściowe A zmieniają się przekazując aktualną wartość rejestrów Z.
4. Na skutek zmiany wartości adresu B, dane wyjściowe B przybierają wartość drugiej pary rejestrów. Zgodnie z oczekiwaniemi zachodzi to bez widocznego na symulacji opóźnienia.
5. Bit i_we_dm przybiera wartość 1 umożliwiając wpisanie danych z sygnału do rejestru adresowanego za pomocą i_addr_mode. W tym przypadku jest to predekrementowany rejestr Z. i_we_xyzs umożliwia zmianę jego wartości w tym trybie adresowania. Oznacza to, że adres, do którego wpisywane są dane wynosi o 1 mniejszej, niż dotychczasowa wartość rejestru Z. Czyli 0x5F co jest adresem ogólnym rejestrów ustawień. Ta wartość dodatkowo pojawia się na wyjściu o_adres. Jego wartość powinna się zmienić na wartość z i_input_data czyli 0x33, co widać po jednym cyklu zegara. Nowa wartość rejestru Z również jest ustawiona dopiero po jednym cyklu zegara. D

6. Tryb adresowania zmienia się na postinkrementację rejestru Z. Powoduje to natychmiastowe ustawienie dotychczasowej wartości rejestru Z na wyjściu o_addresses. Wcześniej na porcie była wartość 0x5E ze względu na utrzymanie się poprzedniego trybu adresowania. Teraz pojawia się 0x5F, a po jednym cyklu zegara wartość rejestru wynosi 0x60.
7. Tryb adresowania zmienia się na dodanie do rejestru Z wartości z sygnału wejściowego i_data_imm. Oznacza to, że wartość wyjściowa na porcie o_addresses zmienia się na 0x01000x0060 czyli 0x0160. Ta zmiana nie jest zapisywana do rejestru Z.



Rysunek 32. Przebieg wybranych sygnałów symulacji rejestrów. Opracowanie własne.

7.3.2 Symulacje działania modułu ALU

W symulacji samego modułu ALU najistotniejsze jest sprawdzenie poprawności obliczania bitów ustawień. Pozostałe funkcje modułu są powtarzalne i zostaną sprawdzone w symulacji całego procesora.

7.3.3 Symulacje działania całego modułu

Aby sprawdzić działanie modułu wykonywania zasymulowano wykonanie kilku różnych instrukcji które umożliwiają działanie funkcjonalności modułu. Symulacja, na której sygnały dotyczące każdej instrukcji są zaznaczone kolorami, jest przedstawiona na rysunku 33. Instrukcje są wykonywane kolejno, a jej efekty są widoczne 1 cykl zegara później. Symulowane instrukcje to:

1. EOR r1 r1 - Wykonuje funkcję logiczną XOR na dwóch rejestrach i wynik zapisuje do pierwszego z nich. W przypadku użycia dwa razy tego samego rejestru jest to prosty sposób na jego wyzerowanie. Aby wykonać tą instrukcję ustawiane jest 5 sygnałów:
 - i_alu_opcode ustawiony jest na wartość 0x09, umożliwiający wykonanie tej funkcji logicznej przez ALU.
 - i_addr_a i i_addr_b przybierają wartości wskazujące przekazywane przez in-

strukturę rejestrów. W tym przypadku są to rejestrory r1 wskazywane adresem 0x01.

- *i_we_a_regs* i *i_we_flag* umożliwiają zapisanie rezultatów do odpowiednich rejestrów.

W efekcie wykonania instrukcji gorny bit pierwszej pary rejestrów zeruje się. Jednocześnie zapisywany jest register ustawień, co jest to zgodne z oczekiwaniami.

2. LDI r28, 0x5f - ma za zadanie zapisanie danych przekazywanych przez instrukcję do rejestrów. W tym celu zostają ustawione następujące sygnały:

- *i_alu_opcode* zostaje ustawiony na wartość zlecającą ALU przepisanie drugiego wejścia danych do wyjścia.
- *i_addr_a* zostaje ustawiony na adres rejestrów z instrukcji czyli 28, co w systemie szesnastkowym jest równe 0x1C.
- *i_we_reg_a* umożliwia zapisanie wyniku operacji ALU do wybranego rejestrów.
- *i_data_imm* zawiera dane które mają zostać zapisane do rejestrów czyli 0x5f.
- *i_alu_imm_sel* zostaje ustawiony w celu wybrania przez ALU wartości zawartej w instrukcji.

W efekcie na dolnym bajcie 15. pary rejestrów zostaje ustawiona wartość 0x5f, co jest zgodne z oczekiwaniami. Ta instrukcja nie modyfikuje rejestrów ustawień.

3. OUT 0x3f, r28 - służy do komunikacji zewnętrznej procesora. Na wyjściu danych procesora ustawia zawartość przekazywanego przez instrukcję rejestrów, a na wyjściu adresu- przekazywany adres. Ustawia też sygnał *o_we*. W tym przypadku jest to 0x3e. Moduł wykonywania odczytuje sygnały wejściowe:

- *i_alu_opcode* sprawia, że ALU przepisuje dane wchodzące na jego pierwszy port danych, czyli z rejestrów A.
- *i_addr_a* wybiera rejestr wchodzący do ALU. Nie zostanie on zapisany ze względu na nie ustawienie sygnału *i_we_a*.
- *i_we_mem* umożliwia ustawienie sygnału wyjściowego procesora WE.

W efekcie na zostaje zapisany rejestr ustawień, ponieważ jego adres ogólny to 0x3f.

4. RJMP +2 - służy do zmiany wartości wskaźnika PC o wartość przekazywaną przez instrukcję. Służą do tego dwa sygnały wejściowe:

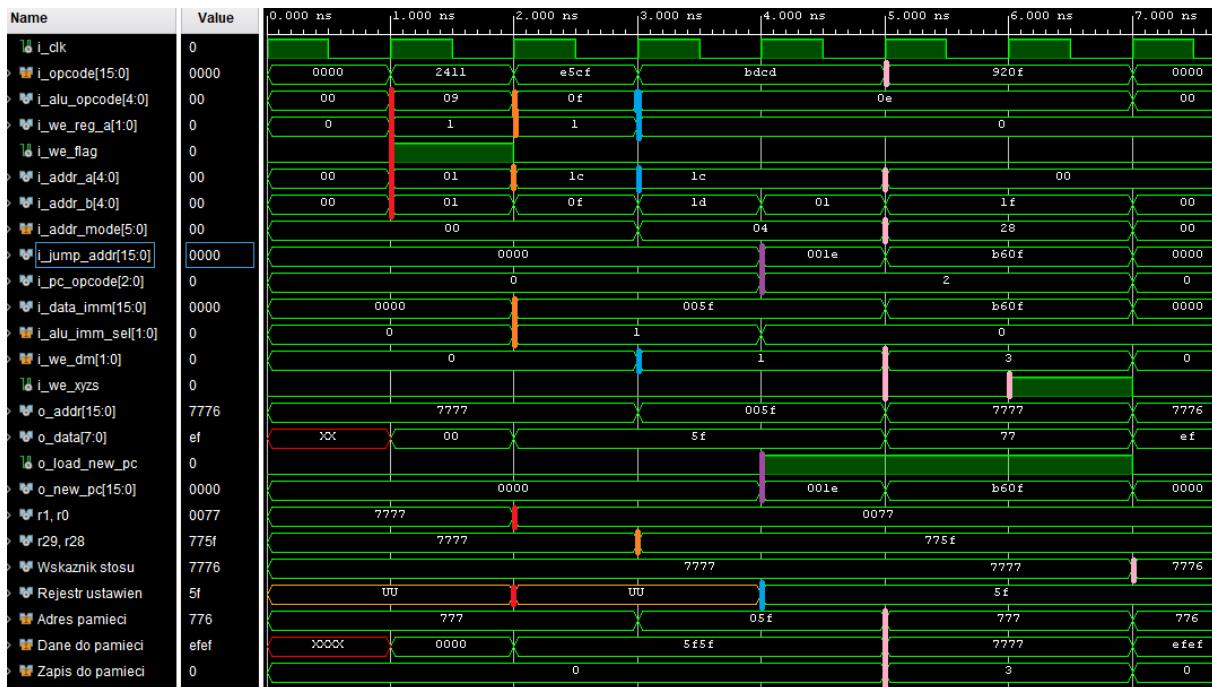
- *i_pc_opcode* przechowuje informacje o zmianie wartości PC na tą z *i_jump_addr*.
- *i_jump_addr* przechowuje nową wartość PC.

ALU nie bierze udziału w tej instrukcji ze względu na nie zapisywanie danych do rejestrów. W efekcie od razu zostają ustawione sygnały *o_load_new_pc* i *o_skip_inst* wraz z nową wartością wskaźnika PC równą tej ustawionej na sygnale *i_jump_addr*.

5. PUSH r0 - ma za zadanie zapisać zawartość rejestru 0 do stosu znajdującego się w pamięci danych oraz zmniejszyć wartość wskaźnika stosu o 2. W tym celu ustawia następujące sygnały:

- *i_alu_opcode* jest ustawiony tak, aby ALU przepisało dane ze swojego pierwszego wejścia.
- *i_addr_a* jest ustawiane na adres rejestru r0 czyli 0.
- *i_addr_mode* ma wartość odpowiadającą dekrementacji wskaźnika stosu przez moduł rejestrów.
- *i_we_dm* ma ustawione oba bity. Powoduje to umożliwienie zmiany obu bajtów wskazywanych przez adres pamięci danych
- *i_we_xyzs* jest ustawione w drugim cyklu zegara trwania instrukcji. Powoduje to zapisanie nowej wartości wskaźnika dopiero w drugim cyklu zegara.

W efekcie na sygnale wejściowym pamięci danych pojawia się zawartość rejestru 0. Jest to 0x77. Umożliwiony jest też jej zapis do adresu wskazywanego przez wskaźnik stosu, a on jest dekrementowany z początkowej wartości 0x7777 do 0x7776. Zapis do pamięci następuje w pierwszym cyklu zegara, a zapisanie nowej wartości wskaźnika, w drugim.



Rysunek 33. Przebieg wybranych sygnałów symulacji wykonywania instrukcji. Opracowanie własne.

7.4 Symulacje całego procesora

Ze względu na wiele zależności pomiędzy poszczególnymi modułami, symulacje poszczególnych modułów są utrudnione i dają gorszygląd działania procesora niż symulacje jego całości. Dlatego przedstawione są symulacje całego mikrokontrolera ze szczególnym uwzględnieniem jego poszczególnych funkcjonalności.

7.4.1 Szczegółowa symulacja programu

Przeprowadzono liczne symulacje działania programu wgranego wcześniej do pamięci CPU. W celu nakreślenia przebiegu przeprowadzenia tych symulacji szczegółowo została omówiona tylko jedna z nich. Dla kilku innych zostały przedstawione jedynie rezultaty końcowe. Dla przykładowego programu, którego kod w języku C jest przedstawiony na listingu 1, przedstawiony jest przebieg wybranych symulowanych sygnałów obrazujących poprawne działanie całego procesora. Wpierw podłącza on DPU o numerze 0 do interfejsu o numerze 4. Następnie wysyła w pętli do użytkownika 4 litery.

```
1 int main(){
2     select_interface(0, 4);
3     char my_text []= "ABCD";
4     while(1){
5         send_bytes(my_text, 4);
6     }
7 }
```

Listing 1. Program przesyłający 4 litery do użytkownika

Ten program jest zamieniany przez kompilator do serii instrukcji zapisywanych później do pamięci programu. Na poniższym listingu jest pokazana ta ich część, która jest wykonywana przez procesor. Są one pobrane z pliku .iss i przedstawione w taki sposób, w jaki przedstawia je kompilator.

Wskaźnik PC będzie zawsze przybierał dwukrotnie mniejsze wartości, niż oczekuje tego program, ponieważ pamięć programu jest 16 bitowa, a kompilator używa adresowania 8-bitowego. Oznacza to, że instrukcja, która zgodnie z powyżej przedstawionym plikiem powinna być zapisana do adresu 0x26, będzie zapisana do adresu 0x13 pamięci.

Każdy program zaczyna się od adresu 0. Najpierw procesor inicjuje wybrane rejestrysty, w ramach czego wykonywane są kolejno operacje:

1. Instrukcja RJMP zmienia wartość wskaźnika PC o połowę 0x26 czyli 0x13, adresując początek inicjalizacji procesora.
2. Instrukcja EOR czyści pierwszy rejestr pamięci.
3. Instrukcja OUT 0x3f zapisuje wartość wcześniej wyczyszczonego rejestrud do rejestrust ustawień.
4. Instrukcje LDI zapisują wartości 95 i 4 do dwóch rejestrów R29 i R28.
5. Instrukcja OUT zapisuje dane z rejestrów R29 i R28 do wskaźnika stosu inicjalizując jego wartość.

```

00000000 <__vectors>:
 0: 12 c0          RJMP .+36      ; 0x26 <__ctors_end>

00000026 <__ctors_end>:
26: 11 24          EOR r1, r1
28: 1f be          OUT 0x3f, r1 ; 63
2a: cf e5          LDI r28, 0x5F ; 95
2c: d4 e0          LDI r29, 0x04 ; 4
2e: de bf          OUT 0x3e, r29 ; 62
30: cd bf          OUT 0x3d, r28 ; 61
32: 02 d0          RCALL .+4    ; 0x38 <main>
34: 0c c0          RJMP .+24   ; 0x4e <_exit>

10 00000036 <__bad_interrupt>:
 36: e4 cf          RJMP .-56   ; 0x0 <__vectors>

00000038 <main>:
38: 84 e0          LDI r24, 0x04 ; 4
3a: 83 b9          OUT 0x03, r24 ; 3
3c: 31 e4          LDI r19, 0x41 ; 65
3e: 22 e4          LDI r18, 0x42 ; 66
40: 93 e4          LDI r25, 0x43 ; 67
42: 84 e4          LDI r24, 0x44 ; 68
44: 30 bb          OUT 0x10, r19 ; 16
46: 20 bb          OUT 0x10, r18 ; 16
48: 90 bb          OUT 0x10, r25 ; 16
4a: 80 bb          OUT 0x10, r24 ; 16
4c: fb cf          RJMP .-10   ; 0x44

```

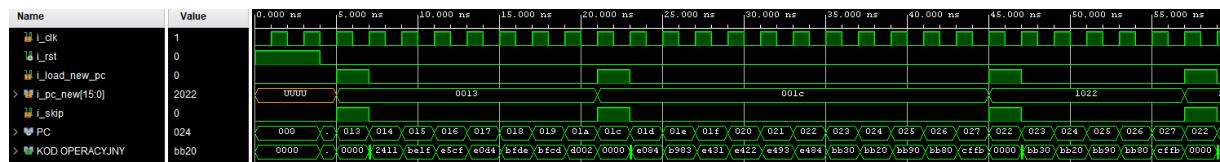
Rysunek 34. Część wygenerowanych przez kompilator instrukcji procesora

6. Po odczytaniu instrukcji RCALL wskaźnik PC zmienia swoją wartość adresując pierwszą instrukcję funkcji main().

Dopiero po tej inicializacji wykonywane są instrukcje zapisane przez programistę w programie. Wcześniej omówiony kod zamieniany jest na kolejne instrukcje:

1. Instrukcja LDI wpisuje do rejestru R24 wartość 0x04.
2. Instrukcja OUT wpisuje do rejestru zewnętrznego o adresie 0x03 wartość z wcześniej zapisanego rejestru R24. W ten sposób wykonywane jest funkcja `select_interface(0, 4)`.
3. 4 instrukcje LDI wpisują wartości 0x41, 0x42, 0x43 i 0x44 do czterech rejestrów R19, R18, R25 i R24. Te wartości w kodzie ASCII reprezentują znaki "A", "B", "C" i "D".
4. Instrukcje OUT wysyłają do rejestru zewnętrznego 0x10 wcześniej zapisane wartości. Adres odpowiada adresowi modułu wysyłania danych do użytkownika.
5. Instrukcja RJMP zmienia wartość wskaźnika PC na 0x22, wykonując raz jeszcze instrukcję OUT. Po ich wykonaniu ta instrukcja znowu zmniejszy wartość PC. W ten sposób zaimplementowana jest stała pętla.

Pamięć programu jest inicjalizowana z tymi instrukcjami, umożliwiając sprawne przeprowadzenie symulacji. Na rysunku 35 przedstawiono przebieg kluczowych sygnałów modułu pobierania instrukcji.



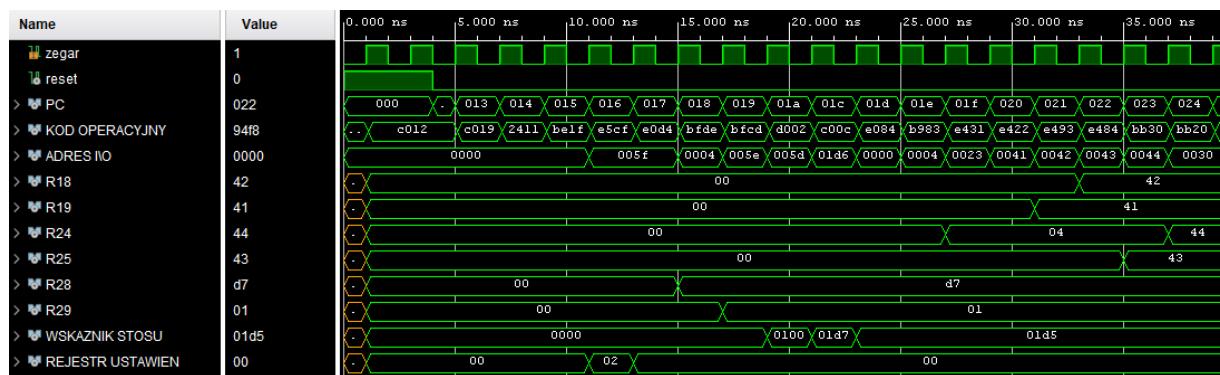
Rysunek 35. Przebieg wybranych sygnałów modułu pobierania instrukcji. Opracowanie własne.

Wskaźnik programu zmienia się zgodnie z oczekiwaniami. Wpierw jego wartość zmienia się na 0x13, rośnie aż do 0x1A po czym przyjmuje wartość 0x1c odpowiadający pierwszej instrukcji funkcji main(). Po jej odczytaniu wartość wskaźnika spada umożliwiając ponowne odczytanie wcześniej wykonanych instrukcji, co odpowiada implementacji pętli.

Na rysunku 36 przedstawiony jest przebieg wartości używanych przez program rejestrów. Widać kolejno następujące zmiany ich wartości:

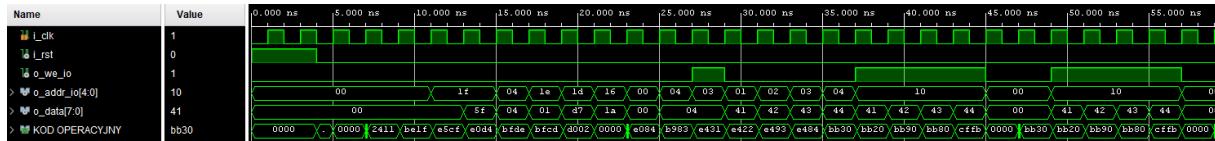
1. W trakcie resetu wszystkie rejesty są czyszczone.
2. Rejestr R1 przyjmuje wartość 0 za pomocą instrukcji EOR. Rejestr ustawień przyjmuje wartość 0x02. Oznacza to, że wynik operacji EOR jest równy zero.
3. Do rejestrów ustawień została zapisana wartość z rejestrów R1, czyli 0.
4. Na skutek dwóch instrukcji LDI do rejestrów R28 i R29 została zapisana wartość 0xD7, a do rejestrów R24, R19, R18 i R25 wartość 0x00.
5. Do rejestrów wskaźnika stosu za pomocą dwóch instrukcji OUT została zapisana wartość 0x01D7.
6. Na skutek instrukcji RCALL wskaźnik stosu zmniejszył się o 2. Jest to spodziewane, ponieważ do stosu została zapisana wartość PC.
7. W wyniku operacji LDI rejestr R24 przybrał wartość przekazywaną przez te instrukcje.

Każda z tych zmian jest zgodna z oczekiwaniami i pokazuje poprawne działanie procesora.



Rysunek 36. Przebieg wybranych sygnałów modułu wykonywania. Opracowanie własne.

Na rysunku 37 przedstawiono przebieg sygnałów końcowych z procesora. Wpierw wysyłane są dane 0x04 do adresu 0x03. Następnie do adresu 0x10 wysyłane są kolejno dane 0x41, 0x42, 0x43 i 0x44, które w kodzie ASCII odpowiadają liczbom "A", "B", "C" i "D". Następnie ponawiane jest ich wysłanie, a w dalszej, nie zamieszczonej tutaj części symulacji wysyłane są one w nieskończoność zgodnie z zasadą działania pętli stałej `while(1)`. Całe działanie programu jest zgodne z oczekiwaniami co obrazuje, że procesor jest w pełni sprawny.



Rysunek 37. Przebieg wybranych sygnałów symulacji procesora. Opracowanie własne.

7.5 Symulacja przerwań

Do przetestowania działania wymiany informacji pomiędzy CPU a DPU służą programy, których fragmenty odpowiednio przedstawiono na listingach 2 i 3. Można wyróżnić kolejne etapy działania mikrokontrolera:

1. CPU inicjalizację swoją transmisję z DPU 0, po czym wysyła dane do DPU i czeka na przerwania. W tym samym czasie DPU aktywuje obsługę przerwań.
2. DPU odbiera wektor `INTER_MESS_FROM_CPU`, odczytuje dane po czym wysyła je z powrotem do CPU.
3. CPU odbiera wektor `INTER_MESS_FROM_DPU`, po czym odczytuje dane z DPU 0, inkrementuje je, a następnie wysyła je do użytkownika, odsyła do DPU i umożliwia odebranie kolejnego przerwania od DPU.

Kroki 2 i 3 są powtarzane w nieskończoność. Zgodnie z programem, na skutek współpracy dwóch rdzeni powinnien zachodzić stopniowy wzrost wysyłanych do użytkownika wartości.

```

1 int main(void){
2     uint8_t data = 0;
3     clear_dpu_from_cpu_transmision();
4     add_dpu_from_cpu_transmision(0);
5     send_data_from_cpu_to_interface(data);
6     allow_mes_from_dpus();
7     sei(); // enable global interrupts
8     while(1);
9 }
10 ISR(INTER_MESS_FROM_DPU){
11     uint8_t data;
12     data = DPU0 + 1;
13     send_byte_to_user(data);
14     _delay_us(10);
15     send_data_from_cpu_to_interface(data);
16     allow_mes_from_dpus();
17     sei();
18 }
```

Listing 2. Program CPU wymieniający informację z DPU

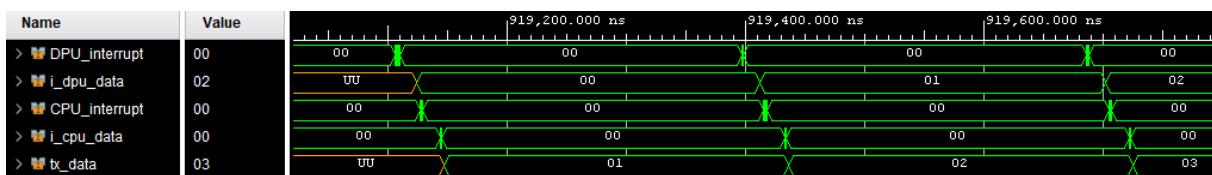
```

1 int main(void){
2     sei();
3     while(1);
4 }
5 ISR(INTER_MESS_FROM_CPU){
6     data = SELECT_CPU_DATA;
7     send_data_to_cpu(data);
8     sei();
9 }
```

Listing 3. Program DPU wymieniający informację z CPU

Po wgraniu kodu programu do symulowanego mikrokontrolera za pomocą sygnału UART Fragment symulacji powyższego programu przedstawiony jest na rysunku 38. Uzyskano ją po przez wgranie skompilowanego kodu programu do symulowanego mikrokontrolera za pomocą sygnału UART. Widać na niej kolejno wykonywane czynności:

1. Generacja wektora przerwań do DPU.
2. Wybranie nowych danych wejściowych do DPU.
3. Wysłanie wektora przerwań do CPU.
4. Wybranie nowych danych wejściowych do CPU.
5. Wysłanie zinkrementowanych danych do modułu UART.

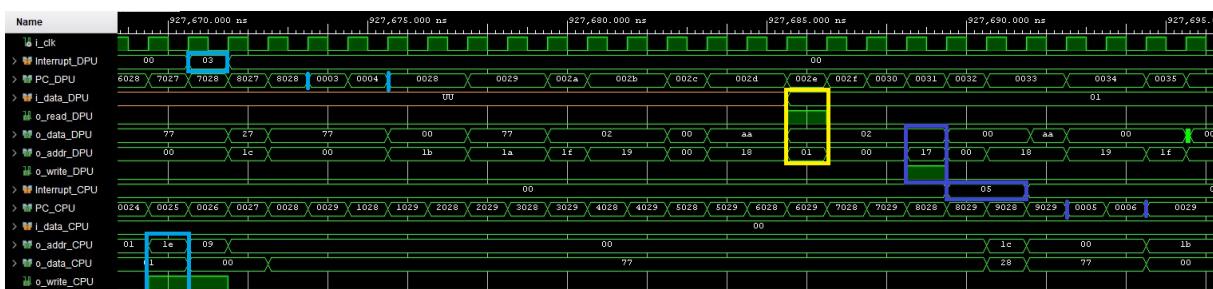


Rysunek 38. Fragment symulacji. Opracowanie własne.

Z każdym razem dane są o 1 większe niż otrzymane poprzednio co pokazuje, że układ działa zgodnie z oczekiwaniami. Nie ukazuje on jednak dokładnego mechanizmu wysyłania i odbierania wektorów przerwań, dlatego na rysunku 39 przedstawiono dokładnie w jaki sposób one działają. Zachodzą kolejne, zaznaczone na symulacji zdarzenia:

1. CPU wysyła na odpowiedni adres liczbę 1.
2. DPU otrzymuje wektor przerwania 0x03, co odpowiada wysłaniu wiadomości przez CPU.
3. Na skutek zmiany kodu operacyjnego wskaźnik PC zmienia swoją wartość na 0x03, co odpowiada wektorowi przerwania. Wskazuje on teraz instrukcję RJMP.
4. Po wykonaniu tej instrukcji wskaźnik PC przyjmuje wartość 0x28, rozpoczynając wykonywanie funkcji ISR(INTER_MESS_FROM_CPU).
5. DPU odczytuje liczbę 1 po wybraniu jej za pomocą odpowiedniego adresu.

6. DPU zapisuje do odpowiedniego rejestru odczytane dane, w ten sposób wysyłając przerwanie do CPU.
7. Wektor przerwania DPU jest odczytany, co powoduje analogiczne działania, jak w przypadku DPU.
8. Wektor się zeruje cykl zegara szybciej niż w przypadku DPU ze względu na to, że jest on ustalany w procesie synchronicznym, a nie w asynchronicznym jak w przypadku DPU.



Rysunek 39. Fragment symulacji. Opracowanie własne.

Rozdział 8: Opracowanie interfejsów

Interfejsy umożliwiają komunikację za pomocą protokołów komunikacji między podłączonym do niego DPU a czujnikami zewnętrznymi. W celu zademonstrowania funkcjonalności systemu zabezpieczeń zostały wybrane 4 z nich do implementacji: UART, $I2^2C$, SPI i PWM.

8.1 Opracowanie wspólnych funkcjonalności interfejsów

Wszystkie interfejsy muszą mieć jednakowe porty interfejsu komunikacyjnego wewnętrznego, aby było możliwe podłączenie każdego z nich do każdego procesora. Ze względu na mnogość możliwych konfiguracji protokołów komunikacji, niezbędne jest zastosowanie adresowanych rejestrów wewnętrznych służących do odpowiedniego ustawiania interfejsów.

Interfejsy używają pamięci, do których są zapisywane odebrane dane przed ich wysłaniem do procesora oraz dane, które mają zostać wysłane. Pozwala to na szybsze działanie procesorów, które mogą zlecić wysłanie danych, po czym zająć się innymi obliczeniami. Oprócz tego interfejsy używają adresowane rejestry. Są one zarezerwowane pod różne funkcje. Niektóre z nich są wspólne dla każdego, zaś inne specyficzne dla danego protokołu komunikacji. Wspólne rejesty to:

- Wejście danych. Wysłanie na ten adres spowoduje zapisanie danych do stosu.
- Wyjście odebranych danych. Wybranie tego adresu spowoduje odczytywanie stosu z odczytanymi przez interfejs danymi.
- Tryb wysyłania protokołu.
- Ustawienie bitu wysłania wcześniej zapisanych danych.
- Odczytanie statusu stosów- pełny/pusty.
- Odczytanie statusu interfejsu- zajęty/wolny.
- Odczytanie ostatnio odebranej przez interfejs wartości. Może być odczytywana wiele razy bez ingerowania w stos. Jest to użyteczne do podglądania wartości, a później jej pełnego odczytania.

Dane muszą zostać wpisane do pamięci interfejsu, aby móc zostać wysłane przez protokół komunikacji. W celu uproszczenia działania interfejsu oraz zmniejszenia ilości wykonywanych przez procesor operacji, pamięć jest stosem. Oznacza to, że można do niej tylko zapisywać kolejne dane, bez możliwości zmiany pojedynczego bajtu. W przypadku modyfikacji wysyłanego wyrażenia należy przesłać od początku nowe dane do wysłania. Takie rozwiązanie uprasza programowanie. W zależności od trybu wysyłania protokołu, są one po wysłaniu usuwane lub są wysyłane kolejny raz. Programując należy zwrócić uwagę, aby wysyłane i odbierane dane

zmieściły się w 8 bajtowej pamięci. Do większości zastosowań jest ona w zupełności wystarczająca, ale w sytuacjach, w których nie jest, należy zastosować odpowiedni tryb wysyłania i odbierania danych. Definiuje on częstotliwość, z jaką będą wysyłane informacje do urządzeń zewnętrznych. Istnieje możliwość przesyłania danych do pamięci przed lub po wybraniu trybu pracy. Zaimplementowane tryby pracy to:

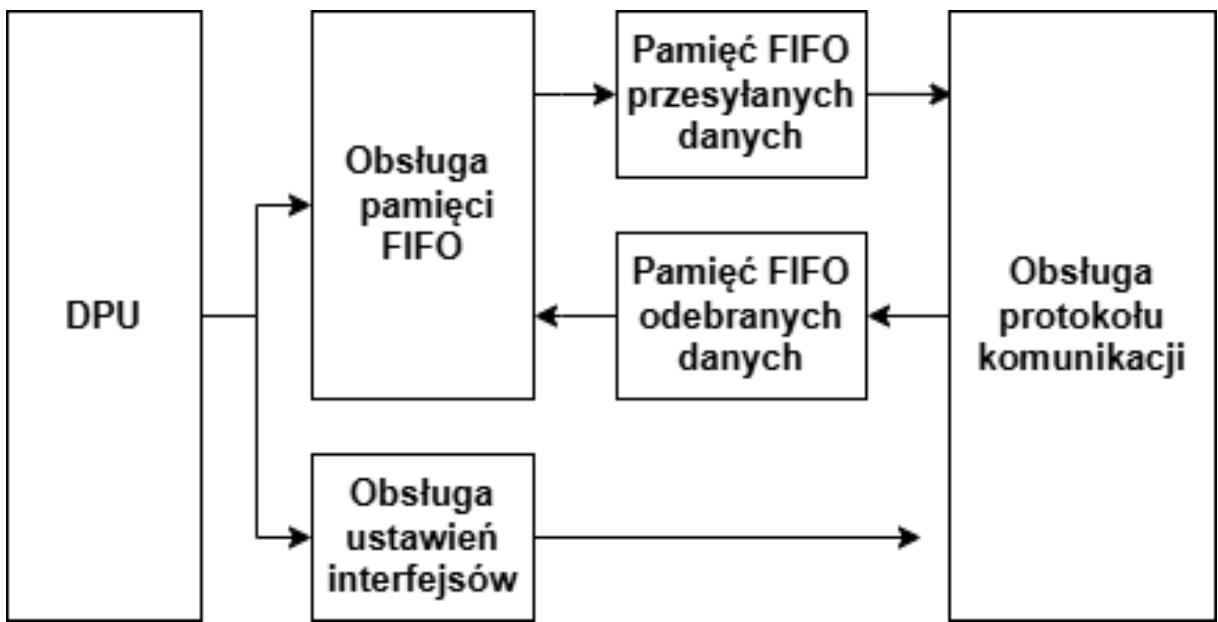
- Wysyłanie jednorazowe. W takim przypadku stos jest traktowany jako struktura FIFO (ang. first in, first out), w którym informacje są wysyłane przez interfejs w takiej kolejności, w jakiej zostały odebrane, a wysyłanie zaczyna się od razu po zapisaniu danych do interfejsu, lub w momencie ustawienia tego trybu.
- Wysyłanie ciągłe, w którym pamięć działa w pętli, nieustannie wysyłając zapisane w pamięci dane.
- Czyszczenie pamięci, która działając w pętli jest cały czas odczytywana. Umożliwia to zmianę wysyłanych danych w trybie ciągłym. Po jego wybraniu niezbędne jest wybranie kolejnego trybu wysyłania i zapisanie nowych danych do pamięci.
- Wysyłanie zapisanego wcześniej protokołu za każdym razem, gdy do specjalnego rejestru przerwania zostaje zapisany bit. Istnieje możliwość wysłania pierwszej serii danych przy wybraniu tego trybu, lub dopiero przy najbliższym sygnale aktywującym wysyłanie danych.
- Oczekiwanie przed wysłaniem. Ten tryb używany jest do zapisania danych, które zostaną wysłane dopiero po zmianie trybu na inny.

Dane odczytane przez interfejs są wpierw zapisywane do lokalnego stosu, a dopiero potem wysyłane do procesora. Podobnie jak dla wysyłania danych, umożliwia to szybsze i programistycznie łatwiejsze działanie procesora. Procesor zarządza odbieraniem danych, a do powiadamiania go o oczekujących na odczytanie informacji służy przerwanie procesora. Interfejs nie ma wbudowanych opcji wyboru sposobu wysyłania przerwania. Indywidualny dla interfejsu procesor zajmuje się ich interpretacją, po czym może odczytać dane ze stosu poprzez wybranie odpowiedniego rejestru.

Na rysunku 40 przedstawiono budowę ogólną interfejsów. Wszystkie interfejsy zawierają ten sam moduł obsługi pamięci FIFO. Pozostałe rejestrów, odpowiedzialne za poszczególne ustawienia modułów komunikacji, są w wydzielonym module, unikalnym dla danego protokołu. Na dalszych rysunkach dla uproszczenia przedstawiono je jako jeden moduł, a ten podział na dwa moduły służy uproszczeniu projektowania dodatkowych protokołów komunikacji.

8.1.1 Opracowanie pamięci interfejsów

Interfejsy używają niewielkiej, parametryzowej pamięci. Działa ona w oparciu o wskaźnik odczytu, który zawiera adres odczytywanego bitu, oraz analogiczny wskaźnik zapisu. Każdy z nich podlega inkrementacji w przypadku odczytania bądź zapisania danych. Te działania wykonują sygnały wejściowe odpowiednio `i_read` i `i_write`. Mogą być odczytane tylko wcześniej zapisane bajty. Do komunikacji z innymi modułami wykorzystywane są sygnały `o_full` i `o_empty`, informujące odpowiednio o zapełnieniu pamięci i o pustej pamięci. W celu realizacji różnych trybów pracy interfejsów komunikacja działa w dwóch trybach:



Rysunek 40. Schemat ogólny działania interfejsów. Opracowanie własne.

- Tryb ciągły buforu. Działa jak pamięć FIFO, w której po odczytaniu wszystkich bajtów, wskaźnik odczytu się zeruje, przez co ponowne odczytanie danych. Powiadomiony zostaje interfejs, dzięki czemu jest możliwe wstrzymanie przesyłania danych aż do momentu otrzymania przerwania.
- Tryb jednorazowy buforu. W tym przypadku działa ona jak bufor cykliczny, w którym po zapisaniu wszystkich bajtów, wskaźnik zapisu zeruje się, przez co umożliwia ponowne zapisanie już odczytanych danych. Analogicznie wskaźnik odczytu zeruje się po przeczytaniu wszystkich danych, umożliwiając ponowne przeczytanie ponownie zapisanych bajtów.

8.2 Opracowanie interfejsu UART

Protokół komunikacji UART jest używany do komunikacji pomiędzy dwoma urządzeniami. Używa on dwóch sygnałów:

- TX- sygnał transmisji. Zawiera dane wychodzące z urządzenia.
- RX- sygnał odbioru. Zawiera dane wchodzące do urządzenia.

Oba sygnały mają identyczne właściwości. Jednak jeden z nich jest wysyłany, a drugi odbierany, co wymaga dwóch niezależnych modułów. Na rysunku 41 przedstawiono wartość sygnałów protokołu UART.

Można wyszczególnić kilka kolejno wykonywanych etapów transmisji:

- Stan bezczynny. Sygnał jest równy 1.
- Rozpoczęcie transmisji. Sygnał jest równy 0.
- Transmisja danych. Sygnał przybiera wartość zależną od transmitowanych danych. Bity danych transmitowane są od najwyższego do najniższego.
- Zatrzymanie transmisji. Sygnał przybiera wartość 1.

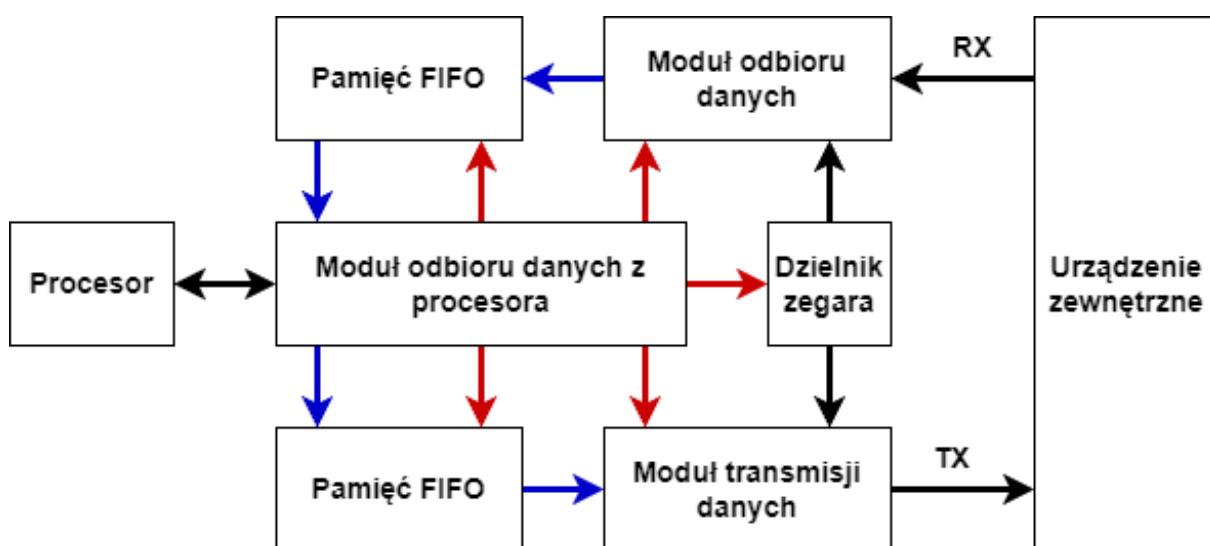
Bezczynny	Start	8 bitów danych	Stop
-----------	-------	----------------	------



Rysunek 41. Przebieg sygnału protokołu UART. Opracowanie własne.

8.2.1 Implementacja modułów transmisji

Sygnal wysyłany i odbierany ma identyczne właściwości i parametry, ale obie czynności są obsługiwane przez niezależne moduły, co przedstawiono na rysunku 42.



Rysunek 42. Budowa interfejsu UART. Opracowanie własne.

Oba moduły są zaimplementowane za pomocą maszyny stanów, która przechodzi pomiędzy poszczególnymi etapami transmisji. Te etapy są wspólne dla obu modułów, ale inaczej zaimplementowane. To, co moduł wysyłania wysyła, moduł odbioru musi wykryć.

Cykły zegara są odliczane wspólnie dla obu modułów przez dzielnik zegara. Każdy z nich sam zlicza przychodzące do niego impulsy. Jeden bit jest transmitowany w czasie 16 impulsów. Gdy osiągnięta zostaje ta wartość, układ albo przechodzi do kolejnego stanu albo wysyła lub odczytuje kolejny bit danych.

8.2.2 Implementacja modułów wysyłania

Moduł wysyłania rozpoczyna transmisję w momencie, gdy jego sygnał wejściowy `i_start` przyjmuje wartość 1. W większości przypadków dzieje się tak wtedy, gdy pamięć danych do transmisji nie jest pusta. W takiej sytuacji sygnał wyjściowy przyjmuje wartość 0, a sygnał `o_read_tx_fifo` przyjmuje wartość 1, dzięki czemu wskaźnik pamięci lokalnej jest inkrementowany, co umożliwia późniejsze przeczytanie kolejnego bajtu. Dopiero w następnym stanie wysyła najniższy bit danych, po czym przesuwa zawartość pamięci wysyłanych danych o jeden bit w prawo. Te czynności są powtarzane aż do wysłania wszystkich bitów danych.

W przypadku, w którym bit parzystości jest używany, układ wysyła policzony w trakcie transmisji bit tego typu. Jest on liczony za pomocą funkcji logicznej XOR między samym sobą a przesyłanym bitem danych, z wartością początkową zależną od jego parzystości bądź nieparzystości.

Następnie sygnał przybiera wartość 1, czym sygnalizuje zakończenie transmisji danych. Długość trwania tego stanu jest definiowana przez użytkownika i może wynosić między 16 a 32 impulsów dzielnika zegara, co odpowiada od 1 do 2 okresów wysyłania bitu. Zakończenie transmisji jest sygnalizowane za pomocą sygnału o_tx_done.

8.2.3 implementacja modułu odbioru

Po wykryciu rozpoczęcia transmisji, czyli wykryciu wartości 0 na sygnale RX, układ rozpoczyna odliczanie impulsów z dzielnika zegara. Gdy wykryje ich 16, rozpoczyna odczytywanie kolejnych bitów danych, które są zapisywane do najniższego bitu rejestru, który następnie jest przesuwany w lewo. Dzięki czemu pierwszy odebrany bit będzie najwyższy spośród odebranych danych.

Jeżeli bit parzystości jest używany, moduł sprawdza, czy odebrana wartość zgadza się z wartością policzoną w trakcie odbioru danych w analogiczny sposób, jak przy wysyłaniu. W momencie, gdy jest zgodny z tą wartością, po zakończeniu odbierania zakończenia transmisji, sygnał o_rx_done przyjmuje wartość 1, umożliwiając zapisanie odebranych danych do pamięci FIFO.

8.2.4 Opcje interfejsu

Protokół ma kilka powszechnie wykorzystywanych opcji, które procesor musi wybrać przed rozpoczęciem transmisji. Są to:

- Bod protokołu. Jest to miara prędkości przesyłania bitów protokołu. Określa, ile bitów danych może zostać przesłanych w ciągu jednej sekundy. Najczęściej wykorzystywane wartości to: 9600, 19200, 38400, 115200 i wyższe. Do interfejsu należy wysłać 8 bitową wartość zdefiniowaną jako 1/16 liczby cykli zegara przypadających na wysyłany bit. Z tego względu należy wysłać wartość określona na podstawie poniższego wzoru.

$$Wartosc = \frac{f_{zegara}}{16*bod}$$

- Bit parzystości. Jest on używany w celu wykrywania błędu transmisji w przypadku zmiany wartości pojedynczego transmitowanego bitu. Jest on umieszczony bezpośrednio po ramce danych. Bit może być parzysty lub nieparzysty i zależy od tego, czy będzie on równy 1 w przypadku parzystej czy nieparzystej liczby jedynek w przesyłanej wiadomości. W celu jego ustawienia należy ustawić dwa dolne bity specjalnego rejestru.
- Liczba bitów danych. Istnieje możliwość zdefiniowania ile bitów danych ma być w pojedynczej transmisji. Jest to przydatne jeżeli chcemy transmitować litery zapisane w kodzie wymagającym tylko 7 bitów. Nie zaimplementowano rzadko używanej 9 bitowej transmisji, ale jest możliwe wysłanie 8 bitów danych, a po czym bit parzystości. Powszechnie używa się od 5 do 9 bitów danych.
- Liczba bitów zatrzymania określa to, ile bitów jest wymaganych aby wykryć koniec transmisji. Powszechnie używa się 1, 1,5 lub 2 bity zatrzymania. W celu wybrania jednej z tych wartości należy do specjalnego adresu wysłać liczbę od 0 do 2.

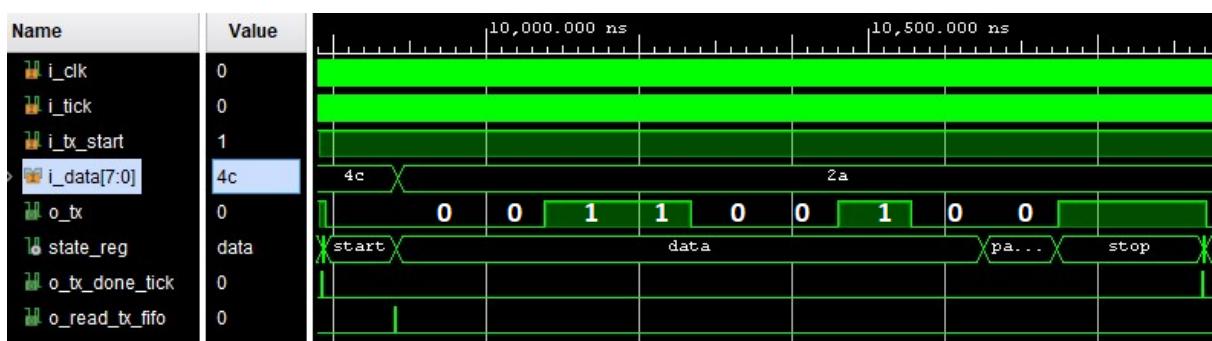
- Sprzętowa kontrola przepływu jest czasami używana w celu sprawdzenia czy dane zostały prawidłowo odebrane. Ze względu na małą popularność tej opcji, która wymaga dodatkowych portów zewnętrznych, nie została ona zaimplementowana.

8.2.5 Symulacje interfejsu UART

Przeprowadziłem serię symulacji każdej funkcjonalności interfejsu, co umożliwiło poprawienie licznych błędów przed testowaniem interfejsu na układzie FPGA.

8.2.5.1 Symulacje modułu wysyłania

Na rysunku 43 przedstawiony jest przebieg wybranych sygnałów modułu wysyłania sygnału TX. Wprowadzone ustawienia to 8 bitów danych, parzysty bit nieparzystości, 2 bity zatrzymania, jeden bit na 64 cykle zegara. Dane do wysłania to 0x4C. Bit nieparzystości powinien przyjąć wartość 0, ponieważ wysyłana liczba ma nieparzystą liczbę bitów o wartości 1.



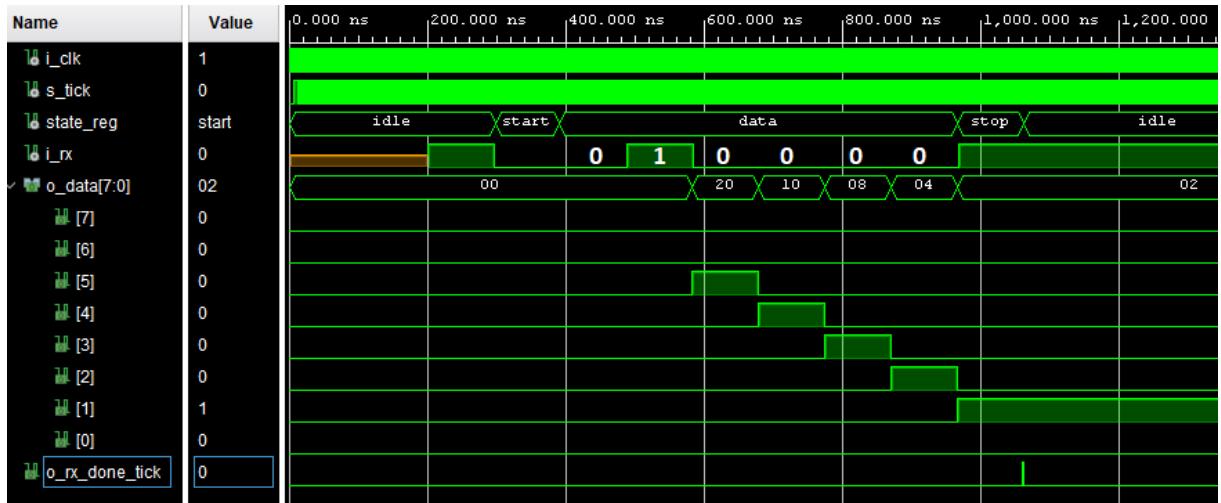
Rysunek 43. Symulowany przebieg sygnałów modułu wysyłania UART. Opracowanie własne.

Zgodnie z wcześniejszą określonymi wymaganiami, sygnał *o_tx* przyjmuje wartość 0 sygnalizując rozpoczęcie transmisji, po czym zaczyna przesyłać kolejne bity danych. Przesłał dane 00110010. Zgodnie ze specyfikacją protokołu są one odwrócone. Odpowiadają one przesyłanej liczbie 0x4C. To właśnie ona znajduje się na porcie *i_data* zawierającym dane do wysłania. Obliczony bit nieparzystości wyniósł 0. Zakończenie sygnału zajmuje dwukrotnie więcej, niż przesłanie jednego bitu danych, co oznacza, że przesłano dwa bity zatrzymania. Sygnał *o_tx_done_tick* został ustawiony po wysłaniu całej ramki komunikacji, umożliwiając zapisanie danych. Po rozpoczęciu transmisji został ustawiony sygnał *o_read_tx_fifo*, umożliwiając odczytanie kolejnych danych z buforu. Widać, że ten sygnał zadziałał, ponieważ sygnał *i_data* zmienił swoją wartość.

8.2.5.2 Symulacje modułu odbioru

Na rysunku 44 przedstawiony jest przebieg wybranych sygnałów modułu odbierania sygnału RX. Wprowadzone ustawienia to 6 bitów danych, brak bitu parzystości, 1 bit zatrzymywania, jeden bit na 64 cykle zegara. Odbierane dane pochodzą z symulacji i zawierają liczbę 2, która w reprezentacji binarnej jest równa 000010.

Sygnał został prawidłowo odczytany, ponieważ na porcie *o_data* w momencie ustawienia bitu *o_rx_data_tick* są dane 0x02. To właśnie wtedy dane z portu są zapisywane do pamięci. Widać, że sygnał *o_data* ma zmienioną wartość, w której nowo odczytany bit jest zapisywany

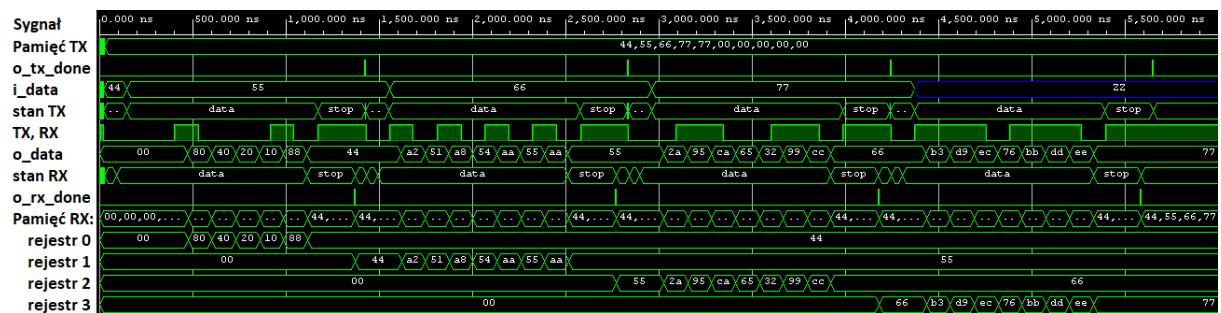


Rysunek 44. Symulowany przebieg sygnałów interfejsu UART. Opracowanie własne.

jako górny bit danych, po czym dane są obracane w lewo. Po pierwszym spadku sygnału wejściowego *i_rx* widać prawidłowe wykrycie warunku startu.

8.2.5.3 Symulacje interfejsu

Najlepszym sposobem na sprawdzenie działania modułu jest połączenie sygnału wysyłanego i odbieranego tak, aby dane wpierw wysyłane były odczytywane przez drugi moduł. W ten sposób zawartość pamięci wysyłanych danych powinna być skopiowana do pamięci odbieranych danych. Dane wpisane do pamięci wysyłania to 0x44, 0x55, 0x66 i 0x77. Powinny być one wysłane, odebrane, po czym zapisane do pamięci odbieranych danych. Na rysunku 45 przedstawiono wycinek tej symulacji. Zgodnie z oczekiwaniemi, wysłane zostały wszystkie 4 bajty wpisane do pamięci TX, a następnie odebrane i wpisane do pamięci RX, co pokazuje, że układ działa poprawnie.



Rysunek 45. Symulowany przebieg wybranych sygnałów interfejsu UART. Opracowanie własne.

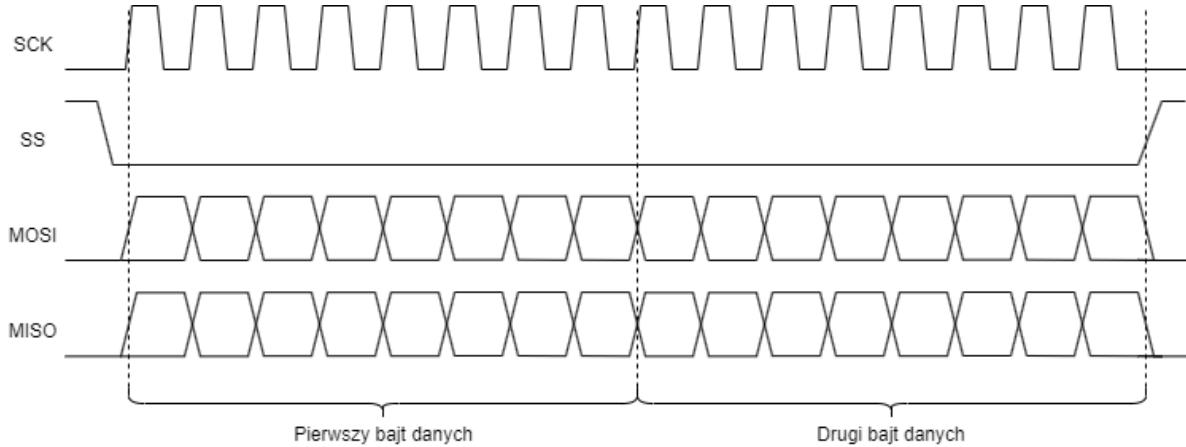
8.3 Opracowanie interfejsu SPI

Protokół komunikacji SPI (ang. Serial Peripheral Interface) jest powszechnie używany do przesyłania danych pomiędzy mikrokontrolerem a urządzeniami zewnętrznymi. Używa on co najmniej 4 sygnały:

- SCK (ang. Serial Clock)- sygnał zegara. Określa częstotliwość transmisji danych.

- CS (ang. Chip Select)- sygnał wyboru urządzenia zewnętrznego. Wymagane jest tyle tego typu sygnałów, ile jest urządzeń, z którymi komunikuje się danych interfejs.
- MOSI (ang. Master out, Slave in)- sygnał wysyłanych danych. Zawiera dane wychodzące z urządzenia.
- MISO (ang. Master in, Slave out) - sygnał odbioru danych. Zawiera dane wchodzące do urządzenia.

Na rysunku 46 przedstawiona jest wartość sygnałów protokołu SPI.



Rysunek 46. Przebieg sygnału protokołu SPI. Opracowanie własne.

Dane mogą być wysyłane i odbierane jednocześnie lub po sobie, w zależności od specyfikacji obsługiwanej urządzenia zewnętrznego. Częstym przypadkiem jest przesłanie jednego bajtu sygnałem MOSI, po czym odebranie drugiego sygnałem MISO. To, który sygnał SS jest wybrany, zależy od zawartości rejestru definiującego adres, który musi zostać zapisany przed rozpoczęciem transmisji.

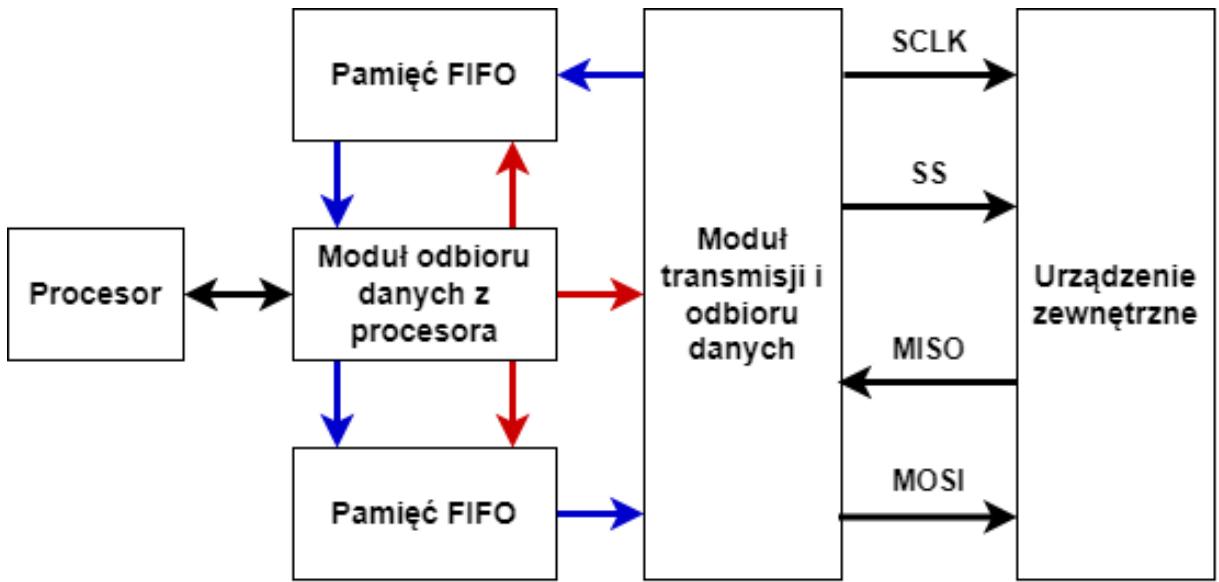
8.3.1 Opcje interfejsu

Protokół ma kilka powszechnie wykorzystywanych opcji, które są wymagane do wybrania przez procesor przed rozpoczęciem transmisji. Są to:

- Polaryzacja zegara. (ang. clock polarity), w skrócie CPOL. Określa wartość zegara w stanie spoczynku.
- Faza zegara (ang. clock phase), w skrócie CPHA. Określa to, na którym zboczu zegara dane są wysyłane i odbierane.
- Częstotliwość zegara. Określa, ile bitów danych może zostać przesłanych w ciągu jednej sekundy. Może przybierać różne wartości, ale nie większe, niż połowa częstotliwości zegara, co jest w zupełności wystarczające dla komunikacji. Do specjalnych rejestrów należy przesłać 16 bitową wartość zgodnie ze wzorem:

$$Wartosc = \frac{f_{zegara}}{2*f_{sclk}}$$

Na rysunku 47 przedstawiono schemat opisujący działanie interfejsu SPI.



Rysunek 47. Schemat interfejsu SPI. Opracowanie własne.

8.3.2 Implementacja modułu transmisji danych

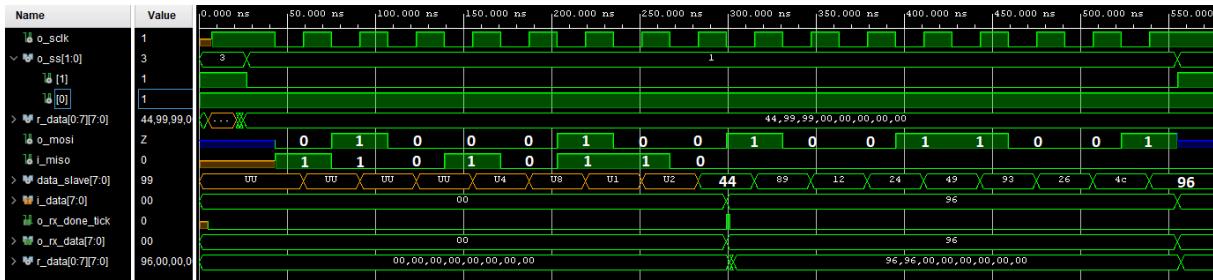
Jako moduł transmisji i odbioru danych SPI używany jest zmodyfikowany moduł transmisji danych autorstwa Scott'a Larson'a z firmy DIGI-KEY[55]. Zawiera on wszystkie niezbędne funkcjonalności i pozwolił na zmniejszenie nakładu pracy. Wymagane było dodanie obsługi sygnałów odczytu i zapisu pamięci oraz dostosowanie sygnałów wejściowych.

8.3.3 Symulacje interfejsu

W celu sprawdzenia interfejsu zdecydowałem się na podłączenie go w symulatorze do drugiego, innego interfejsu SPI działającego jako urządzenie zewnętrzne typu niewolnik (ang. slave). Ten moduł jest wykorzystywany jedynie do testów, dlatego jest oparty na internetowym poradniku[56], co pozwoliło pewniej zweryfikować jego poprawność, minimalizując szanse na błąd symulacji.

Do pamięci wysyłanych danych zostaną zapisane wartości 0x44 i 0x99. Te dane powinny być wysłane przez testowany układ master SPI po czym odebrane przez układ slave SPI i widoczne na jego wyjściu `data_slave`. Ma on adres 1 co oznacza, że reaguje na sygnały tylko wtedy, gdy drugi bit sygnału SS przyjmuje wartość 1. Dodatkowo po rozpoczęciu transmisji wyśle on dane 0x96, które powinny zostać odczytane przez interfejs i zapisane do pamięci FIFO. Na rysunku 48 zostały przedstawione wybrane symulowane sygnały. Polaryzacja jest ustaliona na dodatnią, a faza zegara na przeciwną. Dla takich ustawień sygnały powinny zmieniać swoją wartość tylko dla opadających zboczy zegara[57].

Widac na nim, że sygnał MOSI przyjmuje wartości 01000100 i 10011001. W reprezentacji szesnastkowej są to 0x44 i 0x99, co odpowiada oczekiwaniom. Sygnał `data_slave` przyjmuje te wartości po zakończeniu wysyłania danych, co pokazuje, że są one wysyłane poprawnie. Po zainicjowaniu transmisji urządzenie zewnętrzne wysyła dane równe 0x96, które są poprawnie zapisywane do pamięci odebranych danych. Drugi bit sygnału SS przyjmuje wartość 0 w trakcie transmisji. Zgodnie z oczekiwaniami sygnał SCLK w trakcie spoczynku ma wartość 1, a sygnały MOSI i MISO zmieniają swoją wartość tylko na jego opadającym zboczu.



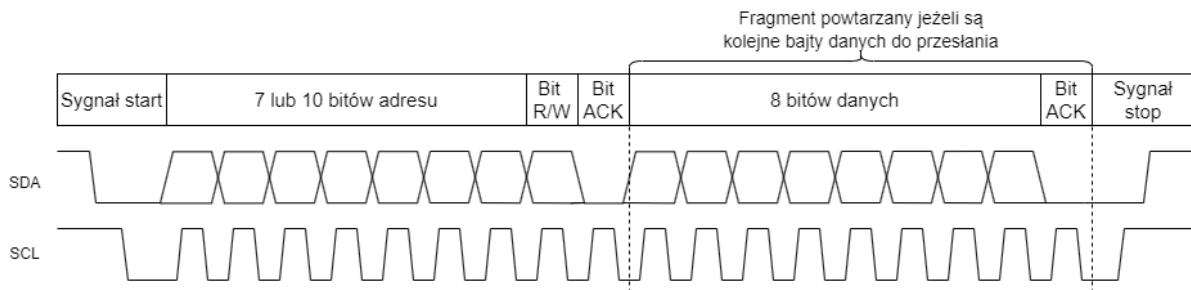
Rysunek 48. Przebieg wybranych sygnałów symulacji interfejsu SPI. Opracowanie własne.

8.4 Opracowanie interfejsu I^2C

Protokół komunikacji I^2C jest powszechnie wykorzystywany do komunikacji z wieloma adresowalnymi urządzeniami zewnętrznymi. Kontroler steruje komunikacją pomiędzy różnymi targetami, z których każdy ma unikalny adres. Używane są dwa sygnały:

- SDA - linia danych. Jest obsługiwana zarówno przez kontroler jak i target.
- SCL - linia zegara. Dane są pobierane na jego rosnącym zboczu i nie mogą się zmienić do opadającego zbocza.

Transmisja składa się z kilku etapów, które przedstawiono na rysunku 49.

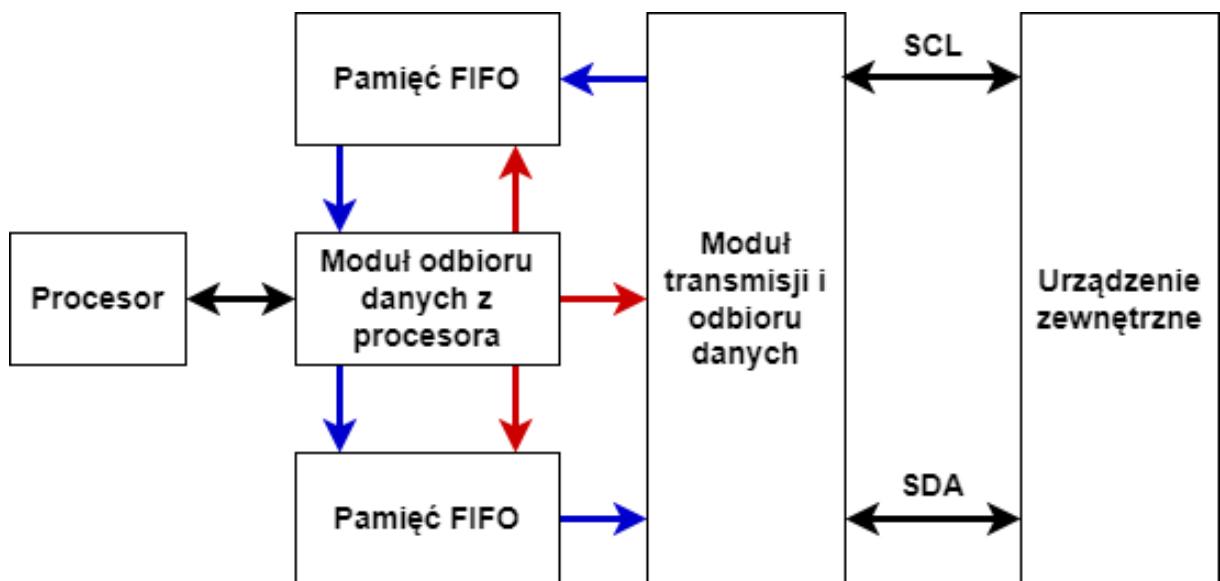


Rysunek 49. Przebieg sygnałów protokołu I^2C . Opracowanie własne.

- Sygnał start- Sygnał SDA jako pierwszy przyjmuje wartość 0, dopiero po nim sygnał spada sygnał SCL. Sygnalizuje to rozpoczęcie transmisji.
- Adres urządzenia- 7 bitowy adres
- Bit R/W- bit odczytu lub zapisu. Ostatni bit bajtu adresu zawiera informacje o tym, czy należy wysłać czy odczytać dane.
- Bit ACK- Potwierdzenie odebrania danych. Jeżeli target poprawnie odczytał swój adres, przejmuje kontrolę na sygnalem SDA wysyłając 0. To samo następuje po każdym bajcie danych i sygnalizuje ich odebranie. To samo robi kontroler w przypadku, w którym to on odczytuje dane.
- Bajt danych- może być wysłany przez kontroler lub target w zależności od bitu R/W. Ten etap, wraz z bitem ACK są powtarzane tyle razy, ile bajtów należy wysłać.
- Sygnał stop- sygnał SCL jako pierwszy przyjmuje wartość 1, dopiero po nim rośnie sygnał SDA. Sygnalizuje to zakończenie transmisji.

Przed odczytaniem danych należy zdefiniować to, ile bajtów danych odczytujemy. Służy do tego 8-bitowy rejestr. Dane będą wysyłane przez target tak długo, jak kontroler będzie potwierdzał odbieranie danych. Odbierane dane są liczone, a po osiągnięciu zdefiniowanej liczby urządzenie zaprzestało ich odczytywania. Niepotwierdzenie poprzez bit ACK skutkuje przerwaniem wysyłania danych przez target. Ten 8-bitowy rejestr jest niezbędny do odczytywania i będzie ono zachodziło tylko po zapisaniu do niego wartości różnej od zera. Zapisanie zera oznacza tryb wysyłania danych.

Adres targetu jest zapisywany do specjalnego 10-bitowego rejestru przez DPU i nie jest przechowywany w pamięci FIFO, w której znajdują się tylko dane. Ze względu na możliwość wysyłania danych w pętli zaimplementowano dwa sposoby jej działania. Po wysłaniu adresu dane z pamięci mogą być wysyłane w pętli lub po każdej pętli może być ponownie wysłany adres targetu. Implementacja interfejsu jest przedstawiona na rysunku 50



Rysunek 50. Schemat interfejsu I^2C . Opracowanie własne.

8.4.1 Opcje interfejsu

Protokół komunikacji I^2C ma dwa parametry:

- Częstotliwość- Sygnał SCL dowolną częstotliwością do 100kHz w trybie standardowym lub do 400kHz w trybie przyspieszonym[29]. Jest ona ustalana za pomocą 16 bitowego rejestru. Target dostosowuje swoją częstotliwość nadawania do sygnału SCL. Nie może przekroczyć połowy częstotliwości zegara. Zapiswaną wartość wyznacza się ze wzoru:

$$Wartosc = \frac{f_{zegara}}{2*f_{SCL}}$$
- Szerokość adresowania- Adresowanie może być 7 bitowe lub 10 bitowe. W przypadku rzadziej używanego 10 bitowego adresowania do przesłania adresu używane są dwa bajty zgodnie ze rysunkiem 51. Używany adres jest zapisywany do 16 bitowego rejestru. 10 bitowy adres jest używany, gdy ustawiony jest najwyższy bit tego rejestru.



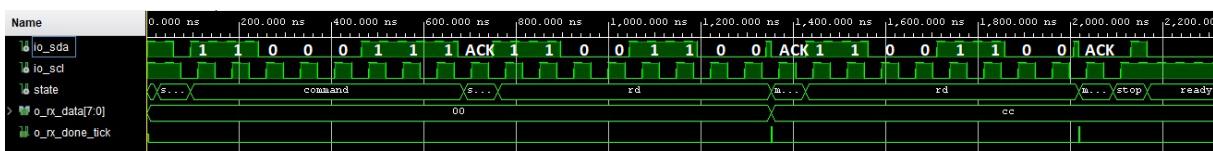
Rysunek 51. Schemat obrazujący wysyłanie rozszerzonego adresu. Opracowanie własne.

8.4.2 Implementacja modułu transmisji danych

Jako moduł transmisji danych I^2C używany jest zmodyfikowany moduł transmisji danych autorstwa Scott'a Larson'a z firmy DIGI-KEY [58]. Zawiera on obsługę sygnałów SPI i SCK i pozwolił na zmniejszenie nakładu pracy. Niezbędne było dodanie wielu funkcjonalności, takich jak obsługa sygnałów odczytu i zapisu pamięci, obsługa 10-bitowych adresów i zliczanie odczytywanych danych. Jednak najbardziej kluczowe odczytywanie i wysyłanie danych nie zostały zmienione.

8.4.3 Symulacje interfejsu I^2C

W celu sprawdzenia działania interfejsu I^2C najlepiej podłączyć je do symulowanego urządzenia zewnętrznego obsługującego ten protokół komunikacji. Zdecydowałem się na użycie gotoowego modułu autorstwa Petera Samarina[59]. Adres urządzenia jest zdefiniowany jako 0x63, który w reprezentacji binarnej jest równy 11000011 Symulowane są dwa warianty. Odczytania i zapisania danych. Dla łatwiejszego przedstawienia danych jest to przedstawione na dwóch symulacjach. W symulacji odczytu, przedstawionej na rysunku 52, liczbę odczytywanych rejestrów zdefiniowano jako 2. Dane, które powinien wysłać target to 0xCC.



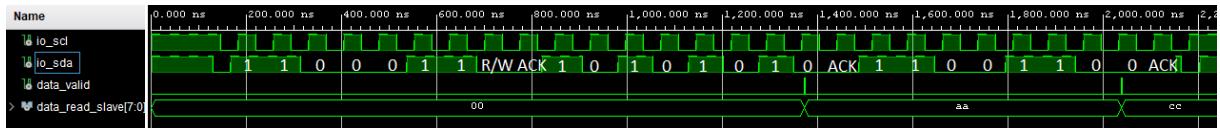
Rysunek 52. Przebieg wybranych symulowanych sygnałów interfejsu I^2C . Opracowanie własne.

Wpierw sygnał SDA spadł, po nim sygnał SCL sygnalizując rozpoczęcie transmisji. Pierwszy wysłany bajt zawiera adres i bit R/W. Wysyłany adres to 1100011 i zgadza się z adresem targetu. Po nim następuje wysyłany przez target sygnał ACK. Po nim są wysyłane przez target dwa bajty danych. Oba są równe 1001100, czyli 0xCC. Zgadza się to z oczekiwaniami. Po pierwszym bajcie danych kontroler wysyła sygnał ACK. Po drugim nie jest on wysyłany, sygnalizując zakończenie odczytywania danych. Na koniec następuje wpierw wzrost wartości SCL, po czym wzrost SDA sygnalizując zakończenie transmisji.

Odczytane bajty są widoczne na sygnale `o_rx_data`. Wartość sygnału zgadza się z danymi wysłanymi przez target. Sygnał `o_rx_done_tick` sygnalizuje zakończenie odczytywania danych i jest używany do zapisu wartości do pamięci.

W symulacji zapisu dane zapisane do pamięci wysyłania to 0xx44 i 0x99. To właśnie je powinien wysłać kontroler i przedstawia to rysunek 53.

Bit R/W przyjmuje wartość 1 oznaczając wysyłanie danych. Ponownie, jak podczas symulacji odczytu danych, pierwszy wysłany bajt zawiera adres i bit R/W. Bit R/W przyjmuje wartość 0 oznaczając wysyłanie danych. Drugi wysłany bajt to 10101010 czyli 0x44 w reprezentacji



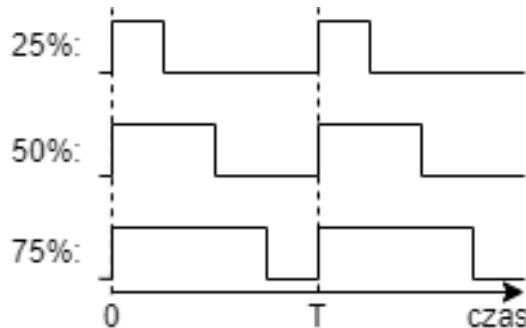
Rysunek 53. Przebieg wybranych symulowanych sygnałów interfejsu I^2C . Opracowanie własne.

szesnastkowej. Po nim jest wysyłany przez target sygnał ACK, bajt 100110 czyli 0x99 i sygnał ACK.

Odebranie danych widać na sygnale data_read_slave, w którym po każdym przesłanym bajcie pojawia się właśnie odczytana przez target wartość. Zgodnie z oczekiwaniami pojawiają się w nim wysłane wartości co pokazuje poprawność działania sygnału. Sygnał data_valid informuje poprawnym odczytaniu.

8.5 Opracowanie interfejsu PWM

Sygnal PWM (ang. Pulse-width modulation) to sposób przesyłania wartości w formie sygnału prostokątnego o różnej długości. Może on przesyłać wartość od 0 do 1, modulując współczynnikiem wypełnienia sygnału. Jest to stosunek długości impulsu do okresu sygnału. W celu lepszego zobrazowania go na rysunku 54 przedstawiono 3 sygnały o różnych współczynnikach wypełnienia. Sygnał PWM używany jest do przesyłania wartości analogowych za pomocą sygnału cyfrowego.



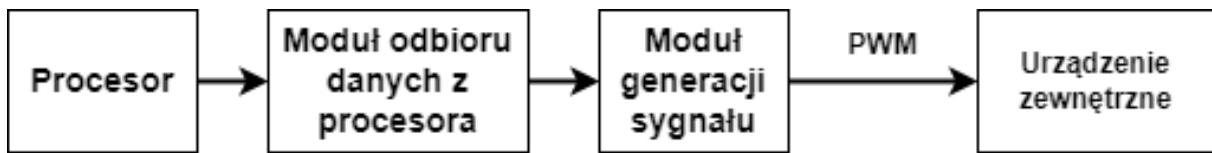
Rysunek 54. Przebieg sygnałów PWM dla różnych współczynników wypełnienia. Opracowanie własne.

Ten interfejs jako jedyny nie ma zaimplementowanego odbierania danych i nie używa żadnych funkcjonalności używanych przez pozostałe interfejsy. Zamiast tego używa dwóch adresowalnych 16-bitowych rejestrów:

- Okres sygnału. Zapisana do niego wartość oznacza, ile okresów zegara mieści się w jednym okresie sygnału. Definiuje on częstotliwość.
- Długość impulsu. Zapisana do niego wartość oznacza ile okresów zegara mieści się w jednym impulsie sygnału. Definiuje on współczynnik wypełnienia.

Tymi dwoma rejestrami można ustawić praktycznie dowolną częstotliwość i współczynnik wypełnienia. Jest on równy stosunkowi wartości zapisanych do nich. Im mniejsza częstotliwość sygnału, tym większa jest jego rozdzielczość. Długość impulsu nie może być większa od długości sygnału. W przypadku wpisania niepoprawnych wartości, wysyłany jest sygnał ciągły równy

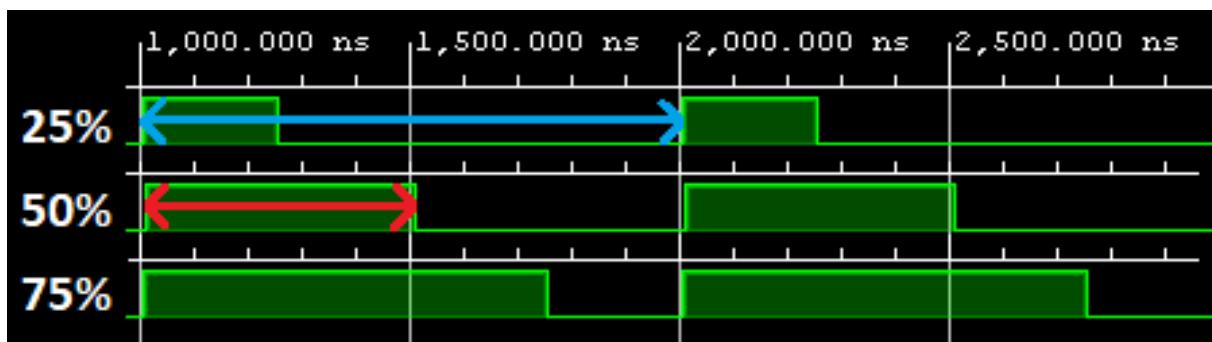
1. Sygnał jest generowany po ustawieniu bitu w specjalnym adresowalnym rejestrze, a przestaje po jego wyczyszczeniu. Schemat działania interfejsu został przedstawiony na rysunku 55.



Rysunek 55. Schemat interfejsu PWM. Opracowanie własne.

8.5.1 Symulacje interfejsu PWM

Na rysunku 56 przedstawione są symulacje działania sygnału PWM dla różnych wartości rejestrów. Okres zegara ustawiono na 1 ns, wartość rejestru okresu sygnału na 1000, wartość rejestru długości impulsu ustawiono na kolejno 250, 500 i 750 dla trzech symulacji. Oznacza to, że współczynniki wypełnienia powinny wynosić kolejno 25%, 50% i 75%. Niebieską strzałką zaznaczono okres sygnału, a czerwoną długość impulsu.

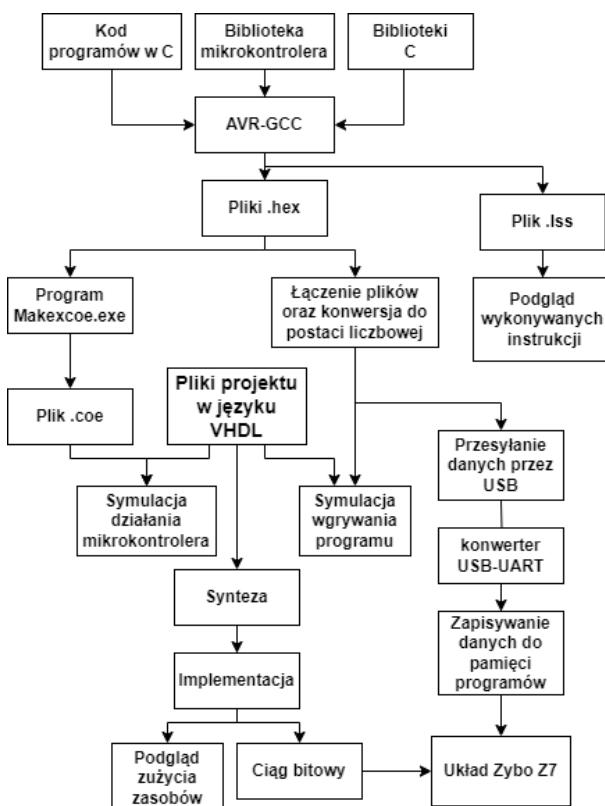


Rysunek 56. Symulowane sygnały PWM dla różnych wartości. Opracowanie własne.

Symulowane脉冲具有对应于250, 500和750ns的脉冲长度，而周期为1000ns。这意味着填充因子等于25%，50%和75%，这与预期一致，表明该设计正确。

Rozdział 9: Realizacja symulacji i testów

Na rysunku 57 przedstawiono podział na poszczególne etapy używane do testowania projektu. W tym rozdziale wytłumaczono każdy z nich tak, aby lepiej przedstawić poszczególne testy.



Rysunek 57. Schemat przedstawiający podział na poszczególne etapy wymagane do testowania projektu. Opracowanie własne.

9.1 Biblioteka mikrokontrolera

Język C można wykorzystać do programowania mikrokontrolera tylko przy zastosowaniu niezbędnej biblioteki. W projekcie użyto biblioteki bazującej na tej, która została dostarczona przez Atmel Corporation dla mikrokontrolerów ATmega. Zapewnia ona najbardziej podstawowe funkcjonalności i zawiera się w plikach:

- `sfr_defs.h` Dostarcza on definicje używane do określania adresów przez kompilator.
- `common.h` Zawiera adresy rejestrów specjalnych.
- `interrupt.h` Odpowiada on za obsługę przerwań.

- pgmspace.h Zawiera funkcje umożliwiające obsługę pamięci danych procesora.

Dzięki wykorzystaniu tych gotowych plików wystarczy, że w autorskim pliku uc_addresses.h są zawarte wszystkie adresy, przerwania i stałe używane przez mikrokontroler.

W celu rozszerzenia funkcjonalności języka C użyto biblioteki stdint.h. Definiuje ona wiele niezbędnych typów danych. Biblioteka util/delay.h zawiera funkcje pozwalające na opóźnienia działania procesora przez określony czas.

Autorskie biblioteki functions_cpu.h i functions_dpu.h zawierają wiele funkcji które stanowią pewnego rodzaju interfejs pomiędzy programistą a adresami mikrokontrolera. Dzięki temu programista nie musi się zastanawiać nad znaczeniem adresów i ich wykorzystaniem. Przykładowo wystarczy użyć funkcji PWM_write_data(uint16_t duty, uint16_t freq) aby ustawić działanie PWM. Inaczej programista musiałby zdefiniować wpisanie danych do 5 rejestrów. Dzięki temu programowanie nie wymaga dokładnej znajomości wszystkich rejestrów, a wystarczy tylko kilka zasad.

Biblioteka functions_cpu.h zawiera następujące funkcje używane przez procesory DPU:

- connect_interface(uint8_t dpu_number, uint8_t interface_number)
- disconnect_interface(uint8_t dpu_number, uint8_t interface_number)
- dpu_interrupt()
- add_dpu_from_cpu_transmision(uint8_t dpu)
- remove_dpu_from_cpu_transmision(uint8_t dpu)
- clear_dpu_from_cpu_transmision()
- send_data_from_cpu_to_interface(uint8_t data)
- send_4_bytes_from_cpu_to_interface(uint32_t data)
- set_gpio(uint8_t gpio)
- clear_gpio(uint8_t gpio)
- set_led(uint8_t led)
- clear_led(uint8_t led)
- select_data_to_cpu_from_interface(uint8_t dpu)
- send_data_to_1_dpu(uint8_t dpu, uint8_t data)
- allow_mes_from_dpus()
- send_data_to_user(char data[], uint8_t len)
- send_byte_to_user(uint8_t data)

Biblioteka functions_dpu.h zawiera następujące funkcje używane przez procesory DPU:

- send_byte_to_interface(uint8_t data)

- send_data_to_cpu(uint8_t data)
- I2C_initiation(uint8_t adres, uint16_t freq)
- I2C_read()
- I2C_write()
- SPI_initiation(uint8_t adres, uint8_t freq)
- PWM_write_data(uint32_t duty, uint32_t freq)
- PWM_disable()
- Interface_initiation()
- UART_initiation(uint8_t baud_rate)

9.2 Zalecane sposoby wykorzystania procesora przez programistę

Mikrokontroler został zaprojektowany ze wzrokiem na konkretne rozwiązania programistyczne i umożliwia ich implementacje. Alternatywne rozwiązania są czasami możliwe, ale nie są tutaj omówione. Nie zawsze są one optymalne ze względu na pozorne skomplikowanie prostych czynności, ale zostały one wybrane dla ułatwienia projektowania mikrokontrolera.

9.2.1 Zalecenia dotyczące DPU

Ze względu na konieczność synchronizacji z CPU, jedynie inicjalizacja powinna przebiegać bez wcześniejszego otrzymania wektora przerwań. Reszta funkcji powinna przebiegać w przerwaniu. Na listingu 4 przedstawiono przykładową implementację funkcji przerwania. Jej zadaniem jest odczytanie danych przesyłanych przez procesor i w zależności od nich uruchomienie odpowiedniego interfejsu. Dzięki temu jest możliwe napisanie wspólnego programu dla każdego rdzenia DPU.

```

1 ISR(INTER_MESS_FROM_CPU){
2     uint8_t data_from_cpu = SELECT_CPU_DATA;
3     switch (data_from_cpu) {
4         case PWM:
5             PWM_implementaction();
6             break;
7         case UART:
8             UART_implementaction();
9             break;
10        case default:
11            send_data_to_cpu(WRONG_VALUE);
12    }
13    sei();
14 }
```

Listing 4. funkcja DPU odbierająca daną z CPU

Po wysłaniu danych do CPU, dane są zapisywane do specjalnego rejestrów mikrokontrolera. W tym momencie czekają na ich odebranie przez CPU, które, jeżeli to umożliwiło za pomocą funkcji allow_mes_from_dpus(), otrzymuje przerwanie. Może się zdarzyć sytuacja, w której

kilka DPU w tym samym momencie, lub w chwili przed umożliwieniem przerwań, wyśle przerwanie do CPU. W takiej sytuacji DPU dostaje wektor przerwania `INTER_MESS_TO_CPU_NOT_SENT`. W momencie umożliwienia otrzymania przerwania przez CPU, DPU dostaje przerwanie `CPU_ALLOW_MES_FROM_DPUS`, na co powinno zareagować ponowną próbą wysłania danych. Te próby za pomocą tych dwóch przerwań powinny być powtarzane aż do skutku. W przypadku jednoczesnego wysłania przerwania przez 2 lub więcej DPU, CPU otrzymuje przerwanie z DPU o najmniejszym numerze. Implementację wysyłania danych za pomocą tych wektorów przerwań przedstawiono na listingu 5

```
1 int main(void){
2     send_data_to_cpu(data);
3     sei();
4     while(1);
5 }
6 ISR(INTER_MESS_TO_CPU_NOT_SENT){
7     // Ewentualna reakcja na nie wysłanie danych do CPU
8     sei();
9 }
10
11 ISR(INTER_MESS_ABLE_TO_SENT){
12     send_data_to_cpu(data); //Ponowna proba wyslania danych do CPU
13     sei();
14 }
```

Listing 5. funkcja DPU komunikująca się z CPU

9.2.2 Zalecenia dotyczące CPU

Podczas inicjalizacji CPU powinno wpierw połączyć interfejsy z DPU, po czym poinformować je o tym. Zajmuje się tym funkcja przedstawiona na listingu 6.

```
1 int main(void){
2     connect_interface(dpu, interface)
3     send_data_to_1_dpu(dpu, data);
4     allow_mes_from_dpus();
5     while(1);
6 }
```

Listing 6. funkcja CPU komunikująca się z DPU

Później CPU może analizować informacje przychodzące od DPU i na ich podstawie wysyłać wiadomość do użytkownika. Na listingu 7 przedstawiono funkcję przerwania wykonywaną przez CPU po otrzymaniu danych z DPU nr. 0. Funkcje odbioru danych od pozostałych DPU są analogiczne.

```
1 ISR(INTER_MESS_DPU_0){
2     data = DPU0;
3     allow_mes_from_dpus();
4     sei();
5 }
```

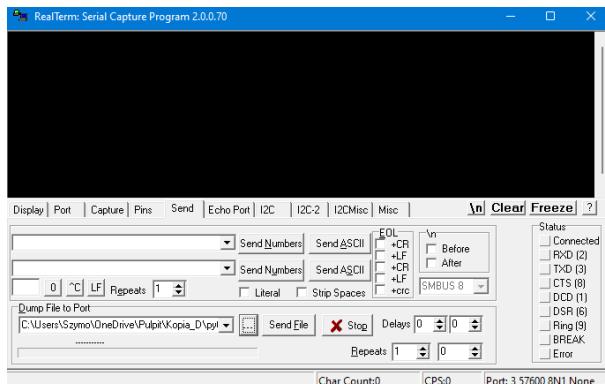
Listing 7. funkcja CPU odbierająca dane z DPU

9.2.3 Realizacja przesyłania oprogramowania do procesora

Oprogramowanie wyjaśniono w rozdziale 9. Do komplikacji użyto kompilator AVR-GCC. Kompilacja działa też na kompilatorze X8, ale nie tworzy on plików zawierających listę instrukcji wgrywanych do pamięci programów (format .lss), które są niezbędne do śledzenia działania procesora w celu jego debugowania.

Kompilator również tworzy plik formatu intel hex (.hex), których zawartość oraz przetwarzanie jest wyjaśnione w rozdziale 5.3.2.1. Na wczesnym etapie testów, przed utworzeniem modułu wgrywania programu, były one wgrywane do pamięci programu jako plik typu .COE. Są one używane przez program Vivado do inicjalizowania bloków pamięci. By je utworzyć użyto minimalnie zmodyfikowanego programu MAKEXCOE.exe autorstwa WD Peterson'a. dla Silicore Corporation. Jednak ten proces wymagał ponownego utworzenia bloków pamięci, które nie rzadko pozostawały niezmienione na skutek błędów programu Vivado. Z tego względu większość symulacji było przeprowadzanych z czasochlonnym wgrywaniem programu przez symulowany interfejs UART. Te kilka minut symulacji były szybsze i pewniejsze, niż zapisywanie pamięci ROM.

Zmodyfikowane pliki HEX w formacie liczbowym wgrywano przez komputer za pomocą programu RealTerm. Umożliwia on przesyłanie i odczytywanie danych za pomocą portu USB. Umożliwia przesyłanie całych plików oraz ustawienie dowolnej częstotliwości sygnału. W przeciwieństwie do konkurencyjnego programu TeraTerm, umożliwia on przesyłanie danych zarówno w formacie ASCII, jak i binarnych. Na rysunku 58 przedstawiono widok programu w trakcie przesyłania przez niego danych.



Rysunek 58. Wygląd programu RealTerm

Dane z komputera trafiają do konwertera USB-UART, który zamienia dane z protokołu USB na UART, pozwalając na odczytanie danych przez mikrokontroler bez konieczności zaprojektowania modułu obsługującego USB.

9.3 Realizacja projektu w języku HDL dla FPGA

Projekt mikrokontrolera zawiera 33 pliki napisane w języku opisu sprzętu VHDL. Może on modelować zachowanie i strukturę systemów cyfrowych na wielu poziomach abstrakcji, od poziomu systemu do poziomu bramek logicznych, w celu projektowania i weryfikacji. Taki podział pozwala na myślenie o poszczególnych, niezależnie testowanych modułach, a nie całym mikrokontrolerze który jest zbyt skomplikowany, by dało się o nim myśleć tylko na poziomie pojedynczych

sygnałów. W celu uproszczenia opisu działań arytmetycznych użyto powszechnie wykorzystywanej biblioteki Numeric_std.

Ilość procesorów i interfejsów obsługiwanych przez układ FPGA jest w pełni parametryzowalne. Oznacza to, że mikrokontroler może mieć dowolną liczbę poszczególnych interfejsów i procesorów, w zależności od zapotrzebowania. Jedyne ograniczenie wprowadza moduł wyboru interfejsów, który obsługuje do 32 interfejsów i do 16 procesów, jednak podczas dotychczasowych testów nie zaistniała potrzeba przekroczenia tych wartości.

Do symulacji poszczególnych części projektu zastosowano 19 różnych plików. W większości przypadków po przetestowaniu jednej funkcji był on używany do przetestowania innej funkcji tego samego modułu. Wszystkie symulacje mikrokontrolera wykonano za pomocą tego samego pliku. Różnił się jedynie program, którego wgrywanie było przez niego symulowane. W efekcie działanie każdego używanego modułu został osobno zasymulowany. Są to, w kolejności alfabetycznej, następujące pliki:

- Binary_Counter.vhd
- Binary_Counter.vhd
- CPU_ALU.vhd
- CPU_Core.vhd
- CPU_data_path.vhd
- CPU_exec_logic.vhd
- CPU_instruction_set.vhd
- CPU_opc_deco.vhd
- CPU_opc_fetch.vhd
- CPU_prog_mem.vhd
- CPU_reg16.vhd
- CPU_registers.vhd
- CPU_status_regs.vhd
- FIFO.vhd
- full_uc.vhd
- I2C_and_interface.vhd
- I2C_master.vhd
- I2C_module.vhd
- Int_selection.vhd
- Interface.vhd
- INTERFACE_regs.vhd
- prog_mem_module.vhd
- prog_mem_writing.vhd
- PWM_master.vhd
- PWM_module.vhd

- spi_and_interface.vhd
- SPI_master.vhd
- SPI_module.vhd
- uart_and_interface.vhd
- UART_module.vhd
- UART_RX.vhd
- uart_rx_2.vhd
- UART_TX.vhd

Odpowiadające im pliki symulacji:

- ALU_TB.vhd
- Binary_Counter_TB.vhd
- CPU_TN.vhd
- Deco_TB.vhd
- EXEC_TB.vhd
- FIFO_TB.vhd
- full_uc_TB.vhd
- I2C_TB.vhd
- I2C_slave.vhd
- Int_selection_TB.vhd
- opc_fetch_TB.vhd
- prog_mem_module_TB.vhd
- prog_mem_TB.vhd
- prog_mem_writing_TB.vhd
- PWM_TB.vhd
- registers_TB.vhd
- RX_TB.vhd
- SPI_TB.vhd
- TX_TB.vhd
- UART_TB.vhd

Dodatkowo wykorzystywano konfigurowalne bloki IP:

- temp_prog_mem
- Regs_mem
- CPU_program_mem
- CPU_data_mem
- clk_wiz_0

9.3.1 Realizacja pliku głównego projektu

Na listingu 8 przedstawiono kod języku opisu sprzętu, który definiuje parametryzowaną liczbę procesorów DPU oraz towarzyszące im pamięci FIFO, które umożliwiają zapisanie przez CPU kolejki danych do DPU. Tutaj też znajduje się system generowania przerwań. Wektor przerwania wchodzący do procesora jest określany przez kod w linijkach od 37 do 41 na podstawie szeregu sygnałów, które służą wyłącznie w tym celu. W linijkach 34-35 określono dane wchodzące do DPU.

```

1 generate_dpu: for i in 0 to PROCESSORS-1 generate
2     DPU_core : entity work.CPU_core(Behavioral)
3     port map(
4         i_clk=> i_clk,
5         i_RST_pm=>rst_pm,
6         i_RST=> cpu_RST,
7         i_PROG_MEM_ADDR=>prog_mem_addr(11 downto 0),
8         i_PROG_MEM_WE=>prog_mem_we(1+i),
9         i_PROG_MEM_DATA=>prog_mem_data,
10        i_INTERRUPT_VEC => interrupt_vec(i),
11        i_DATA=> input_data_dpu(i),
12        o_OPC => opc_dpu(i),
13        o_PC => open,
14        o_DATA => output_data_dpu(i),
15        o_ADDR_IO(4 downto 0) => addr_dpu(i),
16        o_RD_IO => rd_dpu(i),
17        o_WE_IO => we_dpu(i),
18        o_INTER_DONE=>inter_done(i));
19
20    DPU_FIFO :entity work.FIFO(Behavioral)
21    port map(
22        i_CLK=> i_clk,
23            i_RST=>rst_pm,
24            i_LOOP =>'0',
25            o_RETURN => open,
26            i_WRITE => FIFO_write(i),
27            i_READ => FIFO_read(i),
28            o_FULL => open,
29            o_EMPTY => open,
30            i_DATA => FIFO_in_data(i),
31            o_DATA => FIFO_out_data(i)
32    );
33    we_rd_dpu(i) <= rd_dpu(i) or we_dpu(i);
34    input_data_dpu(i) <= data_from_mul_to_dpu(i) when addr_dpu(i) = SELECT_MUL_DATA
35        and rd_dpu(i) = '1' else
36            FIFO_out_data(i) when addr_dpu(i) = SELECT_CPU_DATA and rd_dpu(i) = '1';

```

```

36 FIFO_read(i) <= '1' when addr_dpu(i) = SELECT_CPU_DATA and rd_dpu(i) = '1' else
37   '0';
38   interrupt_vec(i) <= (others=>'0') when inter_done(i) = '1' or i_rst = '1' else
39     INTER_READ_INTERFACE when interrupt_interface_dpu(i) = '1' else
40     INTER_MESS_FROM_CPU when input_interrupt_main(i) = '1' else
41     INTER_MESS_TO_CPU_NOT_SENT when mess_not_sent(i) = '1' else
42     INTER_MESS_ABLE_TO_SENT when mess_able_to_send(i) = '1';
43   output_data_dpu_to_cpu(i) <= output_data_dpu(i) when addr_dpu(i) = WRITE_TO_CPU
44   and we_dpu(i) = '1' else x"00" when i_rst = '1';
45 end generate generate_dpu;

```

Listing 8. generacja procesorów DPU

Z kolei na listingu 9 przedstawiono zdefiniowanie pojedynczych bloków:

- CPU
- modułu wyboru interfejsów
- modułu komunikacji z użytkownikiem

Ten ostatni, ze względu na wspólny interfejs UART, realizuje również funkcję modułu odczytywania pamięci programu. Przy module CPU jest system obsługi i generacji przerwań. Linijka 43 realizuje podział mapy pamięci systemu na poszczególne procesory w zależności od górnych bajtów adresu, który pochodzi z modułu komunikacji z użytkownikiem.

```

1 CPU_core : entity work.CPU_core(Behavioral)
2 port map(
3   i_clk=> i_clk,
4   i_RST=> cpu_RST,
5   i_RST_PM=>rst_pm,
6
7   i_prog_mem_addr=>prog_mem_addr(11 downto 0),
8   i_prog_mem_we=>prog_mem_we(0),
9   i_prog_mem_data=>prog_mem_data,
10
11  i_INTERRUPT_VEC => interrupt_main_vec,
12  i_data => cpu_in_data,
13  o_OPC => opcode,
14  o_PC => pc,
15  o_Data => data_main,
16  o_ADDR_IO(4 downto 0) => addr_main,
17  o_RD_IO => cpu_rd,
18  o_WE_IO => cpu_out_we,
19  o_INTER_DONE=>inter_done_main);
20
21 Multiplexer : entity work.Int_selection(Behavioral)
22 port map(
23   i_clk=> i_clk,
24   i_RST=> cpu_RST,
25   i_data_CPU=>output_data_dpu, --data from dpu
26   o_data_TO_CPU=>data_from_mul_to_dpu, -- data to dpu
27   i_ADDR_CPU=>addr_dpu, --address from dpu
28   i_WE_MAIN=>cpu_out_we,
29
30   i_ADDR_MAIN=>addr_main,

```

```

31    i_data_main=>data_main,
32
33    i_data_from_int=>input_data_int,--data from interace to dpu
34    i_interrupt_int=>output_interrupt_int,--from interface
35    o_interrupt_cpu=>interrupt_interface_dpu,
36    i_we=>we_rd_dpu,
37    o_addr_int=>input_addr_int,
38    o_we_int=>we_int,
39    o_data_to_int=>output_data_int--data from dpu to interface
40  );
41
42 gen_pm_addr: for i in 0 to PROCESSORS generate
43   prog_mem_we(i) <= '1' when unsigned(prog_mem_addr(15+PROCESSORS downto 16)) = i and
44     prog_mem_we_bit = '1' else '0';
45 end generate gen_pm_addr;
46
47 prog_mem : entity work.prog_mem_module(Behavioral)
48 generic map(
49 BAUD_RATE => PROG_BAUD_RATE,
50 STOP_TICKS => PROG_STOP_TICKS
51 )
52 port map(
53   i_clk=>i_clk,
54   i_rst=>i_rst,
55   i_rx=>i_rx_pm,o_tx=>o_tx_pm,
56   o_mem_addr=>prog_mem_addr,
57   o_mem_wr=>prog_mem_we_bit,
58   o_mem_data=>prog_mem_data,
59   o_wait=>pc_wait,
60   i_data=>user_data,
61   i_wr=>user_write,
62   o_rx_data =>rx_data,
63   i_rx_read=>rx_read,
64   o_rx_not_empty=>rx_not_empty
65 );

```

Listing 9. generacja pojedyńczych modułów

Dodatkowo w tym pliku określono szereg mniejszych działań. Są to

- Dane wchodzące do CPU w linijkach od 2. do 5.
- Sygnał odczytania danych wysłanych przez użytkownika w linijce 6.
- Obsługa portów GPIO i LED w linijkach od 13. do 18.
- Obsługa przesyłania danych do DPU w linijkach od 19. do 37.

```

1
2  cpu_in_data <= x"00"  when cpu_rd = '0' else
3    rx_data  when addr_main = READ_RX_DATA_ADDRES else
4      output_data_dpu_to_cpu(to_integer(unsigned(addr_main))-1)
5      when unsigned(addr_main)<PROCESSORS+1 else x"00";
6  rx_read <= '1' when (cpu_rd = '1' and addr_main = READ_RX_DATA_ADDRES) else '0';
7    process (i_clk) begin
8      if rising_edge(i_clk) then

```

```

9      FIFO_write <= (others=>'0');
10     input_interrupt_main <= (others=>'0');
11     if cpu_out_we = '1' then
12       case addr_main is
13         when DPU_WRITE_GPIO=> -- xxxx_xxx gpio value, last bit for value
14           if to_integer(unsigned(data_main)) < GPIOs then
15             o_gpio(to_integer(unsigned(data_main(7 downto 1)))) <= addr_main(0);
16             end if;
17         when DPU_WRITE_LED=> --2 bit value
18           o_led(to_integer(unsigned(data_main(1 downto 0)))) <= addr_main(0);
19         when DPU_WRITE_SEL=>
20           if to_integer(unsigned(data_main)) < PROCESSORS then
21             dpu_data_sel(to_integer(unsigned(data_main)))<= '1';
22             end if;
23         when DPU_INTERRUPT =>
24           input_interrupt_main <= dpu_data_sel;
25         when DPU_WRITE_DATA=>
26           for i in 0 to PROCESSORS-1 loop
27             if dpu_data_sel(i) = '1' then
28               FIFO_in_data(i) <= data_main;
29               FIFO_write(i) <= '1';
30             end if;
31           end loop;
32         when DPU_WRITE_REMOVE=>
33           if to_integer(unsigned(data_main)) < PROCESSORS then
34             dpu_data_sel(to_integer(unsigned(data_main)))<= '0';
35           end if;
36         when DPU_WRITE_CLEAR=>
37           dpu_data_sel <= (others=>'0');
38         when others=>
39           end case;
40         end if;
41       end if;
42     end process;

```

Listing 10. generacja procesorów DPU

Na listingu 11 przedstawiono generacje interfejsów. Ich liczba jest parametryzowalna. Każdy interfejs ma określony numer i to na podstawie tego numeru CPU przypisuje interfejs do odpowiedniego DPU.

```

1  gen_uart: for i in 0 to UARTS-1 generate
2    uart_module: entity work.uart_and_interface(Behavioral)
3      port map(
4        i_clk=>i_clk,
5        i_rst=>i_rst,
6        i_interrupt =>'0',
7        i_reg_we =>we_int(i),
8        i_reg_addr =>input_addr_int(i),
9        i_reg_data=>output_data_int(i),
10       o_reg_data=>input_data_int(i),
11       o_intt=>output_interrupt_int(i),
12       i_rx=>i_rx(i), o_tx=>o_tx(i)
13     );
14   end generate gen_uart;

```

```

15
16 gen_i2c: for i in 0 to I2CS-1 generate
17     i2c_module: entity work.i2c_and_interface(Behavioral)
18         port map(
19             i_clk=>i_clk,
20             i_rst=>i_rst,
21             i_interrupt =>'0',
22             i_reg_we =>we_int(UARTS+i),
23             i_reg_addr =>input_addr_int(UARTS+i),
24             i_reg_data=>output_data_int(UARTS+i),
25             o_reg_data=>input_data_int(UARTS+i),
26             o_intt=>output_interrupt_int(UARTS+i),
27             io_sda=>io_sda(i),
28                 io_scl=>io_scl(i)
29         );
30     end generate gen_i2c;
31
32 gen_SPI: for i in 0 to SPIS-1 generate
33     spi_module: entity work.spi_and_interface(Behavioral)
34         port map(
35             i_clk=>i_clk,
36             i_rst=>i_rst,
37             i_interrupt =>'0',
38             i_reg_we =>we_int(UARTS+I2CS+i),
39             i_reg_addr =>input_addr_int(UARTS+I2CS+i),
40             i_reg_data=>output_data_int(UARTS+I2CS+i),
41             o_reg_data=>input_data_int(UARTS+I2CS+i),
42             o_intt=>output_interrupt_int(UARTS+I2CS+i),
43             o_sclk=>o_sclk(i),
44             o_ss=>o_ss(i),
45             i_miso=>i_miso(i),
46             o_mosi=>o_mosi(i)
47         );
48     end generate gen_SPI;
49
50 gen_PWM: for i in 0 to PWMS-1 generate
51     pwm: entity work.pwm_module(Behavioral)
52         port map(
53             i_clk=>i_clk,
54             i_rst=>i_rst,
55             i_interrupt =>'0',
56             i_reg_we =>we_int(UARTS+I2CS+SPIS+i),
57             i_reg_addr =>input_addr_int(UARTS+I2CS+SPIS+i),
58             i_reg_data=>output_data_int(UARTS+I2CS+SPIS+i),
59             o_pwm=>o_pwm(i)
60         );
61     end generate gen_PWM;

```

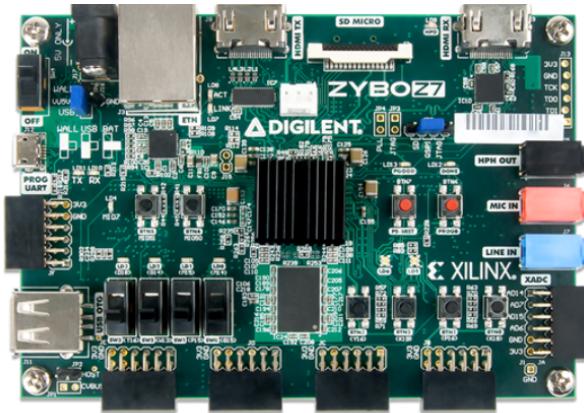
Listing 11. generacja interfejsów

9.3.2 Realizacje testów

9.3.2.1 Płytkę ewaluacyjną Zybo Z7-20

Do testowania działania na sprzęcie udostępniono została płytka ewaluacyjna Zybo Z7-20[60]. Użycie płytki ewaluacyjnej pozwala na zmniejszenie nakładu pracy poświęconego na projekt elektroniki. Umożliwia ona korzystanie z możliwości układu SOC i jest przeznaczona wyłącznie do testów. Na rysunku 59 przedstawiono wygląd tej płytki. Wykorzystywane elementy płytki to:

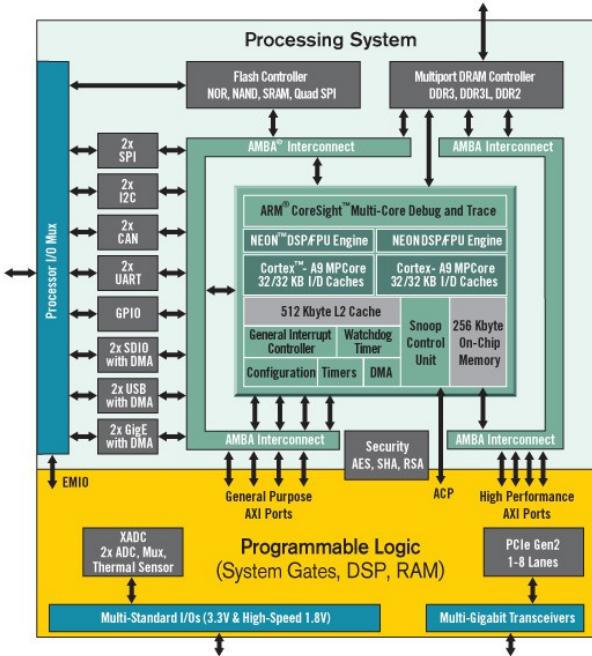
- układ zasilania
- przyciski
- złącza pinowe wykorzystywane do komunikacji
- port USB umożliwiający programowanie układu



Rysunek 59. Wygląd płytki ewaluacyjnej Zybo Z7-20[60]

Płytkę zawiera układ Zynq-7000, który oprócz używanego przez projekt układu FPGA, posiada wbudowany procesor Cortex-A9 oraz liczne złącza. Ten model układu FPGA zawiera 53200 bloków logicznych, 630KB blokowej pamięci RAM i 125 pinów zewnętrznych. To właśnie te parametry są kluczowe dla określenia maksymalnej liczby możliwych do użycia procesorów i interfejsów, ze względu na to, że to właśnie te zasoby są przez nie najczęściej wykorzystywane. Większość możliwości układu nie jest wykorzystywanych ze względu na nie korzystanie z wbudowanego procesora. Na rysunku 60 przedstawiono architekturę tego układu. Wykorzystywana jest jedynie logika programowalna, która na rysunku jest zaznaczona na żółto. Układ był symulowany i wgrywany za pomocą bezpłatnego programu Vivado, ze względu na jego dostępność i wsparcie dla używanego układu.

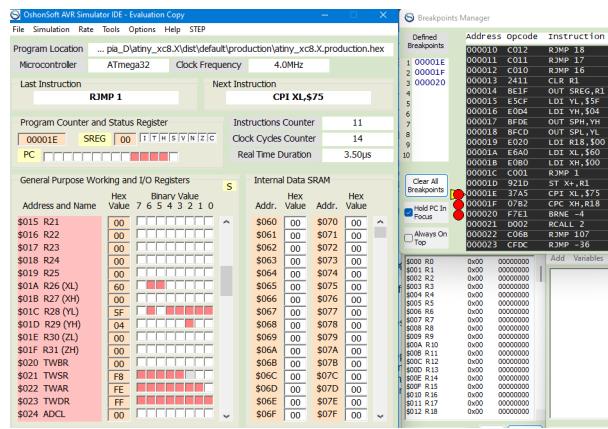
W przypadku większości testów używano zmniejszonego układu o czterech procesorach DPU i dwunastu różnych interfejsach. Pozwoliło to na zmniejszenie czasu tworzenia przez komputer tzw. bitstreamu wgrywanego na układ FPGA. Zegar musi działać z częstotliwością pozwalającą na przejście sygnału pomiędzy jego początkiem a końcem pomiędzy kolejnymi nastającymi zboczami zegara. Z syntez wynika, że maksymalne zaobserwowane opóźnienia sygnałów wynosiły około 15ns, dlatego zegar jest ustawiony 20ns. Taki zapas czasu sprawia, że implementacja będzie zrealizowana poprawnie, a dodatkowo umożliwia szybsze wyznaczenie przez komputer konfiguracji układu FPGA. Oznacza to, że częstotliwość działania układu wynosiła 50MHz.



Rysunek 60. Schemat przedstawiający architekturę układu Zynq-7000[60]

9.3.2.2 Symulacje działania programów

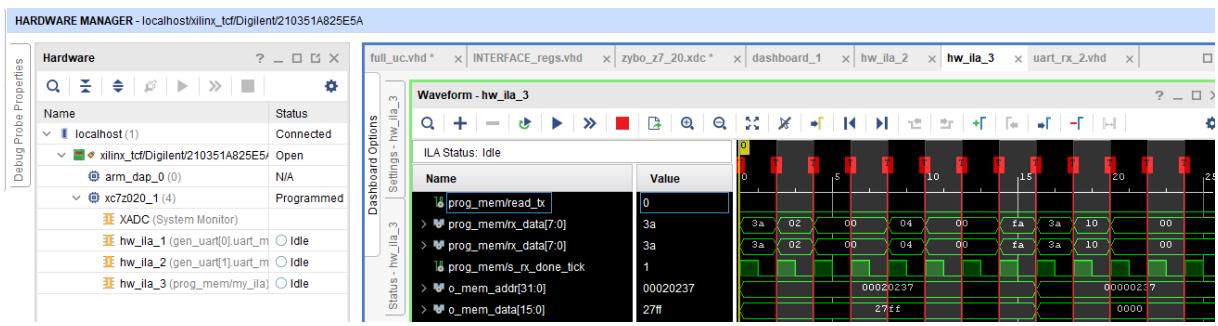
W trakcie niektórych zaawansowanych testów mikroprocesora, ze względu na złożoność programów, w celu określenia spodziewanego zachowania mikrokontrolera zastosowano program AVR simulator IDE.[61] Umożliwia on dokładną symulację programów instrukcję po instrukcji, wraz z czytelnym przedstawieniem zawartości rejestrów, których wartości można dowolnie edytować. Upraszczal on analizę instrukcji generowanych przez kompilator, dzięki czemu z większą pewnością skupiono się na wykrywaniu błędów. Na rysunku 61 przedstawiono wygląd programu w trakcie działania.



Rysunek 61. Program AVR simulator IDE w trakcie symulacji. Opracowanie własne.

9.3.2.3 Moduły ILA

W celu sprawdzenia wartości sygnałów wewnętrznych w trakcie działania systemu użyto modułu logiczne ILA (ang. Integrated Logic Analyzer). Rdzeń ILA zawiera wiele zaawansowanych funkcji nowoczesnych analizatorów logicznych, w szczególności możliwość wyzwalania logicznego.



Rysunek 62. Widok sygnałów modułu ILA w programie Vivado. Opracowanie własne.

Na rysunku 62 przedstawiono wygląd programu widzianego przez użytkownika. Widoczne sygnały odpowiadają wartościami określonych sygnałów wewnętrznych systemu zabezpieczeń. Dzięki niemu możliwe jest określenie przyczyny rozbieżności między symulacją a rzeczywistym działaniem układu, jak również odczytanie wartości wybranych sygnałów. Użyto następujące układy ILA:

- ila_uart
- ila_rx
- ila_cpu
- ila_spi
- ila_uart
- ila_prog_mem

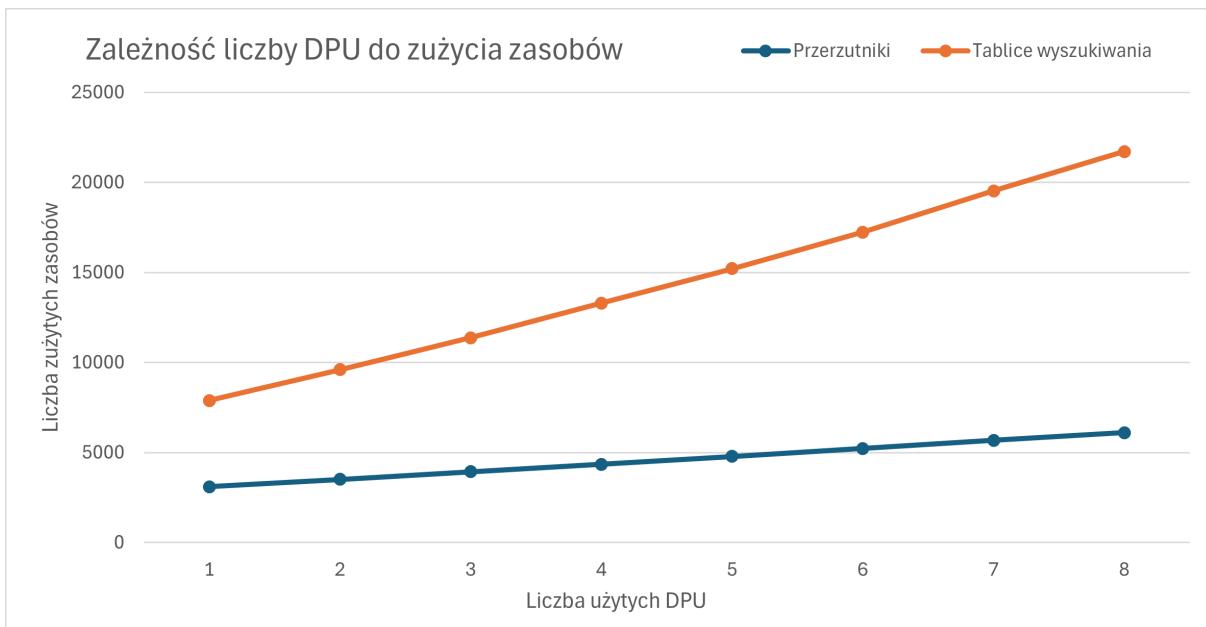
9.4 Zużycie zasobów układu FPGA

Opóźnienia pomiędzy poszczególnymi sygnałami zależą od używanych zasobów. Dzieje się tak ze względu na wydłużenie ścieżek sygnałów pomiędzy coraz dalej od siebie odległymi blokami logicznymi. Na rysunku 63 przedstawiono użycie tablic wyszukiwania LUT i przerutników. Zależności są zbliżone do liniowej ze względu na powielanie tych samych modułów, jednak wraz z ich wyczerpującym się zasobem implementacja logiczna staje się coraz mniej optymalna ponieważ używane jest coraz więcej zasobów na implementację tych samych funkcji logicznych. Liczbę możliwych do wykorzystania procesorów ogranicza liczba wbudowanych bloków pamięci RAM, a liczba interfejsów jest ograniczona przez liczbę odpowiednich portów zewnętrznych układu FPGA.

W tabeli 4 przedstawiono podział minimalnego zużycia zasobów przez poszczególne składowe mikrokontrolera. Moduły wgrywania programu, rejestrów oraz pamięci programu zużywają nieproporcjonalnie dużo zasobów w porównaniu do ich niewielkiego znaczenia i mogą one być dalej optymalizowane.

9.5 Częstotliwość zegara

Opóźnienia poszczególnych sygnałów zależą od tego, z ilu procesorów składa się mikrokontroler. Decydujące znaczenie dla częstotliwości zegara ma maksymalne opóźnienie sygnałów,



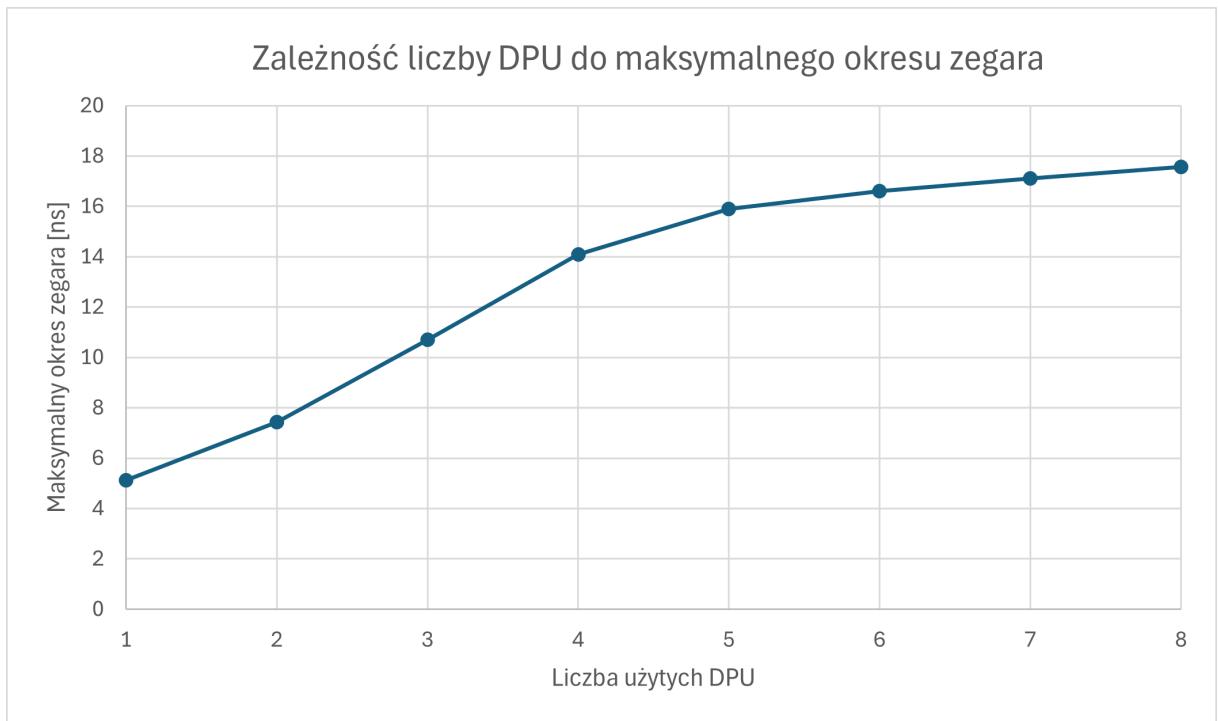
Rysunek 63. Zużycie wybranych zasobów układu FPGA przez mikrokontroler w zależności od liczby DPU. Liczba interfejsów jest stała. Opracowanie własne.

Nazwa modułu	LUT	Rejestry	Slice	Mux 7	Mux 8	Bloki RAM
Moduł komunikacji	1446	2161	679	6	0	3
UART TX	27	23	11	0	0	0
UART RX	22	23	12	0	0	0
Generator zegara UART	10	17	7	0	0	0
Wgrywanie programu	246	338	100	0	0	0.5
Pamięć FIFO	44	15	14	2	0	0.5
Interfejs I2C	146	135	58	1	0	0
I2C master	100	68	35	1	0	0
Interfejs SPI	87	101	39	0	0	0
SPI master	53	58	24	0	0	0
Interfejs PWM	40	49	21	0	0	0
Multiplexer	397	42	207	16	0	0
CPU	1436	414	456	52	16	4
Moduł wykonywania	631	282	305	42	16	2
Rejestry	574	280	275	41	16	0
Pamięć danych	0	0	0	0	0	2
Moduł ALU	58	0	54	1	0	0
Pobieranie instrukcji	178	41	95	0	0	2
Pamięć programu	178	24	91	0	0	2
Dekodowanie instrukcji	636	91	265	10	0	0

Tabela 4. Zużycie wybranych zasobów układu FPGA przez poszczególne, wybrane bloki mikrokontrolera. Mux to skrót oznaczający multiplekser, a występująca po nim liczba to liczba jego wejść. Opracowanie własne.

które nie może wynosić więcej, niż okres zegara. Na rysunku 64 przedstawiono zależność używanych procesorów od minimalnego okresu zegara. Jest on jedynie widoczny dla pierwszych dwóch procesorów DPU, a nie taki parametr nie jest przewidziany do praktycznego zastosowania. W trakcie testów używano okresu zegara 20ns. Oznacza to częstotliwość zegara 50MHz. Pozwalało to na bezproblemowe działanie układu dla różnych parametrów układu bez ryzyka

błędnej lub zbyt długiej syntezy.



Rysunek 64. Zależność maksymalnej częstotliwości zegara od liczby DPU dla stałej liczby interfejsów.
Opracowanie własne.

Rozdział 10: Realizacja testów sprzętowych

W celu sprawdzenia funkcjonalności mikrokontrolera na układzie FPGA wykonano szereg testów. Jednak dla zademonstrowania działania układu wybrano konfigurację, która kompletnie sprawdza wykorzystanie najważniejszych funkcji, a jednocześnie demonstruje potencjalne zastosowanie układu. Poprawne wykonanie skompilowanego programu pozwoli wykryć poprawne działanie poszczególnych instrukcji. Testowany układ opisano w rozdziale 9.3.1.

10.1 Cel testu

Celem programu jest obsługa kilku czujników. Każdy z nich jest obsługiwany za pomocą innego DPU. W przypadku, w którym zostanie wykryty na jednym z nich wzrost temperatury, brzęczyk obsługiwany przez kolejny procesor za pomocą interfejsu PWM zostanie uruchomiony. Wysyłana jest wtedy też informacja do użytkownika z aktualną temperaturą. PWM jest wykorzystywany jako alert bezpośredni. Cechuje się szybszym czasem transmisji niż przesłanie całego zdania za pomocą protokołu UART do użytkownika. Wypełnienie sygnału PWM ma odpowiadać temperaturze, co pozwala na alternatywny sposób odczytania temperatury. Poprzez podłączenie go do brzęczyka, wzrost temperatury powoduje wzrost głośności wydawanego dźwięku.

Jako termometry podłączono kilka różnych urządzeń, które cechują się różnymi sposobami odczytu danych. Są to:

- 2 czujniki MPU 6050
- 2 czujniki MPU 9250

Konfiguracja testu:

Schemat testu przedstawiono na rysunku 65. Test używa 5 DPU, z których każdy jest odpowiedzialny za obsługę innego urządzenia. Zaznaczony na rysunku mikrokontroler odpowiada układowi cyfrowemu wgranemu na układ FPGA.

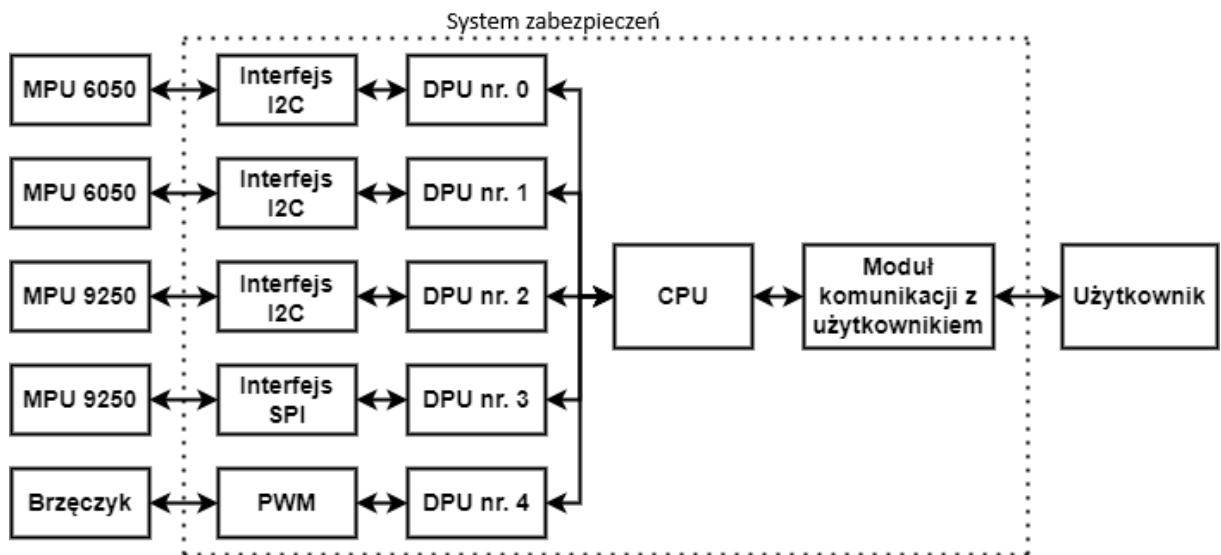
W celu odczytu temperatury wystarczy odczytać dwa rejesty zawierające dolny i górny bajt temperatury. Dla obu czujników są to rejesty odpowiednio 65 i 66. Tak odczytane rejesty są scalane do jednej, 16 bitowej wartości. Odpowiada ona temperaturze zgodnie ze wzorem[62][63]:

$$\text{Temperatura}^{\circ}\text{C} = \text{Wartosc}/340 + 36,53$$

Mimo braku obsługi liczb zmiennoprzecinkowych przez procesor, kompilator powinien prawidłowo przekształcić to równanie w kosztowny obliczeniowo ciąg instrukcji. Programista nie musi się tym zajmować.

W celu komunikacji pomiędzy CPU a DPU wykorzystywana jest ramka danych. Umożliwia to odczytanie kilku bajtów za pomocą jednego przerwania. Składa się ona z następujących kolejno elementów:

1. liczba pozostałych bajtów do odczytania



Rysunek 65. Schemat przeprowadzonego testu. Opracowanie własne.

2. rodzaj obsługiwanej przez DPU interfejsu
3. dane zależne od interfejsu

W skład testu wchodzą następujące etapy opisane w kolejnych sekcjach:

- Wgranie programu na układ FPGA
- Sprawdzenie poprawnego odczytywania danych z czujników
- Sprawdzenie działania sygnału PWM
- Sprawdzenie poprawnego wysyłania danych do użytkownika

10.2 Programy użyte podczas testu

Do przetestowania działania systemu potrzebne było jego zaprogramowanie. W tym celu powstały dwa programy. Jeden dla CPU, drugi dla wszystkich procesorów DPU umożliwiający obsługę różnych protokołów komunikacji.

10.2.1 Program CPU użyty podczas testu

CPU jest odpowiedzialne za inicjalizację poszczególnych DPU oraz komunikację z użytkownikiem. Do osiągnięcia pierwszego celu wykonuje on funkcję przedstawioną na listingu 12. Ustawa ona multiplekser podłączając DPU do interfejsu, po czym wysyła do każdego z nich o szeregu informacji, których znaczenie jest określone w opisie programu DPU. Okres sygnału PWM jest ustawiony na stałą wartość 100us. Początkowo sygnał jest wyłączony poprzez ustawienie zerowego wypełnienia.

```

1 int main(void){
2     // DPU 0
3     connect_interface(0, 2);
4     clear_dpu_from_cpu_transmision();
5     add_dpu_from_cpu_transmision(0);

```

```

6    send_data_from_cpu_to_interface(4);
7    send_data_from_cpu_to_interface(I2C);
8    send_data_from_cpu_to_interface(MPU6050_addr);
9    send_data_from_cpu_to_interface(2);
10   dpu_interrupt();
11   // DPU 1
12   connect_interface(1, 3);
13   clear_dpu_from_cpu_transmision();
14   add_dpu_from_cpu_transmision(1);
15   send_data_from_cpu_to_interface(4);
16   send_data_from_cpu_to_interface(I2C);
17   send_data_from_cpu_to_interface(MPU6050_addr);
18   send_data_from_cpu_to_interface(TempL_addr);
19   send_data_from_cpu_to_interface(2);
20   dpu_interrupt();
21   //DPU 2
22   connect_interface(2, 4);
23   clear_dpu_from_cpu_transmision();
24   add_dpu_from_cpu_transmision(2);
25   send_data_from_cpu_to_interface(4);
26   send_data_from_cpu_to_interface(I2C);
27   send_data_from_cpu_to_interface(MPU9250_addr);
28   send_data_from_cpu_to_interface(TempL_addr);
29   send_data_from_cpu_to_interface(2);
30   dpu_interrupt();
31   //DPU 3
32   connect_interface(3, 5);
33   clear_dpu_from_cpu_transmision();
34   add_dpu_from_cpu_transmision(3);
35   send_data_from_cpu_to_interface(4);
36   send_data_from_cpu_to_interface(I2C);
37   send_data_from_cpu_to_interface(MPU9250_addr);
38   send_data_from_cpu_to_interface(TempL_addr);
39   send_data_from_cpu_to_interface(2);
40   dpu_interrupt();
41   //DPU 4
42   connect_interface(4, 8);
43   clear_dpu_from_cpu_transmision();
44   add_dpu_from_cpu_transmision(4);
45   uint32_t pwm_duty = 0;
46   uint32_t pwm_freq = 5000;
47   send_data_from_cpu_to_interface(9);
48   send_data_from_cpu_to_interface(PWM);
49   send_4_bytes_from_cpu_to_interface(pwm_duty);
50   send_4_bytes_from_cpu_to_interface(pwm_freq);
51   dpu_interrupt();
52   allow_mes_from_dpus();
53   sei(); // enable global interrupts
54   while(1);
55 }
```

Listing 12. Program CPU użyty w trakcie testu

Do odbierania danych CPU wykorzystano przerwania. CPU wysyła temperaturę do użytkownika tylko wtedy, gdy temperatura jest wyższa od poprzedniej. Dodatkowo ustawia ona sygnał PWM na wypełnienie zbliżone do odczytanej temperatury. Ze względu na różne wektory

przerwań dla każdego DPU, potrzebne jest kilka minimalnie różnych funkcji przerwania. Tylko jedna z nich jest przedstawiona na listingu 13, ponieważ pozostałe są analogiczne. Zmienna max_temp jest wspólna dla wszystkich DPUs. Jest inicjalizowana jako 0. W przypadku, w którym jakiś czujnik wykryje większą temperaturę niż wcześniej wykryte jest ona aktualizowana

```

1 ISR(INTER_MESS_DPU_0){
2     uint8_t data;
3     data = DPU0;
4     if (data > max_temp) {
5         max_temp = data;
6         char tekst[] = "Maks temperatura wynosi %d C z interfejsu 0", max_temp;
7         send_data_to_user(tekst, strlen(tekst));
8         send_byte_to_user(data);
9
10        add_dpu_from_cpu_transmision(4);
11        uint32_t pwm_duty = max_temp*50;
12        uint32_t pwm_freq = 5000; // 100us/20ns
13        send_data_from_cpu_to_interface(9);
14        send_data_from_cpu_to_interface(PWM);
15        send_4_bytes_from_cpu_to_interface(pwm_duty);
16        send_4_bytes_from_cpu_to_interface(pwm_freq);
17        dpu_interrupt();
18    }
19    allow_mes_from_dpus();
20    sei();
21 }
```

Listing 13. funkcja odczytująca dane z DPU

Na listingu 14 przedstawiono funkcję przerwania wykonywaną w przypadku odebrania danych od użytkownika. Pozwala ona na odebranie jednego bajtu danych, ponieważ czas transmisji jest wielokrotnie większy niż czas wykonywania funkcji. Użytkownik może wysłać do CPU bajt, który zostanie odesłany z powrotem. Służy to wyłącznie zaprezentowaniu działania układu. Można również wysłać dane służące do czegoś przydatnego, np. ustawiające jakąś zmienną.

```

1 ISR(INTER_MESS_FROM_USER){
2     uint8_t data_from_user = SELECT_USER_DATA;
3     send_byte_to_user(data);
4 }
```

Listing 14. funkcja zwracające użytkownikowi wysłane dane

10.2.2 Program DPU użyty podczas testu

Odbieranie tych danych, jak i przesyłanie ich do interfejsów odbywa się w funkcji przerwania przedstawionej na listingu 15. DPU reaguje na dane wysłane przez CPU.

```

1 ISR(INTER_MESS_FROM_CPU){
2     uint8_t mes_len;
3     mes_len = SELECT_CPU_DATA;
4     mes_len = mes_len - 1;
5     uint8_t mes[16];
6     uint8_t i;
7     uint8_t adres;
8     uint16_t freq;
```

```

9     uint32_t pwm_duty, pwm_freq;
10    interface_select = SELECT_CPU_DATA;
11    i=0;
12    while (i<mes_len){
13        mes[i] = SELECT_CPU_DATA;
14        i++;
15    }
16    if (interface_select== UART){
17        UART_initiation(0x85);
18        i=1;
19        while (i<mes_len){
20            send_byte_to_interface(mes[i]);
21            i++;
22        }
23    }else if (interface_select == I2C){
24        adres = mes[1];
25        freq = 0x0B6E;
26        I2C_initiation(adres, freq);
27        read_addres1 = mes[2];
28        read_lenght = mes[3];
29        I2C_write();
30        send_byte_to_interface(read_addres1);
31        i=0;
32        while (i<read_lenght){
33            I2C_read();
34            i++;
35        }
36        read_byte_n = 0;
37    }else if (interface_select == SPI){
38        read_addres1 = mes[1];
39        read_addres2 = mes[2];
40        freq = 0x0B6E;
41        SPI_initiation(adres, freq);
42        i = 1;
43        send_byte_to_interface(read_addres1);
44        i++;
45    }else if (interface_select == PWM){
46        if (mes_len == 9) {
47            pwm_duty = mes[1]<<24||mes[2]<<16||mes[3]<<8||mes[4];
48            pwm_freq = mes[5]<<24||mes[6]<<16||mes[7]<<8||mes[8];
49            PWM_write_data(pwm_duty, pwm_freq);
50        }
51    }else{
52        send_data_to_cpu(ERROR);
53    }
54    sei();
55 }

```

Listing 15. funkcja DPU odbierająca daną z CPU

Po takim zaprogramowaniu DPU może on obsługiwać większość sytuacji i nie wymaga zmiany tej funkcji. W celu zmiany działania programu, by mógł obsługiwać inne czujniki, wystarczy zmienić program CPU oraz funkcję odczytywania danych. Po każdym odebraniu danych uruchamiana jest funkcja przerywania przedstawiona na listingu 16. Służy ona do odczytania danych oraz do ich analizy. W przypadku, w którym temperatura jest zbyt wysoka, wysyłana jest informacja do

CPU o aktualnej temperaturze.

```
1 ISR(INTER_READ_INTERFACE){
2     uint8_t mes_len;
3     uint8_t read_data = SELECT_MUL_DATA;
4     mes_len = SELECT_CPU_DATA;
5     if (interface_select == 1){
6         if (read_byte_n == 0){
7             read_byte_n = 1;
8             temperature = read_data;
9         } else {
10            read_byte_n = 0;
11            temperature = read_data<<8|temperature;
12            temperature2 = temperature2/340 + 36.53;
13            if (temperature > last_temperature+10 ){
14                uint8_t mes = temperature2;
15                send_data_to_cpu(mes);
16            }
17            last_temperature = temperature;
18            I2C_write(); //ponowne odczytywanie temperatury
19            send_byte_to_interface(read_addres1);
20            I2C_read();
21            I2C_read();
22        }
23    }else if (interface_select == 2){
24        if (read_byte_n == 0){
25            read_byte_n = 1;
26            temperature = read_data;
27            send_byte_to_interface(read_addres2);
28        } else {
29            read_byte_n = 0;
30            temperature = read_data<<8|temperature;
31            if (temperature > MAX_TEMPERATURE*100 ){
32                send_data_to_cpu(MESSAGE);
33            }
34            send_byte_to_interface(read_addres1);
35        }
36    }
37 }
```

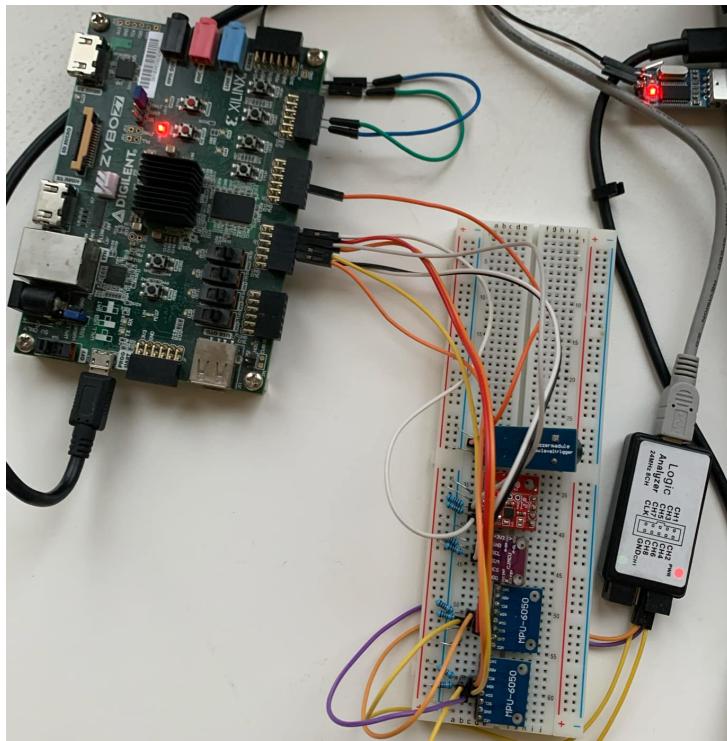
Listing 16. funkcja DPU odbierająca daną z interfejsu

10.3 Przeprowadzenie testu

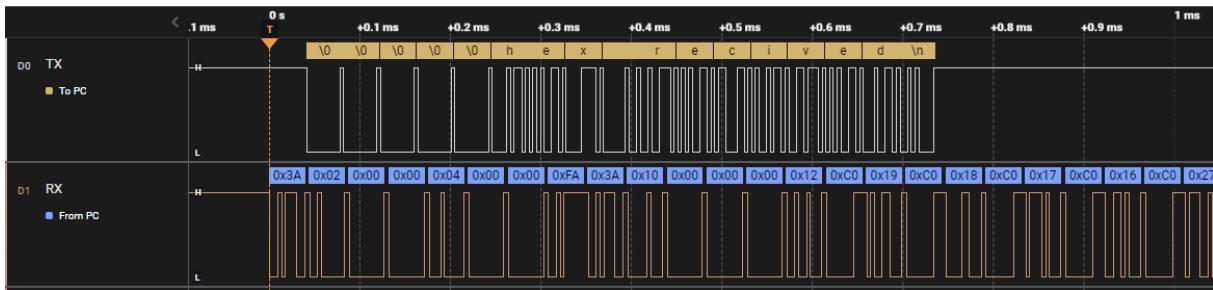
Na zdjęciu 66 przedstawiono układ elektroniczny w trakcie testu. Odzwierciedla on schemat 65. Na płytce stykowej widoczne są 4 czujniki. Są one zasilane z płytki ewaluacyjnej. Połączenia na płytce stykowej ułatwiając wpięcie analizatora logicznego oraz redukując liczbę połączeń. Widoczny jest też wpięty do komputera translator USB-UART.

10.3.1 Wgrywanie oprogramowania

Na rysunku 67 przedstawiono fragment przebiegu wgrywania oprogramowania. Są na nim widoczne dwa sygnały: wchodzący i wychodzący. Widać, że po rozpoczęciu wysyłania programu mikrokontroler wysyła wiadomość "HEX RECEIVED", sygnalizując poprawne odczytanie początku



Rysunek 66. Płytki ewaluacyjne podłączona do czujników. Opracowanie własne.



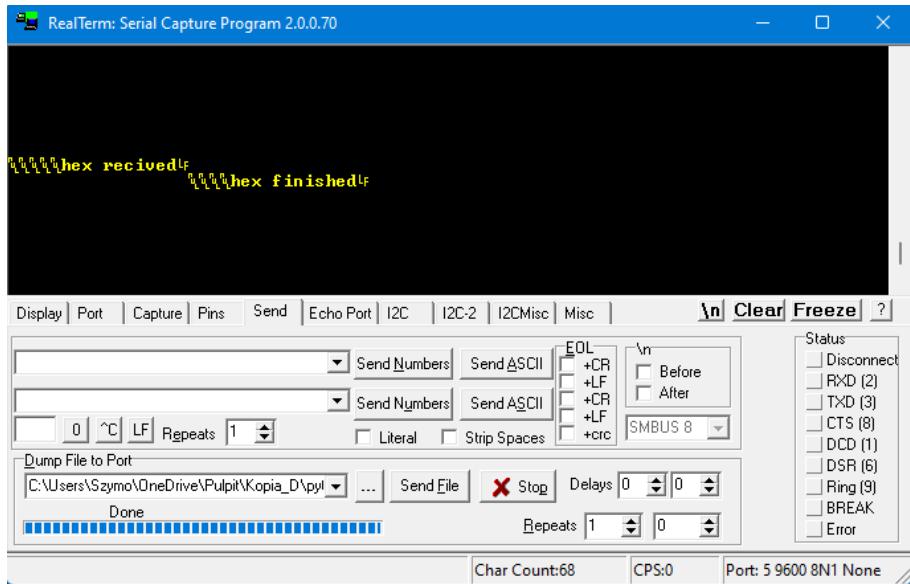
Rysunek 67. Przebieg sygnałów RX i TX podczas rozpoczęcia wgrywania programu. Opracowanie własne.

pliku. Na rysunku 68 przedstawiono widoczny dla osoby wgrywającej oprogramowanie widok. Transmisja programu używa fragmentów interfejsu UART. Transmisja modułu komunikacji różni się jedynie połączeniem poszczególnych modułów interfejsu między sobą. Z tego względu test transmisji można potraktować jako test interfejsu UART. Jest on rzadko używany do komunikacji z czujnikami, dlatego jest to jedyny sposób, w jaki przedstawiono działanie odbierania przez niego danych.

W trakcie wgrywania oprogramowania występował problem polegający na błędym odczytaniu kilku procent przesyłanych bajtów, w wyniku czego program nie był poprawnie transmitowany. Nie naprawiono modułu odbiornika UART, dlatego zdecydowano się na wykorzystanie gotowego, prostszego układu[64].

10.3.2 Obsługa czujników

Na rysunku 69 przedstawiono przebieg sygnałów trzech protokołów I^2C . Obsługują je 3 niezależne DPU, a transmisja do czujników przebiega równolegle. Odebrane dane z tych trzech czujników to kolejno 0x08F0, 0x0900 i 0x08F0. Odpowiadają one temperaturom 43,26, 43,31 i



Rysunek 68. Widok programu RealTerm po bezbłędnym wgraniu programu na układ FPGA. Opracowanie własne.

43,26 stopnia. Jest to wewnętrzna temperatura układu. Tak wysoka wartość wynika z nagrzania czujników. Zgodność pomiarów pokazuje ich poprawną pracę. Pracowały one znaczny czas tuż obok siebie, w identycznych warunkach.

Wpierw, po adresie czujnika, przesyłany jest numer pierwszego rejestru, z którego mają zostać odczytane dane. Następnie, po ponownym zainicjowaniu komunikacji, interfejs przesyła 2 odczytywane rejesty.

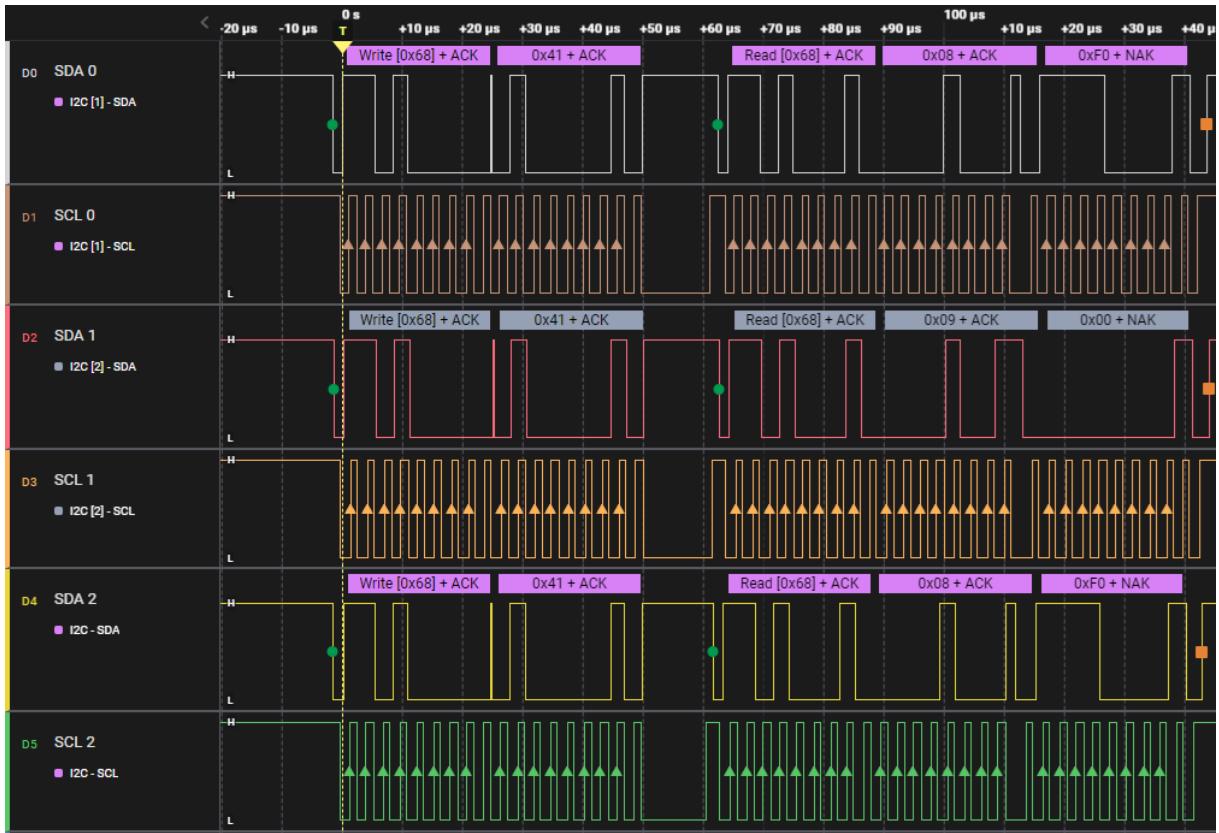
Nie odczytano danych z czujników za pomocą interfejsu SPI, ale nie jest to kluczowe dla działania systemu. Po przetestowaniu czujników za pomocą mikrokontrolera Arduino przy użyciu specjalnej biblioteki również nie działały, co wskazuje na uszkodzone czujniki.

10.3.3 Weryfikacja działania sygnału PWM

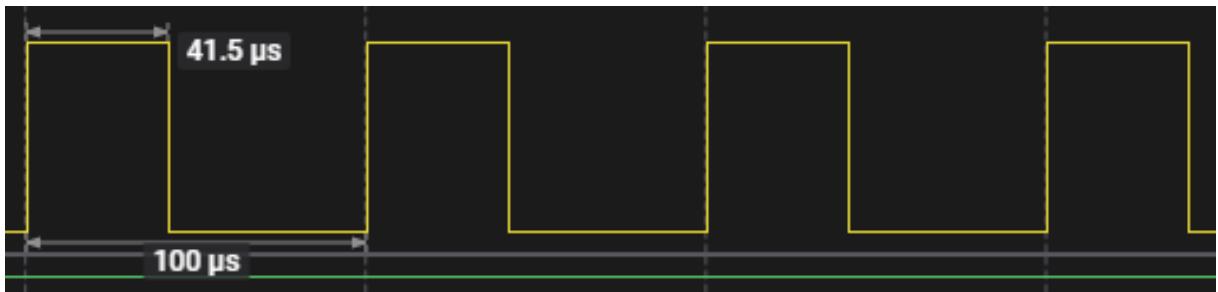
Na rysunku 70 przedstawiono przebieg sygnału PWM w trakcie działania testu. Współczynnik wypełnienia sygnału powinien być zbliżony do temperatury wyrażonej w stopniach Celcjusza. Odczytane wypełnienie wynosi 41,5%, co pokrywa się ze zmierzoną temperaturą. Odchylenie od zamierzonej wartości może wynikać z niskiej rozdzielczości analizatora logicznego zastosowanego w trakcie wykonywania pomiaru. Okres sygnału jest zgodny z przewidywanym.

10.3.4 Weryfikacja odczytu temperatury

Na rysunku 71 przedstawiono wygląd programu RealTerm w trakcie działania testu. Przedstawia on komunikaty wysypane przez system zabezpieczeń do użytkownika. Widać na nim, że przesyłana przez procesor temperatura jest zgodna z oczekiwaniami. Wielokrotne powtarzanie komunikatów dotyczących tej samej temperatury wynika z nieprzesyłania jej części ułamkowych przez DPU do CPU. W trakcie wykonywania tego testu czujnik podłączony do DPU nr. 1 zaczął być ogrzewany powodując zmianę jego temperatury widoczną na tym komunikacie. Zabezpieczenie zadziałało poprawnie, informując o zmianie najcieplejszego czujnika.

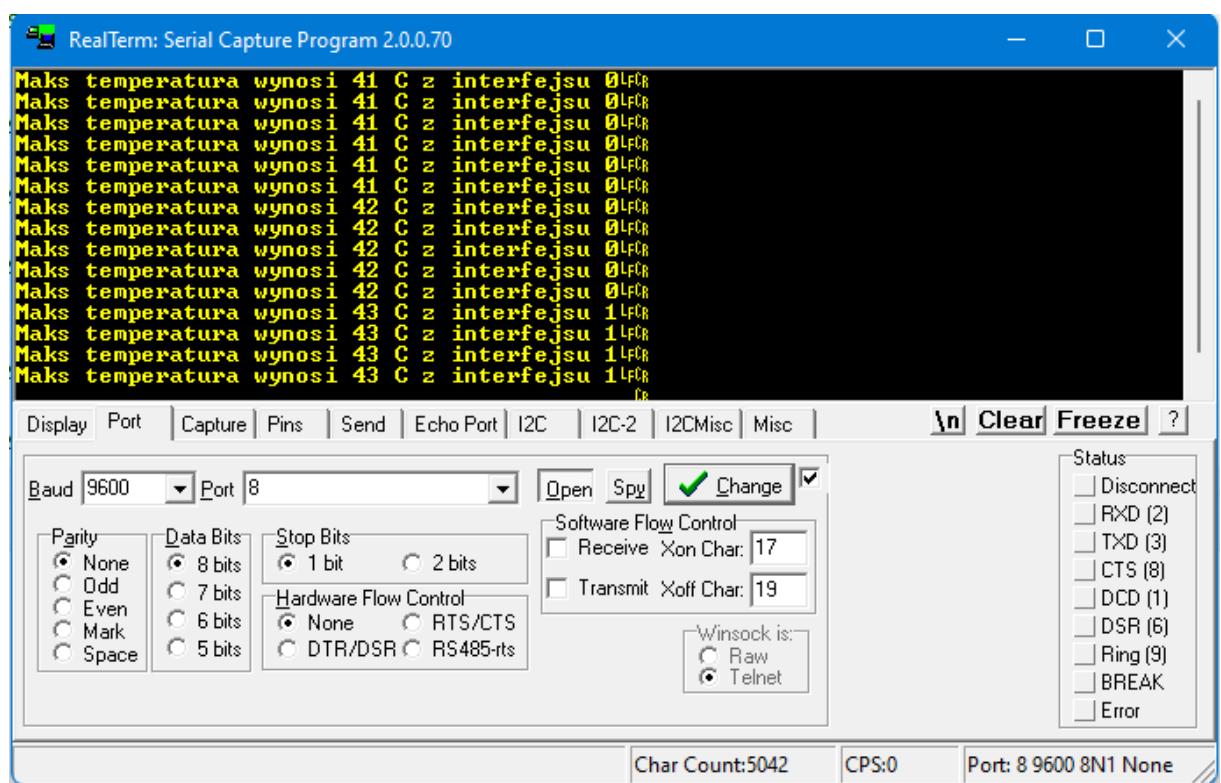


Rysunek 69. Przebieg sygnałów trzech protokołów komunikacji. Opracowanie własne.



Rysunek 70. Sygnał PWM w trakcie przeprowadzanego testu. Opracowanie własne.

Test został zakończony sukcesem. Czujniki działają zgodnie z ich zamierzeniem, a DPU poprawnie rozpoczyna komunikację. Nie przetestowano działania interfejsu SPI, ale nie jest to kluczowe dla działania systemu.



Rysunek 71. Wygląd programu RealTerm w trakcie działania testu. Opracowanie własne.

Rozdział 11: Podsumowanie

Wynikiem pracy jest opracowanie parametryzowalnego wieloprocesorowego systemu FPGA do implementacji modułów zabezpieczeń spełniającego wszystkie postawione przed nim cele. Może on równolegle obsługiwać szereg czujników wraz z algorytmami przetwarzania danych pomiarowych, pozwalając na minimalizację czasu reakcji na sygnał wykryty przez czujnik. Działanie systemu zostało udowodnione na rzeczywistym układzie. W ramach pracy zrealizowano szereg zadań:

- Opracowano i zaimplementowano procesor w formie bloku IPcore-HDL na układ FPGA z zestawem instrukcji ATTEL AVR. Jest on w stanie wykonać dowolny program używający tego zestawu instrukcji. Każda instrukcja jest wykonywana przez trzy kolejne etapy, implementując przez to potokowość. Są to:
 - Etap odczytywania instrukcji z pamięci programu, który zajmuje się obsługą wskaźnika instrukcji.
 - Etap dekodowania instrukcji którego zadaniem jest generacja szeregu sygnałów określający czynności wykonywane w kolejnym etapie.
 - Etap wykonywania instrukcji. Obsługuje on blok arytmetyczny ALU, rejesty wewnętrzne procesora oraz pamięć danych. ALU jest odpowiedzialne za wykonywanie działań matematycznych oraz logicznych. Rejestry obsługują różnorodne instrukcje adresowania, umożliwiając natychmiastową zmianę wartości wybranych rejestrów.
- Opracowano i zaimplementowano w warstwie HDL moduły komunikacji umożliwiające wysyłanie przez procesor danych do użytkownika i odwrotnie.
- Opracowano i zaimplementowano moduł wgrywania programu umożliwiający zaprogramowanie systemu zabezpieczeń. Do tego celu wykorzystywane są pliki bazujące na po-wszechnie wykorzystywanym formacie Intel Hex. Na ich podstawie, za pomocą napisanego skryptu, generowany jest plik zawierający zawartość poszczególnych pamięci programu. Każdy ze rdzeni systemu może wykonywać inny program.
- Opracowano i zaimplementowano moduł wyboru interfejsów umożliwiający połączenie dowolnego rdzenia DPU z dowolnym interfejsem, dzięki czemu jest możliwa zmiana obsługiwanych czujników.
- Opracowano i zaimplementowano system obsługi modułów komunikacji. Obsługuje on dwie pamięci FIFO i pozwala na automatyzację transmisji danych przez protokoły danych oraz obsługę sygnałów z procesora.
- Zaimplementowano interfejs UART umożliwiający komunikację z użyciem tego protokołu komunikacji. Posiada możliwość wybrania szeregu ustawiń.

- Opracowano interfejsy komunikacyjne I2C, SPI i PWM kompatybilne z systemem. Dodano do nich system obsługi ich funkcjonalności przez procesor.
- Zintegrowano wymienione wyżej moduły w jednolity parametryzowalny system. Wydzielono procesory DPU i CPU, wprowadzono system komunikacji między nimi oraz system obsługi przerwań. CPU jest odpowiedzialne za zarządzanie komunikacją zewnętrzną i nadzorowanie pracy DPU, które z kolei obsługują pojedynczy, wybrany przez CPU interfejs.
- Przeprowadzono symulacje systemu zabezpieczeń oraz jego poszczególnych modułów. Umożliwiły one sprawdzenie działania systemu i jego naprawę przed wgraniem go na układ FPGA.
- Napisano biblioteki programistyczne ułatwiające zaprogramowanie systemu. Definiują one rejesty systemu oraz implementują szereg funkcji. Dodatkowo zawierają obsługę architektury AVR bazując na bibliotekach dostarczonych przez firmę ATMEL.
- Przeprowadzono testy systemu na układzie FPGA. Został zaimplementowany przykładowy projekt systemu zabezpieczeń w programie Vivado dla płytki deweloperskiej "Zybo Z7-20"na układ FPGA „Zynq 7000”. Projekt korzysta z przykładowych czujników MPU6050 i MPU9250. Napisano program do ich obsługi symulujący możliwe rzeczywiste zastosowanie, a poprawność odczytywania danych przedstawiono za pomocą analizatora logicznego oraz terminala RealTerm.

System pozwala na szybkie, reakcje systemu obsługującego wiele czujników. Jest w stanie równolegle przetwarzać wiele danych, a czas reakcji jest niezależny od liczby obsługiwanych czujników. Taka własność jest szczególnie istotna w systemach zabezpieczeń pracujących z wieloma czujnikami, gdzie istnieje dodatkowa konieczność implementacji algorytmów analizy danych z czujników zabezpieczeń. Większość zastosowań wbudowanych nie ma stawianych wymagań prawdziwej równoległości. Przy wielu, docelowo nawet kilkudziesięciu często obsługiwanych czujnikach, system zabezpieczeń może okazać się znacznie wydajniejszy od mikrokontrolerów.

Projekt może być dalej rozwijany, w szczególności na zasadach „opensource”. Planowany rozwój zakłada:

- Dodanie modułu umożliwiającego debugowanie programu oraz obsługującego interfejs JTAG.
- Optymalizację wykorzystania zasobów przez moduł rejestrów oraz wgrywania programu.
- Dodanie obsługi dodatkowych protokołów komunikacji.
- Dodanie obsługi timera i systemu jego obsługi.
- Dodanie wsparcia dla zewnętrznej pamięci danych.
- Zwiększenie przejrzystości implementacji w języku opisu sprzętu oraz jej otwartoźródłowe upublicznenie.
- Zmniejszenie procesorów DPU poprzez implementację procesora ATTiny.

Opracowana i zrealizowana architektura systemu łączy plastyczność układów FPGA z prostotą programowania mikrokontrolera. Zgodnie z przeanalizowaną literaturą nie znaleziono takiego rozwiązania. Rozwiązanie znajdzie zastosowanie w miejscach, gdzie konieczna jest szybka analiza danych z wielu czujników pracujących z różnymi interfejsami, takimi jak SPI, 1-wire, I2C, UART, jak również komparatory, szybkie przetworniki ADC, interfejsy 4-20mA itp. System jest parametryzowalny w wielu zakresach. Opracowywanie kodu algorytmów jest natomiast uproszczone poprzez zastosowanie architektury wykorzystującej jednostki procesorowe wykorzystujące architekturę AVR i język C.

Bibliografia

- [1] A. Wojenski i in. „Multichannel measurement system for extended SXR plasma diagnostics based on novel radiation-hard electronics”. W: *Fusion Engineering and Design* 123 (2017), s. 727–731. DOI: 10.1016/j.fusengdes.2017.04.134.
- [2] B. Juszczyszyn i in. „Management and protection system for superconducting tokamak”. W: *Proceedings of SPIE - The International Society for Optical Engineering* 9662 (wrz. 2015), s. 9662–083.
- [3] Koch Alexander W. *Measurement and Sensor Systems: A Comprehensive Guide to Principles, Practical Issues and Applications*. 1st. Springer Nature, 2024.
- [4] Wiesław Winiecki. *Organizacja Komputerowych Systemów Pomiarowych*. Oficyna Wydawnicza Politechniki Warszawskiej, 2006. ISBN: 83-87012-82-3.
- [5] Jacob Fraden. *Handbook of Modern Sensors Physics, Designs, and Applications*. 4st. Springer, 2010. ISBN: 2010932807.
- [6] Thompson S. *Control systems: engineering and design*. Longman Scientific i Technical, 1989.
- [7] F RICHARD M. WHITE. „A Sensor Classification Scheme”. W: *IEEE TRANSACTIONS ON ULTRASONICS, FERROELECTRICS, AND FREQUENCY CONTROL* UFFC-34.2 (1987).
- [8] Webster John G. *Measurement, Instrumentation, and Sensors Handbook: Spatial, Mechanical, Thermal, and Radiation Measurement*. Crc Pr Inc, 2014. ISBN: 9781439848883.
- [9] Propeller 1. URL: <https://www.parallax.com/propeller-1/> (term. wiz. 15.06.2023).
- [10] Propeller 2. URL: <https://www.parallax.com/propeller-2/> (term. wiz. 15.06.2023).
- [11] Green Array Chips. URL: <https://www.greenarraychips.com/> (term. wiz. 15.06.2023).
- [12] „Architecture of a parallel-processing multi-microcontroller system and timing control method thereof”. US20070067605A1. 2005.
- [13] *Fast-forward — comparing a 1980s supercomputer to the modern smartphone*. URL: <https://blog.adobe.com/en/publish/2022/11/08/fast-forward-comparing-1980s-supercomputer-to-modern-smartphone> (term. wiz. 18.09.2024).
- [14] David A. Patterson i John L. Hennessy. „Computer architecture: a quantitative approach”. W: Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.
- [15] *C (programming language)*. URL: [https://en.wikipedia.org/wiki/C_\(programming_language\)](https://en.wikipedia.org/wiki/C_(programming_language)) (term. wiz. 11.10.2024).
- [16] Tood Austin Andrew S. Tanenbaum. *Structured computer organization, sixth edition*. Pearson, 2013. Rozdz. 2.1.3.

- [17] Carlton Neuenfeldt. *ARM Architecture: RISC-Based Computer Design*. Createspace Independent Publishing Platform, 2016.
- [18] David A. Patterson i John L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Morgan Kaufmann Publ Inc, 2020.
- [19] Daniel J. Pack Steven F. Barrett. *Microchip AVR® Microcontroller Primer*. Morgan Kaufmann Publ Inc, 2019.
- [20] Bernard Goossens. *Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis*. Springer, 2023.
- [21] Jim Ledin. *Modern Computer Architecture and Organization - Second Edition*. Packt, 2020.
- [22] Kenneth J. Ayala. *The 8051 Microcontroller Architecture, Programming, and Applications*. West Publishing Company, 1991.
- [23] Jari Nurmi. *Processor Design System-on-Chip Computing for ASICs and FPGAs*. Springer, 2007.
- [24] William Stallings. *Computer organization and architecture designing for performance, tenth edition*. Pearson, 2016. Rozd. 17.1.
- [25] *MIMD- Distributed memory*. URL: https://en.wikipedia.org/wiki/Multiple_instruction,_multiple_data#Distributed_memory (term. wiz. 18.10.2024).
- [26] *IBM 7030 Stretch*. URL: https://en.wikipedia.org/wiki/IBM_7030_Stretch (term. wiz. 19.10.2024).
- [27] Tood Austin Andrew S. Tanenbaum. *Structured computer organization, sixth edition*. Pearson, 2013. Rozd. 2.1.6.
- [28] Haowei Jiang i in. „A 22.3-nW, 4.55 cm² Temperature-Robust Wake-Up Receiver Achieving a Sensitivity of –69.5 dBm at 9 GHz”. W: *IEEE Journal of Solid-State Circuits* 55.6 (2020), s. 1530–1541. DOI: 10.1109/JSSC.2019.2948812.
- [29] *I2C-bus specification and user manual*. UM10204. Rev. 7.0. NXP. Paź. 2021.
- [30] *Introduction to SPI Interface*. Analog Devices. Wrz. 2018.
- [31] *UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter*. Analog Devices. Grud. 2020.
- [32] *Overview of 1-Wire Technology and Its Use*. Analog Devices. Lip. 2008.
- [33] *Pulse-width modulation*. URL: https://en.wikipedia.org/wiki/Pulse-width_modulation (term. wiz. 21.03.2024).
- [34] David A. Patterson i John L. Hennessy. „Computer architecture: a quantitative approach”. W: Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.
- [35] *U.S. Microprocessor Market Size, Share and Trend Analysis Report By Architecture Type (ARM MPU, x64, x86, MIPS), By Application (Smartphones, Personal Computers), And Segment Forecasts, 2024 - 2030*. URL: <https://www.grandviewresearch.com/industry-analysis/us-microprocessor-market-report> (term. wiz. 18.01.2024).
- [36] John L. Hennessy David A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann, 2012. Rozd. E.

- [37] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. 1st. USA: Newnes, 2004. ISBN: 0750676043.
- [38] Jari Nurmi. *Processor Design System-on-Chip Computing for ASICs and FPGAs*. Springer, 2007.
- [39] Mike Santarini. „Xilinx Ships Industry's First 20-nm All Programmable Devices”. W: *Xcell Journal Xilinx, Inc.* (2014).
- [40] *Field Programmable Gate Array (FPGA) Market Size, Share and Trends Analysis Report By Type (Low-End, High-End), By Technology (SRAM, Antifuse, Flash), By Application (Military and Aerospace, Telecom), By Region, And Segment Forecasts, 2023 - 2030*. URL: <https://www.grandviewresearch.com/industry-analysis/fpga-market> (term. wiz. 18.12.2024).
- [41] *OpenCores*. URL: <https://opencores.org/> (term. wiz. 18.12.2024).
- [42] *Design-your-arm-cortex-m0-iot-chip-for-free-on-demand*. URL: <https://community.arm.com/developer/ip-products/processors/designstart/b/blog/posts/design-your-arm-cortex-m0-iot-chip-for-free-on-demand> (term. wiz. 22.12.2024).
- [43] *GENERAL: Intel HEX File Format*. KA003292. ARM.
- [44] *ASCII*. URL: <https://en.wikipedia.org/wiki/ASCII> (term. wiz. 11.04.2024).
- [45] *Hex to ASCII Text String Converter*. URL: <https://www.rapidtables.com/convert/number/hex-to-ascii.html> (term. wiz. 11.04.2024).
- [46] *AVR Core*. URL: https://opencores.org/projects/avr_core (term. wiz. 15.04.2023).
- [47] *CPU Lecture*. URL: https://opencores.org/projects/cpu_lecture (term. wiz. 15.04.2023).
- [48] *8-bit Atmel with 8KBytes In-System Programmable Flash*. Rev.2486AA–AVR. Atmel. Lut. 2013.
- [49] David A. Patterson i John L. Hennessy. „Computer architecture: a quantitative approach”. W: Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. Rozd. 1.7.
- [50] *AVR Instruction Set Manual*. DS40002198B. Atmel. 2021, s. 9–16.
- [51] *Half-carry flag*. URL: https://en.wikipedia.org/wiki/Half-carry%5C_flag%5D (term. wiz. 12.09.2024).
- [52] *Binary-coded decimal*. URL: https://en.wikipedia.org/wiki/Binary-coded_decimal (term. wiz. 12.09.2024).
- [53] *8-bit Atmel with 8KBytes In-System Programmable Flash*. Rev.2486AA–AVR. Atmel. Lut. 2013, s. 24.
- [54] *Block Memory Generator*. Xilinx. Sierp. 2021.
- [55] Scott Larson. *SPI Master (VHDL)*. URL: <https://forum.digikey.com/t/spi-master-vhdl/12717> (term. wiz. 21.03.2024).
- [56] *SPI Slave VHDL design*. URL: <https://surf-vhdl.com/spi-slave-vhdl-design/> (term. wiz. 18.04.2024).
- [57] *Serial Peripheral Interface: Clock polarity and phase*. URL: https://commons.wikimedia.org/wiki/File:SPI%5C_timing%5C_diagram%5C_CS.svg (term. wiz. 21.03.2024).

- [58] Scott Larson. *I²C Master (VHDL)*. URL: <https://forum.digikey.com/t/i2c-master-vhdl/12797> (term. wiz. 21.03.2024).
- [59] *A simple I²C minion in VHD*. URL: https://github.com/oetr/FPGA-I2C-Minion/blob/master/I2C%5C_minion.vhd (term. wiz. 25.02.2024).
- [60] *Zybo Z7 Board Reference Manual*. Wer. Revision B. Digilent. Lut. 2018.
- [61] *AVR SIMULATOR IDE*. URL: <https://www.oshonsoft.com/avr.php> (term. wiz. 27.09.2024).
- [62] *MPU-6000/MPU-6050 Register Map and Descriptions*. RM-MPU-6000A-00. version 4.2. InvenSense. Sierp. 2013, s. 30.
- [63] *MPU-9250 Register Map and Description*. RM-MPU-6000A-00. Revision 1.6. InvenSense. Sty. 2015, s. 33.
- [64] *UART, Serial Port, RS-232 Interface*. URL: <https://nandland.com/uart-serial-port-module/> (term. wiz. 10.01.2024).

Spis rysunków

1	Pozycja sensorów w systemie zabezpieczeń[5]	11
2	Wzrost wydajności procesorów[14]	17
3	Podział procesora na poziomy abstrakcji. Opracowanie własne.	18
4	Architektura von Neumanna. Opracowanie własne.	19
5	Architektura Harwadzka. Opracowanie własne.	19
6	Etapy wykonywania instrukcji. Opracowanie własne.	23
7	Widok przedstawiający składowe prostego bloku LUT.[37]	27
8	Schemat ilustrujący działanie układu FPGA[37]	27
9	Budowa elementu bloku DSP[37]	28
10	Schemat przedstawiający bloki RAM i DSP (tutaj nazwane Multipliers) w układzie FPGA [37]	28
11	Matryca zegarowa generująca sygnały zegarowe[37]	29
12	Schemat przedstawiający ogólną zasadę działania urządzenia. Opracowanie własne.	31
13	Schemat działania RSZ. Opracowanie własne.	32
14	Wybieranie interfejsu przez CPU. Opracowanie własne.	33
15	Przebieg wybranych sygnałów symulacji modułu wyboru interfejsów. Opracowanie własne.	34
16	Schemat przedstawiający moduł komunikacji z użytkownikiem. Opracowanie własne.	35
17	Ramka komunikacji pliku hex wraz z przykładową linijką. Opracowanie własne.	37
18	Przebieg wybranych sygnałów symulacji zapisu pamięci programu. Opracowanie własne.	38
19	Ramka danych wysyłana przez mikrokontroler do użytkownika. Opracowanie własne.	38
20	Etapy wykonywania instrukcji. Opracowanie własne.	41
21	Implementacja potokowości procesora. Opracowanie własne.	42
22	Odczytywanie czterobitowej instrukcji. Opracowanie własne.	43
23	Opracowana architektura modułu wykonywania instrukcji. Opracowanie własne.	53
24	Wykonywanie instrukcji na przykładzie instrukcji SUB. Opracowanie własne.	54
25	Schemat architektury modułu ALU. Opracowanie własne.	55
26	Rejestry ogólne. Opracowanie własne.	55
27	Implementacja adresowania rejestrów. Opracowanie własne.	57
28	Bity rejestru ustawień. Opracowanie własne.	58
29	Mapa adresów pamięci. Opracowanie własne.	59
30	Przebieg wybranych sygnałów symulacji odczytywania danych z pamięci programu. Opracowanie własne.	61

31	Przebieg wybranych sygnałów symulacji dekodowania instrukcji. Opracowanie własne.	63
32	Przebieg wybranych sygnałów symulacji rejestrów. Opracowanie własne.	64
33	Przebieg wybranych sygnałów symulacji wykonywania instrukcji. Opracowanie własne.	66
34	Część wygenerowanych przez kompilator instrukcji procesora	68
35	Przebieg wybranych sygnałów modułu pobierania instrukcji. Opracowanie własne.	69
36	Przebieg wybranych sygnałów modułu wykonywania. Opracowanie własne.	69
37	Przebieg wybranych sygnałów symulacji procesora. Opracowanie własne.	70
38	Fragment symulacji. Opracowanie własne.	71
39	Fragment symulacji. Opracowanie własne.	72
40	Schemat ogólny działania interfejsów. Opracowanie własne.	75
41	Przebieg sygnału protokołu UART. Opracowanie własne.	76
42	Budowa interfejsu UART. Opracowanie własne.	76
43	Symulowany przebieg sygnałów modułu wysyłania UART. Opracowanie własne.	78
44	Symulowany przebieg sygnałów interfejsu UART. Opracowanie własne.	79
45	Symulowany przebieg wybranych sygnałów interfejsu UART. Opracowanie własne.	79
46	Przebieg sygnału protokołu SPI. Opracowanie własne.	80
47	Schemat interfejsu SPI. Opracowanie własne.	81
48	Przebieg wybranych sygnałów symulacji interfejsu SPI. Opracowanie własne.	82
49	Przebieg sygnałów protokołu I^2C . Opracowanie własne.	82
50	Schemat interfejsu I^2C . Opracowanie własne.	83
51	Schemat obrazujący wysyłanie rozszerzonego adresu. Opracowanie własne.	84
52	Przebieg wybranych symulowanych sygnałów interfejsu I^2C . Opracowanie własne.	84
53	Przebieg wybranych symulowanych sygnałów interfejsu I^2C . Opracowanie własne.	85
54	Przebieg sygnałów PWM dla różnych współczynników wypełnienia. Opracowanie własne.	85
55	Schemat interfejsu PWM. Opracowanie własne.	86
56	Symulowane sygnały PWM dla różnych wartości. Opracowanie własne.	86
57	Schemat przedstawiający podział na poszczególne etapy wymagane do testowania projektu. Opracowanie własne.	87
58	Wygląd programu RealTerm	91
59	Wygląd płytki ewaluacyjnej Zybo Z7-20[60]	99
60	Schemat przedstawiający architekturę układu Zynq-7000[60]	100
61	Program AVR simulator IDE w trakcie symulacji. Opracowanie własne.	100
62	Widok sygnałów modułu ILA w programie Vivado. Opracowanie własne.	101
63	Zużycie wybranych zasobów układu FPGA przez mikrokontroler w zależności od liczby DPU. Liczba interfejsów jest stała. Opracowanie własne.	102
64	Zależność maksymalnej częstotliwości zegara od liczby DPU dla stałej liczby interfejsów. Opracowanie własne.	103
65	Schemat przeprowadzonego testu. Opracowanie własne.	105
66	Płyтka ewaluacyjna podłączona do czujników. Opracowanie własne.	110

67	Przebieg sygnałów RX i TX podczas rozpoczęcia wgrywania programu. Opracowanie własne.	110
68	Widok programu RealTerm po bezbłędnym wgraniu programu na układ FPGA. Opracowanie własne.	111
69	Przebieg sygnałów trzech protokołów komunikacji. Opracowanie własne.	112
70	Sygnal PWM w trakcie przeprowadzanego testu. Opracowanie własne.	112
71	Wygląd programu RealTerm w trakcie działania testu. Opracowanie własne.	113

Spis tebel

1	Własności czynników monitorowanych przez systemy zabezpieczeń	13
2	Porównanie architektur procesorów	21
3	Wymagania procesorów w najważniejszych zastosowaniach[34]	25
4	Zużycie wybranych zasobów układu FPGA przez poszczególne, wybrane bloki mikrokontrolera. Mux to skrót oznaczający multiplekser, a występująca po nim liczba to liczba jego wejść. Opracowanie własne.	102