Wydział Automatyki, Elektroniki i Informatyki

mgr inż. Ernest Antolak

Opracowanie efektywnej metodologii dopasowywania struktury sprzętowej oraz optymalizacja zasobów systemu PRET do wymagań czasowych zadań

Rozprawa doktorska napisana pod kierunkiem

dr. hab. inż. Andrzeja Pułki, prof. Pol. Śl.

Gliwice 2022

Pragnę serdecznie podziękować
dr hab. inż. Andrzejowi Pułce, prof. Pol. Śl.
za wprowadzenie i ukierunkowanie
w świecie pracy naukowej,
cierpliwość oraz życzliwą pomoc.

Wielkie podziękowania składam mojej rodzinie,
a w szczególności mojej żonie Joannie
za nieustanną wiarę we mnie, wszelkie wsparcie
oraz pomoc przy korekcie tekstu pracy.

Niniejsza praca doktorska powstała w ramach projektu:

„CyPhiS – Projekt nowoczesnych studiów doktoranckich w dziedzinie systemów cyber–fizycznych”

Projekt ten był dofinansowany z Funduszy Europejskich
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
przez Wydział Automatyki, Elektroniki i Informatyki Politechniki Śląskiej
w ramach Działania 3.2 Studia doktoranckie
na podstawie umowy nr POWR.03.02.00-00-I007/17-00

Spis treści

Spis oznaczeń i symboli 1

1 Wstęp 5

1.1 Uzasadnienie tematu podjęcia badań 7

1.2 Tezy rozprawy 7

2 Przegląd wybranych rozwiązań 9

2.1 Przeplot wątków 9

2.2 Pamięci lokalne typu Scratchpad 10

2.3 Koło pamięci 12

2.4 Kontroler przeplotu 14

2.5 Rozszerzenie zestawu instrukcji procesora 15

3 Założenia wstępne 17

3.1 Metoda implementacji systemu 17

3.2 Platforma sprzętowa 18

3.3 Oprogramowanie 19

4 Opracowanie koncepcji systemu 21

4.1 Charakterystyka systemu 22

4.2 Model zadania 23

4.3 Wykorzystanie przeplotu wątków 28

4.4 Dopasowywanie struktury sprzętowej do zadań 29

4.5 Pamięć systemu oraz komunikacja wątków 30

4.6 Przewidywalność czasowa systemu 33

4.7 Energooszczędność systemu 34

4.8 Model harmonogramowania zadań 35

5 Budowa oraz implementacja systemu 37

5.1 Architektura systemu 37
5.1.1 Pamięć oraz rejestry systemu 38
5.1.2 Potok przetwarzania danych 39
5.1.2.1 Ograniczenia przeplotu wątków 43
5.1.3 Sterowanie oraz obsługa przeplotu wątków 45
5.2 Konfigurowalność systemu 48
5.2.1 Konfigurowalność struktury sprzętowej 48
5.2.2 Konfigurowalność potoku 52
5.3 Magistrala wymiany danych między wątkami 54
5.3.1 Adresacja wątków 54
5.3.2 Proces wysyłania zapytania wymiany danych 55
5.3.3 Autorska implementacja magistrali dwukierunkowej 56
5.3.3.1 Architektura magistrali dwukierunkowej 57
5.3.3.2 Przewidywalność czasowa wymiany danych 64
5.3.4 Autorska implementacja magistrali jednokierunkowej 65
5.3.4.1 Architektura magistrali jednokierunkowej 68
5.3.4.2 Kontroler magistrali rdzenia 75
5.3.4.3 Typy obsługiwanych wątków 78
5.3.4.4 Przewidywalność czasowa wymiany danych 81
5.3.5 Porównanie wersji magistral 82
5.4 Kompilator zadań 84
6 Metodologia harmonogramowania zadań 87
6.1 Pozyskiwanie parametrów zadań 87
6.2 Przykładowe zadania 88
6.3 Podział zadań 90
6.3.1 Algorytm BLTS 91

6.3.1.1 Metodologia MINRES 95
6.3.1.2 Metodologia MAXPRO 97
6.3.1.3 Metodologia SFERA 99
6.4 Szeregowanie zadań 101
6.4.1 Algorytm szeregowania zadań (wątków sprzętowych) 101
6.4.1.1 Wątki Precyzyjne - SHT 103
6.4.1.2 Inicjalizacja szeregowania wątków 104
6.4.1.3 Szeregowanie właściwe 109
6.4.1.4 Szeregowanie wątków precyzyjnych - SHT 113
6.4.1.5 Eliminacja jałowych cykli potoku 116
6.4.1.6 Usuwanie błędów powstałych po zapętleniu zszeregowanego cyklu 119
6.4.2 Program szeregujący zadania 124
6.5 Symulacja wykonywania zadań 126
6.5.1 Opis Symulatora 126
6.5.1.1 Czas trwania symulacji 126
6.5.1.2 Minimalny i maksymalny czas wykonania zadań 130
6.5.1.3 Opis implementacji metody wyszukiwania maksymalnej liczby cykli zegara, potrzebnej do wykonania zadania. 134
6.5.2 Program umożliwiający symulacje 136
6.5.3 Optymalizacja procesu szeregowania 139
7 Weryfikacja działania systemu 141
7.1 Weryfikacja konfigurowalności systemu 141
7.1.1 Scenariusz testowy 141
7.1.2 Badanie zapotrzebowania systemu na zasoby 143
7.1.3 Badania zapotrzebowania systemu na energie 148

7.1.4 Porównanie systemu wielozadaniowego z klasycznym systemem jednozadaniowym....154
7.2 Weryfikacja przewidywalności czasowej systemu....158
7.2.1 Opis środowiska pomiarowego....159
7.2.2 Przewidywalność czasowa mikroarchitektury systemu....160
7.2.2.1 Zadania niewspółpracujące....161
7.2.2.2 Zadania współpracujące CTHM....165
7.2.2.3 Zadania współpracujące CT....168
7.2.2.4 Zadania mieszane....172
7.3 Przykładowa realizacja systemu....175
7.3.1 Proces harmonogramowania zadań....175
7.3.1.1 Szeregowanie i symulacja - scenariusz MINRES....176
7.3.1.2 Szeregowanie i symulacja - scenariusz MAXPRO....176
7.3.1.3 Szeregowanie i symulacja – scenariusz SFERA....177
7.3.2 Badania zapotrzebowania na moc i zasoby systemu....177
7.3.3 Weryfikacja przewidywalności czasowej metodologii harmonogramowania....179
8 Podsumowanie....186
8.1 Porównanie z innymi rozwiązaniami....186
8.2 Wnioski....188
8.3 Zalecenie dla projektantów systemów PRET....192
9 Spis literatury....193
Dodatki....201
A. Opublikowane wzory i algorytmy....201
B. Dane z procesu weryfikacji przewidywalności czasowej metodologii harmonogramowania....206
C. Mechanizm kontroli czasu wykonania zadania....213

Spis oznaczeń i symboli

ALU – faza wykonywania operacji arytmetyczno logicznych

ASIC – (ang. *Application-Specific Integrated Circuit*) – Specjalizowany układ scalony

BLIS – (ang. *Balanced Load Independent Task Scheduling*) algorytm podziału zadań

BLTS – (ang. *Balanced Load Task Scheduling*) algorytm podziału zadań

CLK – zegar taktujący

CMOS – (ang. *Complementary Metal-Oxide-Semiconductor*) technologia wytwarzania układów scalonych

COTAS – (ang. *Co-Operating Task Scheduling*) algorytm podziału zadań

CP – cykl przeplotu

CT – (ang. *Cooperative Thread*) wątek współpracujący

CTMH – (ang. *Cooperative Thread Memory Hard*) wątek współpracujący o stałym czasie wymiany danych z innym wątkiem

DEMUX – demultiplekser

DF – (ang. *Data Fetch*) faza pobrania danych z pamięci

EDF – (ang. *Earliest Deadline First*) – algorytm harmonogramowania zadań

EXE – (ang. *Execute*) faza wykonania instrukcji

FPGA – (ang. *Field-Programmable Gate Array*) programowalny układ logiczny

GALS – (ang. *Globally Asynchronous Locally Synchronous*) – układy globalnie asynchroniczne, lokalnie synchronicznie

GPR – (ang. *General Purpose Register*) rejestr ogólnego przeznaczenia

HT – (ang. *Hard Thread*) wątek standardowy

IC – (ang. *Idle Cycle*) faza oczekiwania

ID – (ang. *Instruction Decode*) faza zdekodowania instrukcji

IF – (ang. *Instruction Fetch*) faza pobrania instrukcji z pamięci

ISA – (ang. *Instruction Set Architecture*) architektura instrukcji procesora

LC – (ang. *Load Cycle*) faza wysyłania odpowiedzi

LFSR – (ang. *Linear-Feedback Shift Register*) rejestr przesuwający z liniowym sprzężeniem zwrotnym

MAXPRO – (ang. *Maximum Processors*) – metodologia podziału zadań dążąca do minimalizacji zapotrzebowania na moc

MEM – (ang. *Memory Access*) faza dostępu do pamięci

MI – instrukcja odwołania do pamięci innego wątku

MINRES – (ang. *Minimun Resources*) – metodologia podziału zadań dążąca do minimalizacji zasobów

MMCM – (ang. *Mixed-Mode Clock Manager*) układ podziału/zwielokrotnienia częstotliwości, odpowiednik PLL)

MTP – (ang. *Multi Task Processor*) procesor wielozadaniowy

MUX – multiplekser

NCT – (ang. *Noncooperative Thread*) wątek niewspółpracujący

ONEHOT – typ kodowania z jednym bitem aktywnym

PC – licznik rozkazów

PLL – (ang. *Phase Locked Loop*) pętla synchronizacji fazy

PRET – (ang. *Precision-Timed Machines*) systemy przewidywalne czasowo

RTOS – (ang. *Real-Time Operating System*) system operacyjny czasu rzeczywistego

SA – (ang. *Select Address*) faza wyboru adresu oraz banku pamięci

SC – (ang. *Store Cycle*) faza wysyłania zapytań magistrali

SD – (ang. *Select Data*) faza wyboru banku pamięci

SFERA – (ang. *Scheduling for Energy Reduced Applications*) – metodologia podziału redukcji zapotrzebowania na moc

SFT – (ang. *Shift*) faza przesuwania operandów

SHT – (ang. *Strong Hard Thread*) wątek precyzyjny

SI – instrukcja standardowa

SoC – (ang. *System on Chip*) – układ scalony zawierający kompletny układ system elektroniczny

SP – (ang. *Scratchpad*) – pamięci typu Scratchpad

SR – (ang. *Select Registers*) faza wyboru banku rejestrów

SRF – (ang. *Shortest Remaining Time First*) – algorytm harmonogramowania zadań

STP – (ang. *Single Taks Processor*) procesor przetwarzający tylko jedno zadanie

TDMA – (ang. *Time Division Multiple Access*) – technika dostępu kliku urządzeń do jednego medium, wykorzystująca podział dostępu ze względu na czas

TH – (ang. *Thread*) wątek

WB – (ang. *Write Back*) faza zapisania wyników instrukcji

1 Wstęp

Współczesne procesory oraz systemy wieloprocesorowe charakteryzują się bardzo dużym stopniem złożoności sprzętowej i są zdolne do przetwarzania bardzo skomplikowanych algorytmów programowych. Dodatkowo, błyskawiczny rozwój technologii półprzewodnikowych sprawia, że systemy te są taktowane coraz szybszymi sygnałami zegarowymi. Paradoksalnie, ten postęp technologiczny sprawia, że pojawiają się problemy związane z przewidywalnością czasową takich systemów [1]. Początkowo, zagadnienia przewidywalności czasowej procesorów zaczęto rozwiązywać za pomocą oprogramowania, tworząc systemy operacyjne czasu rzeczywistego, czyli tzw. RTOS [2], [3]. Niestety takie podejście do problemu obarczone jest dużą dozą niedokładności, ponieważ obsługa zadań odbywa się na bardzo wysokim poziomie abstrakcji [4], [5]. Większość języków programowania takich jak C [6], [7], czy C++ nie daje bezpośredniej kontroli nad czasem upływu zadań [8], co w połączeniu z dużą liczbą poziomów abstrakcji umożliwia jedynie bardzo przybliżoną kontrolę czasu [9]–[11].

Koncepcja układów przewidywanych czasowo, tzw. PRET (ang. PREcision Time machines) została zaprezentowana już 2007 roku przez Edwards'a i Lee [12]. Zakładała ona, że system taki powinien zawsze, niezależnie od obciążenia wykonać zaplanowane zadania na czas. Koncepcję tę rozwijało wielu naukowców: Thiele i Wilhelm [13] sformułowali zestaw rekomendacji i wytycznych ułatwiających projektowanie krytycznych czasowo systemów wbudowanych. Ip i Edwards [14] zaproponowali rozszerzenie listy rozkazów procesorów RISC o instrukcję DEADLINE umożliwiającą kontrole czasu wykonywania zadań. Działająca na uniwersytecie w Berkeley grupa CHESS (Center for Hybrid and Embedded Software Systems) [15] pod kierunkiem Prof. Edwarda Lee rozwinęła ideę procesora PRET, zaproponowała ona miedzy innymi sposób przewidywalnego dostępu do pamięci proponując metodę tzw. „koła pamięci (z ang. „*memory wheel*". Grupa naukowców z Danii, Austrii, Francji oraz USA zaprezentowała architekturę procesora Patmos [16]. Następnie w 2015 roku grupa 24 europejskich naukowców zaprezentowała projekt T-CREST [17], w którym pokazano wielordzeniowe podejście do pierwotnej koncepcji systemu PRET. W [18]–[23] opublikowano problemy wynikające z harmonogramowania zadań pracujących równolegle w systemach wielordzeniowych wraz z próbami ich rozwiązania.

Zapotrzebowanie na takie systemy wynika z urządzeń od których bezpośrednio zależy ludzkie życie, takich jak aparatura medyczna, lotnictwo czy przemysł samochodowy [24] w którym coraz częściej stosuje się systemy bezpieczeństwa obsługiwane przez procesory. W dobie Internetu i koncepcji Internetu rzeczy (ang. Internet of things, IoT) istnieje duże zapotrzebowanie na cyber-fizyczne urządzenia końcowe, od których często zależy nasze bezpieczeństwo [25], [26]. W przypadkach, w których czas wykonania takich zadań jest krytyczny, urządzenie takie musi bezwzględnie posiadać sprzętowe wspomaganie przewidywalności czasowej [27].

Jednym z zasadniczych efektów prezentowanej rozprawy doktorskiej jest opracowanie oryginalnego modelu rekonfigurowalnej architektury wielozadaniowego systemu czasu rzeczywistego oraz praktyczna implementacja tego systemu w układzie FPGA. Dodatkowo, system ten został wzbogacony o kilka nowatorskich rozwiązań sprzętowych.

Współczesne procesory RISC są procesorami potokowymi. Potokowość wprowadza jednak wiele nieprzewidywalności czasowej związanej z uruchamianiem potoku czy hazardami danych i sterowania. Jednym ze sposobów eliminacji tych nieprzewidywalności jest zastosowanie przeplotu zadań (wątków). Tę metodę po raz pierwszy zaproponowali Lee i Messerschmitt [28]. Koncepcję przeplotu wątków rozwinięto również w Instytucie Elektroniki w Politechniki Śląskiej [29]. W przypadku systemów wielordzeniowych, elementem, z którego wynika najwięcej nieprzewidywalności czasowej jest wymiana danych między rdzeniami. W sugerowanym rozwiązaniu została zaproponowana metoda redukcji powyższych nieprzewidywalności, która opiera się na rozszerzonej technice przelotu wątków.

Obok architektury system czasu rzeczywistego bardzo ważną jego częścią jest sposób harmonogramowania zadań oraz kontrola upływającego czasu na wykonanie zadania (ang. *deadline*). Klasyczne podejście prowadzi do używania algorytmów takich jak EDF [30]–[32] (ang. *Earliest deadline first*) czy SRF (ang. *Shortest remaining time first*) lub ich wariacji w znanej nam architekturze [33]. W przypadku użycia algorytmu EDF priorytet w wykonywaniu mają zadania o najniższym pozostałym czasie *deadline*. Algorytm SRF przyznaje pierwszeństwo w wykonywaniu zadań według pozostałego czasu wykonania zadania (najkrótszy czas posiada najwyższy priorytet). W sugerowanym rozwiązaniu zostanie przeanalizowany uniwersalny modelu systemu, którego

architektura może być dopasowana do zadań, które system ma wykonać. Harmonogramowanie zadań polega więc na zmianie struktury sprzętu w taki sposób, aby mógł on je wykonać przy zadanych ograniczeniach czasowych.

1.1 Uzasadnienie tematu podjęcia badań

W ramach rozprawy doktorskiej i kontynuacji wyżej przedstawionych zagadnień została opracowana metodologia dopasowywania ilości oraz rodzaju zasobów (sprzętu) do zadań i wymagań czasowych, w jakich one mają być wykonane. Metoda ta pokazuje nowatorskie podejście do problemu harmonogramowania zadań w systemach czasu rzeczywistego, ponieważ przy założeniu poprawnego oprogramowania, zapewnia, że każde zaplanowane zadanie zostanie wykonane na czas. Metodologia taka będzie bardzo pomocna w projektowaniu niewielkich systemów wbudowanych od których zależy ludzkie bezpieczeństwo i zdrowie. Miejscem docelowym funkcjonowania takich systemów jest aparatura medyczna, układy stosowane w systemach komunikacyjnych (pojazdy autonomiczne, lotnictwo czy kolej), systemy wyłączania awaryjnego maszyn w przemyśle czy w ogólnodostępnych urządzeniach końcowych IoT, nadzorujących np. prace domowego kotła. Projektowanie systemów PRET może być bardzo skomplikowane, dlatego w większości wypadków stosuje się proste do implementacji systemy RTOS, które obarczone są dużą dozą niedokładności co jest niezgodne z ideą bezpieczeństwa. Dlatego proponowana rozprawa doktorska ma ułatwić proces projektowania architektury i harmonogramowania zadań w systemie PRET.

Przeprowadzenie tego zadania wymagało utworzenia syntezowalnego niskopoziomowego modelu wielordzeniowego parametryzowanego systemu PRET. Model ten został zaimplementowany w układzie FPGA, gdzie poprawność jego działania zweryfikowano praktycznie.

1.2 Tezy rozprawy

Biorąc pod uwagę powyższe problemy i potrzeby sfomułowano tezę zasadniczą oraz tezy pomocnicze pracy:

Teza główna brzmi:

Metodologia dopasowywania struktury oraz zasobów systemu PRET do wymagań czasowych wykonywanych zadań, pozwala na efektywną implementację sprzętową krytycznie przewidywalnego systemu czasu rzeczywistego

Tezy pomocnicze brzmią:

1. Język opisu sprzętu Verilog pozwala na opisanie parametrycznego modelu systemu PRET, w którym konfigurowalna jest struktura systemu wynikająca z parametrów zadań
2. Dopasowana struktura sprzętowa systemu PRET pozwala wyznaczyć minimalny i maksymalny czas wykonania zadania, który jest niższy od zakładanego czasu deadline zadania
3. Metoda przeplotu wątków pozwala wyeliminować nieprzewidywalności czasowe wynikające z dostępu do danych innych zadań
4. Metodologia dopasowywania struktury oraz zasobów systemu PRET, pozwala na minimalizację zużytych zasobów sprzętowych i/lub minimalizację energii potrzebnej do funkcjonowania systemu

2 Przegląd wybranych rozwiązań

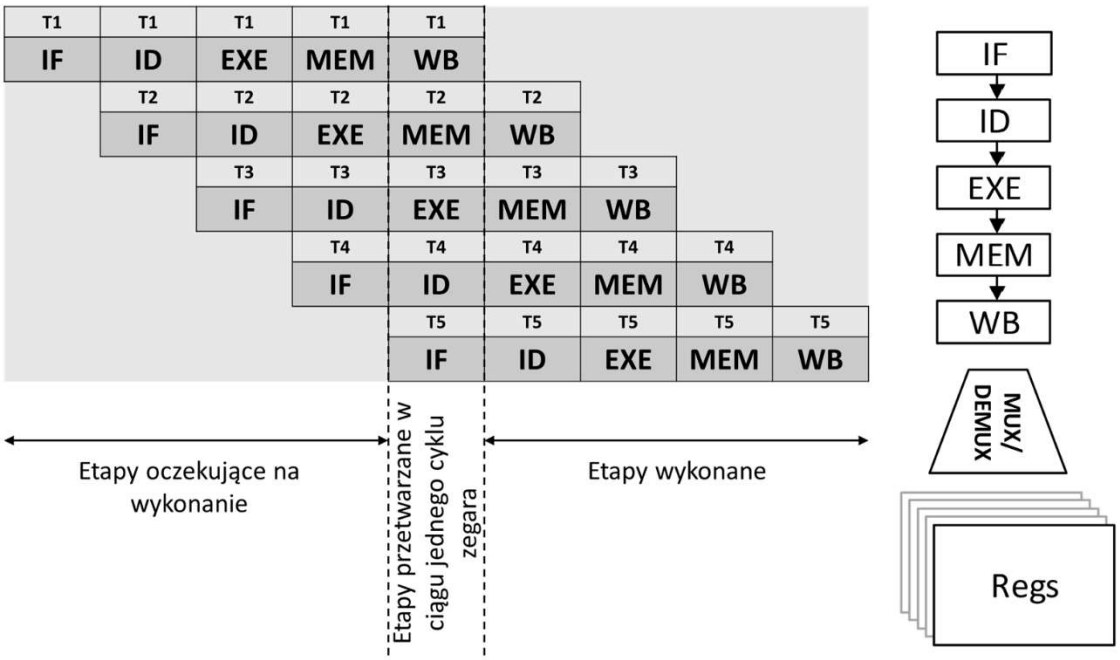
W rozdziale tym zostaną przedstawione najistotniejsze wg. autora tej pracy rozwiązania problemów, które są źródłami nieprzewidywalności czasowej wykonywania zadań. Rozwiązania te miały wpływ na mikroarchitekturę systemu zaproponowaną przez autora.

2.1 Przeplot wątków

Głównymi źródłami nieprzewidywalności czasowej architektur potokowych są hazardy danych i hazardy sterowania [12], [34]. Istnieją wprawdzie metody eliminacji błędów spowodowanych tymi hazardami takie jak np. układy przesuwania danych do przodu (ang. *data forwarding*) [35] lub układy przewidywania rozgałęzienia programu, niestety w większości nie są one przewidywalne czasowo. W zależności od rodzaju, kolejności i wyniku przeprowadzanych operacji, czas wykonywanych rozkazów może się od siebie różnić.

Metoda przeplotu wątków może wyeliminować wyżej wymienione problemy [8], [28], [29], [36]–[38]. W zaproponowanej architekturze pracuje znana, większa niż określona, minimalna liczba zadań, które są kolejno przeplatane w potoku procesora. Na Rys. 1 przedstawiono model klasycznej pięcioetapowej architektury wykorzystującej przeplot pięciu zadań.

Każde z zadań musi mieć swoją niezależną kopię rejestrów ogólnego przeznaczenia. Przy takim założeniu każde z nich może pracować niezależnie. Dzięki temu, że w jednym cyklu zegara przetwarzane są etapy instrukcji różnych zadań, w takiej architekturze nie występują problemy związane z hazardami. Należy jednak pamiętać, że przeplot zadań nie gwarantuje przewidywalności czasowej systemu. Metoda ta zostanie wykorzystana i rozwinięta w niniejszej rozprawie.



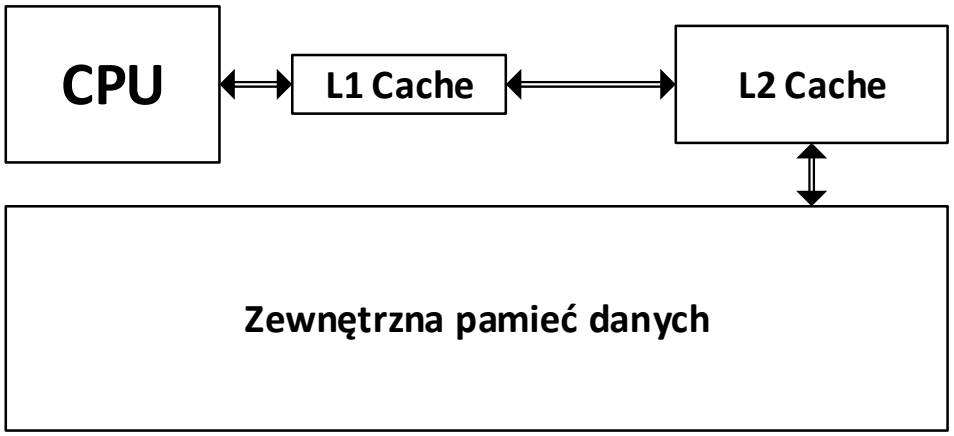
Rys. 1. Model przeplotu wątków

2.2 Pamięci lokalne typu Scratchpad

We współczesnych systemach elektronicznych powszechną praktyką jest dążenie do maksymalizacji mocy obliczeniowej systemu. Obecnie stosowane pamięci posiadają duże pojemności co negatywnie wpływa na czas dostępu do danych w nich zapisanych. Aby możliwe było wystarczająco szybkie dostarczanie przetwarzanych danych do potoku przetwarzania systemu, zaczęto stosować hierarchiczność takiej pamięci. Najszybsza, najmniej pojemna pamięć, znajduję się jak najbliżej procesora. Następnie występuje kolejny poziom pamięci, o większej pojemności, ale wolniejszej. Ostatnim poziomem takiej hierarchii jest pamięć zewnętrzna, najbardziej pojemna, ale również najwolniejsza. Na Rys. 2 przedstawiono dwupoziomowy model takiej pamięci, zwanej pamięcią notatnikową lub podręczną [39] (ang. *Cache*).

W pamięci notatnikowej przechowywane są dane, dla których występuje największe prawdopodobieństwo, że będą używane przez potok przetwarzania w następnych cyklach zegara. Jeżeli procesor po wykonaniu operacji dostępu do pamięci danych nie znajdzie ich w pierwszym poziomie pamięci notatnikowej, nastąpi chybienie (ang. Cache Miss). Następnie dana będzie poszukiwana w kolejnym poziomie pamięci notatnikowej, lecz niestety operacja ta będzie trwała dłużej. W najgorszym możliwym wypadku, dana

zostanie pobrana z zewnętrznej pamięci danych i w takim przypadku operacja ta będzie trwała najdłużej.



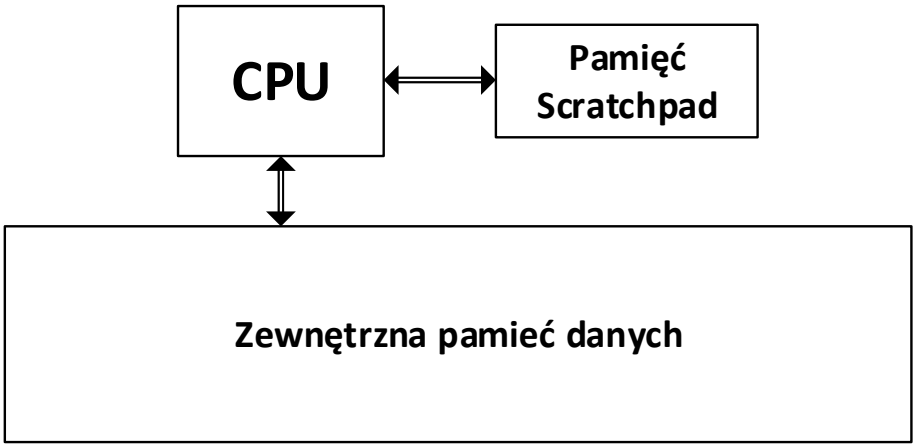
Rys. 2. Model pamięci Cache

Z wyżej wymienionego powodu, czas dostępu do danych w pamięciach notatnikowych, jest bardzo trudny do przewidzenia. Operacja dostępu do pamięci danych, może trwać różny okres czasu, w zależności od poprawności wytypowania danych[40]. Powoduje to, że systemach przewidywalnych czasowo unika się używania tej metody dostępu do pamięci [12], [13], [15], [29], [41] lub ogranicza się jej wpływ na system do minimum [17], [42]–[45]. Ponadto każdy kolejny poziom pamięci notatnikowej, zwiększa zapotrzebowanie systemu na energie [46].

Innym podejściem zapewniające przewidywalność czasową dostępu do pamięci jest wykorzystanie pamięci typu SP (ang. *Scratchpad*) [47]. W rozwiązaniu takim przyjęto, że procesor może komunikować się bezpośrednio z pamięcią lokalną (SP) oraz też bezpośrednio z zewnętrzną, wolniejszą pamięcią danych [48]. Model takiej pamięci pokazano na Rys. 3.

Czas dostępu do pamięci SP oraz pamięci zewnętrznej jest różny (czas dostępu do SP jest znacznie krótszy) [49]. Wadą takiego rozwiązania jest to, że używane dane trzeba podzielić na dwa obszary i zdecydować, które z nich będą używane częściej (aby zapisać je w szybszej pamięci SP), co nie zawsze jest możliwe. Do zapewnienia przewidywalności czasowej, wymagane jest również zaprojektowanie mapy pamięci, zawierającej czasy dostępu do poszczególnych jej fragmentów. Podział ten jednak

zapewnia przewidywalność czasową pozwalającą na precyzyjną, ale skomplikowaną analizę czasu dostępu do pamięci danych.



Rys. 3. Model pamięci Scratchpad

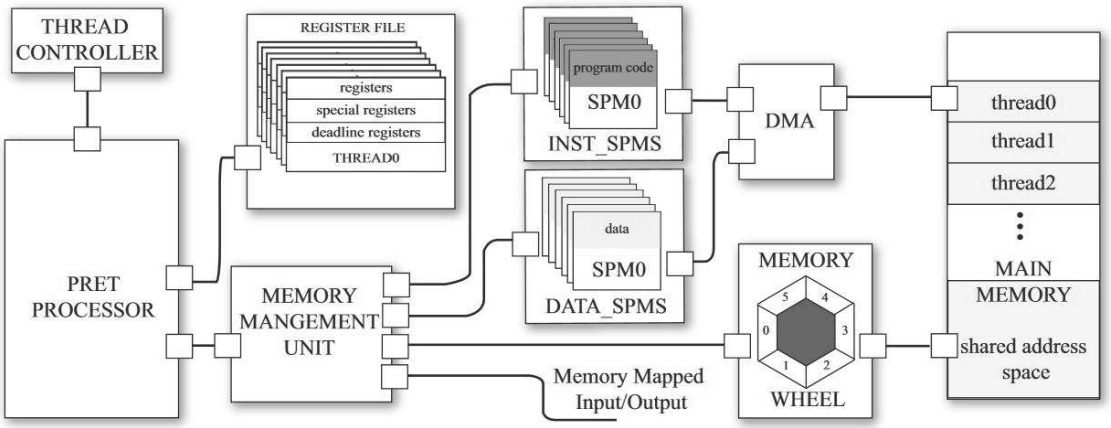
W niniejszej rozprawie postanowiono nie poruszać problemu związanego z różnymi typami pamięci. Projektowany system ma być stosowany w prostych zastosowaniach, gdzie krytyczna przewidywalność czasowa jest głównym warunkiem pracy systemu. Założenia takie powodują, że wykonywane programy muszą być zakodowane w prosty sposób, najlepiej w języku programowania o niskim poziomie abstrakcji. Zadania takie nie wymagają więc bardzo dużych pamięci, a system nie pracuje na wysokich częstotliwościach. Maksymalna częstotliwość pracy aktualnej implementacji systemu wynosi około 250[MHz], co powoduje, że podział na szybszą i wolniejszą pamięć nie jest potrzebny. Zdecydowano więc, że cała pamięć używana w systemie będzie zaimplementowana jako pamięć wewnętrzna układu FPGA (Pamięć jest implementowana za pomocą odpowiedniej dyrektywy, tzw. makra: BRAM_SINGLE_MACRO [50]).

2.3 Koło pamięci

Komunikacja między zadaniami w systemach wielozadaniowych również jest źródłem nieprzewidywalności czasowej. Powodem takiej nieprzewidywalności jest konflikt równoczesnego dostępu dwóch lub więcej zadań do tego samego obszaru pamięci. Równoczesny dostęp jest niemożliwy, dlatego niektóre z tych zadań muszą

oczekiwać na możliwość takiego dostępu. Czas dostępu do takiej pamięci zależy więc od liczby zadań, które żądają/wymagają wymiany danych w określonym przedziale czasu. W nowoczesnych systemach, komunikacja zadań odbywa się poprzez pamięci podręczne, które są kolejnym źródłem nieprzewidywalności czasowej.

Metodą zapewniającą przewidywalność czasową wymiany danych w systemie wielozadaniowym jest koło pamięci [15], diagram blokowy przedstawiający wykorzystania tej metody umieszczono na Rys. 4.



Rys. 4. Przykład użycia koła pamięci. Źródło:[15]

W metodzie koła pamięci, wyznaczony jest wspólny fragment pamięci, służący do wymiany danych z innymi zadaniami. Transmisja danych z procesora do tego fragmentu pamięci oraz odwrotnie, odbywa się przez układ centrum (ang. „hub"), w którym znajduje się koło, zawierające przedziały dla wszystkich zadań pracujących w systemie. Zarówno od strony procesora, jak pamięci, zawsze aktywny jest tylko jeden taki slot. Koło to, cyklicznie i przewidywalnie czasowo zmienia swoje położenie, zmieniając przedziały koła, które mają połączenie z procesorem oraz pamięcią. Wymiana danych zawsze więc zachodzi po obrocie koła o 180 stopni. Znając prędkość obrotu takiego koła oraz liczbę jego przedziałów możliwe jest obliczenie maksymalnego czasu wymiany danych.

Metoda wymiany danych zaproponowana w niniejszej rozprawie została zainspirowana metoda koła pamięci oraz techniką przeplotu zadań. Wykorzystanie fragmentów obu metod pozwoliło zaprojektować autorską metodę wymianę danych.

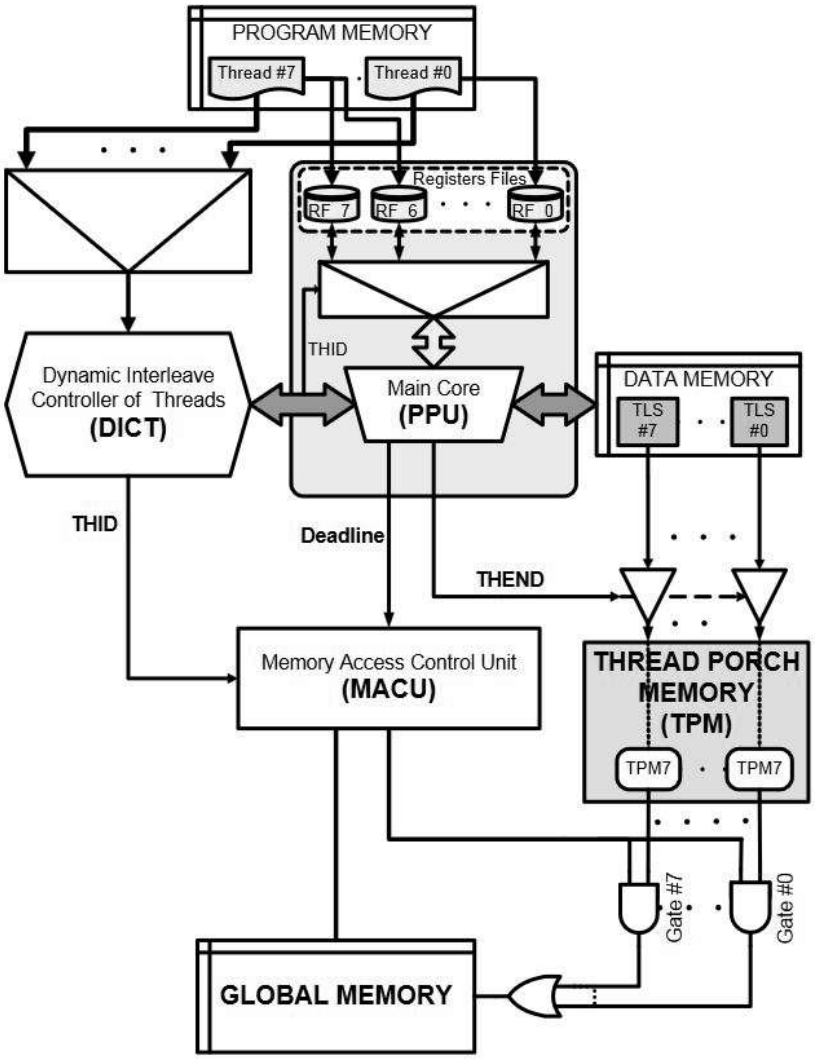
2.4 Kontroler przeplotu

Kontroler przeplotu decyduje o kolejności i częstości dostępów wątków do procesora. Aby zapewnić przewidywalność czasową wykonywania zadań, wątki muszą być tak przeplatane, aby były wykonane przed swoim maksymalnym, krytycznym czasem wykonania (ang. *deadline*).

Projekt takiego kontrolera pozwalający zapewnić pod pewnymi warunkami wykonanie zadań przed ich czasem deadline przedstawiono w [29], [51]. Autorzy zaproponowali dodatkowy etap potoku, którego zadaniem jest wybranie następnego przetwarzanego zadania. Dodatkowo oprócz parametru *deadline*, zaproponowali również dodatkowy parametr *penalty*, który zawiera dodatkowe informacje związane np. z instrukcjami, które mogą powodować brak przewidywalności czasowej systemu. Przykładem instrukcji powodującej nieprzewidywalność czasową może być skok warunkowy. Autorzy zaproponowali dynamiczną metodę wybierania zadania, które ma być przetwarzane w następnym cyklu przetwarzania. Aby zapewnić, że nie dojdzie do zdarzenia, w którym któreś z zadań nie będzie przetwarzane w ogóle, co kilka cykli wybierania zadań metodą dynamiczną, następuje cykl statyczny. W cyklu tym przetwarzane są wszystkie zadania bez względu na parametry zadań.

Przykład architektury zawierającej kontroler przelotu (DICT) zamieszczono na Rys. 5. Kontroler przeplotu odpowiada za generację identyfikatora zadania THID, którego instrukcja ma zostać wykonana w kolejnym cyklu przetwarzania.

Metoda zaprezentowana w niniejszej rozprawie, proponuje przygotowanie kolejności sekwencji przetwarzanych zadań jeszcze przed uruchomieniem systemu. Kolejność ta generowana jest statycznie, na podstawie parametrów zadań. Dzięki takiemu rozwiązaniu można sprawdzić czas wykonania zadań jeszcze przed uruchomieniem systemu, na podstawie zawartości pamięci kontrolera przeplotu. Pozwala to zapewnić (przy założeniu poprawnej implementacji zadań), że wszystkie zadania wykonają się w czasie obliczonym przed uruchomieniem systemu.



Rys. 5. Przykład użycia kontrolera przeplotu. Źródło:[29]

2.5 Rozszerzenie zestawu instrukcji procesora

Jedną z metod zapewniających pod pewnymi warunkami kontrolę nad czasem wykonywania zadań, jest wprowadzenie dodatkowych instrukcji, kontrolujących za pomocą sprzętowych liczników czas wykonywania zadań. Jedną z takich instrukcji, jest instrukcja DEADLINE [14], [52]. Instrukcja ta ma dwa parametry (dr_i oraz v), gdzie dr_i jest numerem sprzętowego rejestru, natomiast v jest jego wartością. Jeżeli wartość dr_i jest równa zero, użycie tej instrukcji powoduje ustawienie wartości v do rejestru dr_i. Następnie, w każdym cyklu zegara wartość ta jest dekrementowana. Jeżeli wartość dr_i jest różna od zera, użycie instrukcji DEADLINE spowoduje zablokowanie przetwarzania,

aż wartość ta osiągnie zero, dopiero wtedy wartość v zostanie wpisana do dr_i i przetwarzanie zostanie odblokowane. Systematyczne wprowadzanie takiej instrukcji do programowanego zadania przez programistę, może zapewniać pod pewnymi warunkami kontrolę nad czasem wykonywania zadania.

W niniejszej rozprawie autor postanowił nie wykorzystywać instrukcji DEADLINE w zaprojektowanym systemie. Po wstępnych eksperymentach [53] uznano, że zaproponowana metodologia statycznego harmonogramowania nie wymaga takiej kontroli czasu wykonania zadania. Dodatkowo instrukcja ta znacząco utrudniłaby użycie zaproponowanej metodologii harmonogramowania zadań. W projektowanym systemie wprowadzono jednak inne instrukcje, np. instrukcję kończącą pracę zadania, pozwalającą zwolnić wykorzystywane zasoby systemu.

3 Założenia wstępne

W rozdziale tym zostaną zaprezentowane założenia implementacji systemu, forma i struktura pisanego kodu, wykorzystana platforma sprzętowa oraz oprogramowanie użyte do implementacji oraz prototypowania systemu.

Po pierwsze, praca dotyczy harmonogramowania statycznego i należy przyjąć założenie, że przed rozpoczęciem projektowania i konfigurowania systemu znane są wszystkie zadania, które mają pracować w systemie. Ponadto zakładamy, że znany jest maksymalny dopuszczalny czas ich wykonania oraz maksymalna liczba instrukcji potrzebna do realizacji danego zadania. Takie dane można pozyskać np. z maksymalnego czasu jego wykonania, znając częstotliwość pracy systemu, zakładając, że zadanie wykonuje się co stałą liczbę cykli zegara. Operowanie taktami zegara, a nie czasem, pozwala uniezależnić sposób opisu złożoności zadania od wydajności systemu. Jak zostanie to pokazane, rozwiązanie zaprezentowane w tej pracy oparte na harmonogramowaniu statycznym, może dać całkowitą przewidywalność czasową pracy systemu. Podczas harmonogramowania zawsze istnieje limit zadań, które jest w stanie wykonać procesor w danym czasie. Dlatego nie znając tych zadań i ich złożoności, zawsze może dojść do obsługi takiego, którego procesor fizycznie nie będzie w stanie wykonać w danym przedziale czasowym.

3.1 Metoda implementacji systemu

System został zamodelowany i zaimplementowany z użyciem języka opisu sprzętu. Dzięki temu możliwe było opracowanie projektu systemu, jego symulacja oraz implementacja w układzie FPGA. Na język implementacji wybrano język Verilog HDL [54], [55], ponieważ jest on bardzo popularny oraz autor rozprawy posiada doświadczenie w pracy z tym narzędziem. Składnia języka Verilog jest stosunkowo prosta, co jest niestety źródłem pewnych problemów [56]. Ze względu na dużą dowolność opisu, dość łatwo można popełniać błędy, które nie są zgłaszane przez kompilator. Drugą poważną wadą języka Verilog jest brak mechanizmów dedykowanych np. do parametrycznej zmiany liczby portów wyjściowych modułu. Wszystko trzeba pisać bardzo prosto, co nie jest zawsze efektywne. Dlatego projektując konfigurowalny system, trzeba stosować pewne, specyficzne rozwiązania pozwalające obchodzić te ograniczenia.

Aby kod był jak najbardziej przejrzysty, korzystano z szeregu założeń podczas jego tworzenia. Każdy z modułów znajduje się w osobnym pliku, o takiej samej nazwie co nazwa modułu, dzięki temu pliki nie są zbyt długie i w szybki sposób można znaleźć interesujący fragment kodu. Wszystkie parametry są pisane wersalikami, a ich wartości przypisywane są tylko na początku pliku, w którym się znajdują.

Wszystkie nazwy parametrów, modułów i sygnałów posiadają nazwy w języku angielskim. Porty wejściowe modułów zawsze posiadają przedrostek „i_", a porty wyjściowe „o_" (z ang. *input* oraz *output*) dla rozróżnienia kierunku przesyłanych informacji. Takie nazewnictwo pomaga również szybko rozróżnić skąd pochodzi dany sygnał. Jeżeli określony sygnał został wygenerowany w danym module, nie posiada on takiego przedrostka. Jeżeli z kolei taki przedrostek występuje, to oznacza, że sygnał został przekierowany z innego modułu lub zostanie przekierowany do innego modułu. Sygnały posiadają taką samą nazwę jak port, z którego pochodzą, ale nie dysponują przedrostkiem „o_" lub „i_".

Każdy z modułów posiada swój kilkuliterowy skrót, np. moduł *select_regs* znajdujący się w pliku *select_regs.v* dysponuje skrótem *sr*. Sygnały pochodzące z portu wyjściowego posiadają w swojej nazwie właśnie taki przedrostek („*_sr"*). Pozwala to na szybszą identyfikację pochodzenia danego sygnału.

Moduły są podzielone na części kombinacyjne oraz rejestry. Dla rozróżnienia rejestru od sygnału, każdy rejestr posiada przedrostek „r_". W części kombinacyjnej wszystkie sygnały są obliczane za pomocą konstrukcji „assign". Jeżeli sygnał ma być sygnałem rejestrowym, najpierw tworzony jest sygnał, którego wartość jest obliczana i przypisywana za pomocą instrukcji „assign". Następnie jest tworzony rejestr, o takiej samej nazwie co sygnał, tylko z przedrostkiem „r_". Później za pomocą konstrukcji „always @ (posedge clk)" do rejestru przypisywany jest wcześniej obliczony sygnał. Taki sposób implementacji daje dużą kontrolę nad strukturą pliku oraz sprzętu, który powstanie po implementacji projektowanego modułu.

3.2 Platforma sprzętowa

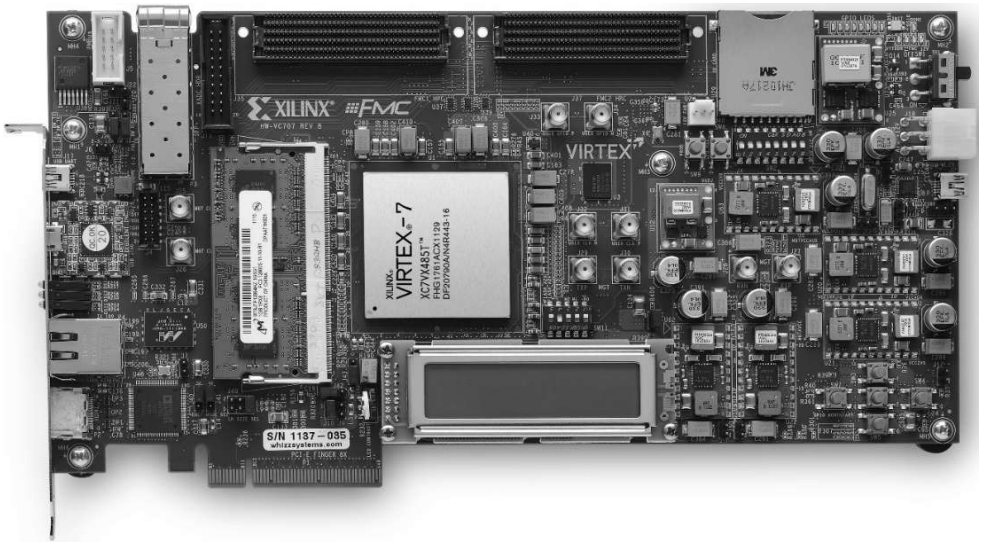
Prototyp projektowanego systemu docelowo będzie zaimplementowany oraz przetestowany w układzie FPGA. Zdecydowano się na układ Virtex-7 XC7VX485T-2FFG1761 firmy Xilinx. Układ ten jest dostępny na płycie prototypowej VC707

Evaluation Kit [57], zakupionej w ramach projektu „CyPhiS: Program nowoczesnych interdyscyplinarnych studiów doktoranckich w dziedzinie systemów cyber-fizycznych”. Przedstawiona na Rys. 6 płyta VC707, w momencie zakupu posiadała układ FPGA z największą dostępną komercyjnie ilością zasobów, co przy tworzeniu konfigurowalnego systemu pozwoliło uruchomić scenariusze z największą liczbą rdzeni i wątków. Dodatkowo płyta ta nie posiada zbędnych w tym zastosowaniu drogich interfejsów światłowodowych i sieciowych zwiększających cenę układu. Płyta ta oprócz samego układu FPGA posiada szereg peryferii takich jak przyciski, diody LED czy wyświetlacz pozwalających na wstępne testowanie oraz debugowanie systemu.

3.3 Oprogramowanie

Ponieważ, jak wspomniano omawiany system został zaimplementowany w języku opisu sprzętu Verilog. Aby było to możliwe, niezbędne jest środowisko obsługujące ten język. Ze względu na dostępność oraz doświadczenie autora, zdecydowano się na dwa programy.

Pierwszy z nich to QuestaSim 10.6a firmy MentorGraphics, w które są wyposażone laboratoria Politechniki Śląskiej. Dzięki temu autor ma swobodny dostęp do tego oprogramowania. Umożliwia ono przeprowadzenie symulacji sprzętu opisanego między innymi językiem Verilog. Na wczesnych etapach projektu był on używany do sprawdzenia koncepcji modelu. Niestety to środowisko nie pozwala na syntezę sprzętu.



Rys. 6. VC707 Evaluation Kit

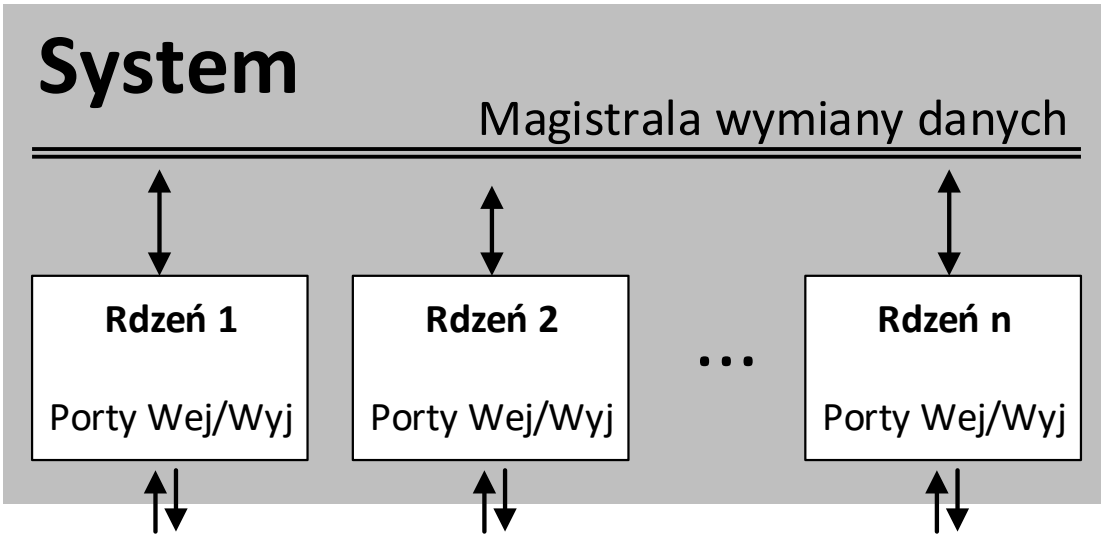
Drugim oprogramowaniem używanym podczas opracowywania rozprawy jest Vivado 2018.3 firmy Xilinx. System Vivado został dostarczony wraz z płytą prototypową VC707. Vivado umożliwia symulację, syntezę oraz możliwość implementacji struktury sprzętowej do układu FPGA [58]. Ten system został wykorzystany do docelowego prototypowania oraz uruchamiania systemu.

Podczas realizacji niezbędne okazało się opracowanie kompilatora programów, uruchamianych w projektowanym systemie. Zadaniem kompilatora jest zamiana instrukcji języka asembler na kod maszynowy systemu. Zaproponowana mikroarchitektura rdzeni różni się na tyle dużo od klasycznych rozwiązań, że użycie gotowych kompilatorów nie jest możliwe. Opracowano również dodatkowe programy wspomagające harmonogramowanie zadań. Programy włącznie z kompilatorem zostały napisane w języku C# za pomocą środowiska Visual Studio 2017.

4 Opracowanie koncepcji systemu

W rozdziale tym zostanie przedstawiony ogólny projekt systemu. Proponowana koncepcja jest rozwiązaniem sprzętowym, wprowadzającym nowy typ architektury systemu PRET. Architektura jest konfigurowalna, wielordzeniowa i wielowątkowa oraz będzie wykorzystywać przeplot wątków. Architektura rdzeni jest wzorowana na rozwiązaniach stosowanych w potokowych strukturach stosowanych przez firmę ARM [59]. Dodatkowo przewidziano możliwość komunikacji między wątkami pracującymi w jednym systemie.

Struktura systemu składa się z jednego lub więcej rdzeni, podłączonych do wspólnej magistrali (Rys. 7). Używana jest do wymiany informacji między wątkami. Rdzenie systemu komunikują się z peryferiami zewnętrznymi za pomocą portów wejścia/wyjścia, które są mapowane jako część pamięci danych każdego wątku. Każdy wątek może mieć swoje indywidualne porty wejścia/wyjścia. Każdy z wątków dysponuje sygnałem zewnętrznym, za pomocą którego można taki wątek uruchomić. Można to zrobić (uruchomić wątek) sprzętowo (za pomocą przerwania z zewnątrz) lub programowo za pomocą innego wątku.



Rys. 7. Struktura systemu

4.1 Charakterystyka systemu

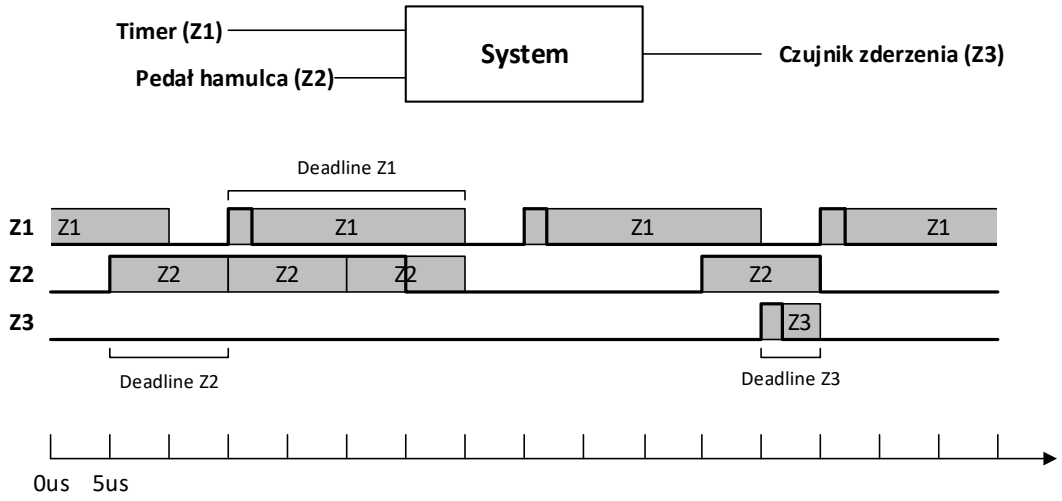
Zadania pracujące w projektowanym systemie, muszą zostać wykonane przed swoim maksymalnym czasem wykonania (ang. deadline), bez względu na obciążenie systemu oraz moment rozpoczęcia pracy zadania. Jest to bezwzględny wymóg systemu pracującego w reżymie przewidywalno-czasowym. Zadania mogą być wywoływane zewnętrznymi sygnałami, których zachowanie jest bardzo trudne lub niemożliwe do przewidzenia. Dlatego system musi być w stanie wykonać zadanie w określonym czasie, licząc ten czas od momentu sygnalizacji potrzeby wykonania zadania.

System musi więc zawsze posiadać wolne zasoby, tak, aby możliwe było poprawne i zgodne z wymogami czasowymi wykonanie zdań nawet w najgorszym scenariuszu. Zdarzeniu, w którym wszystkie zadania pracujące w systemie, zostaną uruchomione w takim przedziale czasowym, który wymusza przetwarzanie wszystkich zadań równocześnie. Jeżeli któreś z zadań zostało już wykonane możliwe jest jego ponowne wykonanie, jeżeli zajdzie taka potrzeba (sygnał uruchamiający dane zadanie będzie ponownie aktywny).

Przykładem zastosowania takiego systemu może być system bezpieczeństwa w samochodzie. W systemie takim może działać zadanie cykliczne, wywoływane co 25[µs] przez sprzętowy timer (Zadanie Z1, na Rys. 8). Zadanie to musi być wykonane w czasie krótszym niż 20[µs] i może obliczać przybliżoną odległość hamowania przy danej prędkości oraz warunkach atmosferycznych. Dodatkowo po każdym wciśnięciu hamulca przez kierowcę, trzeba obliczyć czy, a jeżeli tak, to które koła wpadły w poślizg. Aby systemy bezpieczeństwa takie jak ABS czy kontrola trakcji zareagowały poprawnie, takie zdarzenie musi zostać wykryte nie później niż np. po 10[µs] od momentu wciśnięcia pedału hamulca (Zadanie Z2, na Rys. 8). Koła mogą wpaść w poślizg również po rozpoczęciu hamowania, dlatego sprawdzanie czy koła wpadły w poślizg powinno odbywać się przez cały czas, kiedy hamulec jest wciśnięty (Stan wysoki sygnału Z2). W tym samym czasie może zostać zgłoszony inny sygnał, np. od czujnika zderzenia (Sygnał Z3). Przykładowo w czasie 5[µs], należy sprawdzić czy sygnał nie jest fałszywy oraz jeżeli nie, które poduszki powietrzne uruchomić.

Powyższy przykład mógłby być również zrealizowany za pomocą kilku standardowych procesorów, gdzie każdy wykonuje tylko jedno zadanie. Omawiane w tej pracy rozwiązanie jest alternatywą dla standardowego podejścia. Powinno ułatwić

wymianę danych pomiędzy zadaniami oraz zminimalizować zasoby sprzętowe potrzebne do funkcjonowania systemu.



Rys. 8. Przykład zastosowania systemu

4.2 Model zadania

Każde zadanie w systemie czasu rzeczywistego, charakteryzuje parametr *deadline* określający czas wykonania zadania. W literaturze [60], [61] istnieje podział na 3 typy zadań:

- *Hard* – Są to zadania, których wykonanie po czasie *deadline* może spowodować katastrofalne skutki
- *Firm* – Są to zadania, których wykonanie po czasie *deadline* jest nieprzydatne dla systemu, ale opóźnienie nie powinno spowodować żadnych uszkodzeń systemu
- *Soft* – Są to zadania, których wykonanie po czasie *deadline* jest dalej przydatne dla systemu, ale taka sytuacja może powodować problemy z wydajnością systemu.

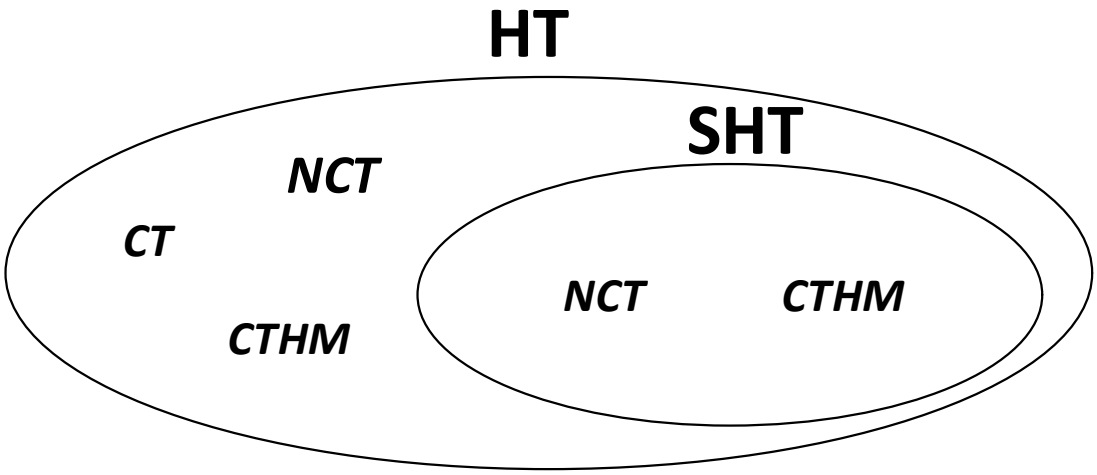
W projektowanym systemie wszystkie zadania są typu *Hard* (HT). Nie jest dopuszczalne, aby któryś z wątków nie wykonał się w czasie *deadline*. Przedstawiana w tej pracy metodologia, pozwoli jeszcze przed uruchomieniem systemu uzyskać takie informacje jak maksymalny i minimalny czas wykonania zadania (maksymalny czas wykonania będzie mniejszy lub równy zakładanemu czasu deadline). Dodatkowo

zdecydowano wprowadzić odrębną kategorię zadań o bardzo precyzyjnym czasie realizacji (*Strong Hard* - SHT). Od zadań tych wymagamy, by były wykonywane dokładnie w określonym czasie (zbyt wczesne zakończenie zadania również nie jest dopuszczalne). Wobec tego podczas harmonogramowania musimy zapewnić, by różnica między maksymalnym a minimalnym czasem wykonania była jak najmniejsza.

Wprowadzono również podział ze względu na to, czy dane zadanie będzie komunikowało się z innymi. Wyodrębniono 3 typy zadań:

- *NCT* – zadania niezależne, które nie współpracują ze sobą
- *CT* – zadania współpracujące, w których czas wymiany danych ma wartość minimalną i maksymalną
- *CTHM* – zadania współpracujące o stałym czasie wymiany danych

Zadania typu CT nie mogą być zadanymi typu SHT. Schemat typów zadań przedstawiono na Rys. 9



Rys. 9. Typy zadań pracujących w systemie

Zadania różnią się między sobą np. długością lub mocą obliczeniową jaką trzeba poświecić do ich wykonania. Aby poprawnie zaplanować wykonanie zadania, należy opisać te parametry. Stopień złożoności zadania można sparametryzować za pomocą oczekiwanego lub maksymalnego czasu wykonania zadania [33]. Jednak wartość ta jest uzależniona od częstotliwości pracy zegara sytemu, na którym był dokonywany pomiar. Dlatego w projektowanym systemie zdecydowano się użyć maksymalnej liczby instrukcji

potrzebnych do wykonania zadania. Wartość tę można zamienić z maksymalnego czasu wykonania zadania, znając liczbę instrukcji zadania, częstotliwość pracy systemu oraz liczbę cykli zegara potrzebnych do wykonania jednego rozkazu, za pomocą wzoru (1).

$$Max_{instr} = \frac{Max_{time}}{\frac{1}{Sys_f} \cdot Inst_{cycles}} \tag{1}$$

gdzie:

Max_{instr} – Maksymalna liczba instrukcji potrzebnych do wykonania zadania

Max_{time} – Maksymalny czas wykonania zadania

Sys_f[MHz] – Częstotliwość pracy systemu

$Inst_{cycles}$ – Maksymalna liczba cykli zegara potrzebnych do wykonania jednej instrukcji

Omawiane rozwiązanie przewiduje możliwość wymiany danych z innym zadaniem. Niestety operacja taka, musi trwać dłużej niż podstawowa instrukcja. Pamięć innego zadania może być zajęta i nie zawsze możliwy jest natychmiastowy dostęp do niej. Dlatego wprowadzono podział na instrukcje standardowe oraz instrukcje dostępu do pamięci innych zadań.

Podsumowując, zadania przeznaczone do pracy w systemie są scharakteryzowane za pomocą 4 parametrów, opisanych za pomocą wzoru (2).

Nie wszystkie zadania wymagają komunikacji z innymi wątkami. Jeżeli parametr M_i, danego zadania wynosi 0, oznacza to, że jest to zadanie niezależne (NCT). W przeciwnym wypadku (M_i różne od 0) jest to zadanie współpracujące (CT lub CTHM). Więcej informacji o typach wątków znajduję się w następnych rozdziałach (5.3.4.3 Typy obsługiwanych wątków).

$$T_i = \{C_i, M_i, D_i, CTHM_i, SHT_i\} \quad (2)$$

gdzie:

C_i – Liczba standardowych instrukcji, potrzebnych do wykonania zadania i (nie obejmuje instrukcji typu M_i)

M_i – Liczba instrukcji wymiany danych z innym wątkiem, potrzebnych do wykonania zadania i

D_i – Maksymalny dopuszczalny czas wykonania zadania i w (deadline)

$CTHM_i$ – W przypadku wartości M_i większej od zera, flaga ta wskazuje czy jest to zadania CT czy CTHM

SHT_i – Flaga typu zadania: 1 – Zadanie typu SHT, 0 – Zadanie typu HT

Zdecydowano się wprowadzić również nowy parametr częstotliwość *i*-tego zadania (ang. *Task Frequency*) TF_i [53], ułatwiający późniejsze harmonogramowanie zadań. Parametr ten rozwinięto względem poprzedniej wersji i przedstawiono we wzorze (3). Poprzednia jego wersje przypomniano we wzorze (32) na stronie 201. Dzięki wprowadzeniu do wzoru wartości $Min_{indistance}$, wartość TF_i jest obliczana dokładniej. Poprzedni wzór zakładał zbyt często bardzo pesymistyczny, najgorszy możliwy przypadek czasu wymiany danych między zadaniami, czyli taki, w którym w systemie pracuje tylko jedno zadanie (tylko w takiej sytuacji liczba taktów była równa liczbie rozkazów). Powodowało to zawyżoną wartość parametru TF_i, co powodowało szybsze przetwarzanie zadania niż w rzeczywistości było potrzebne. Parametr TF_i wprost wskazuje z jaką częstotliwością należy przetwarzać *i*-te zadanie, aby wykonać je przed czasem deadline.

Wartość parametru M_{dur}, zależy od struktury i konfiguracji systemu. Jego pochodzenie zostanie wyjaśnione w następnych rozdziałach (5.3 Magistrala wymiany danych między wątkami). Jego wartość wskazuje maksymalny czas wymiany danych w taktach zegara. Parametr $Min_{indistance}$, oznacza jak często dane zadanie może być przetwarzane (co ile taktów zegara), wartość tego parametru ogranicza architektura systemu. Dzieląc te dwie wartości przez siebie, można uzyskać wskaźnik mówiący o tym

jakiej maksymalnej liczbie standardowych instrukcji odpowiada czas potrzebny na wykonanie wymiany danych w systemie. Dodatkowe dwie instrukcje rekompensują proces opóźnienia startu oraz zatrzymania zadania, spowodowane przetwarzaniem potokowym. Do wykonania pierwszego rozkazu wymagane są jałowe cykle zegara, wypełniające potok przetwarzania etapami pierwszej instrukcji. Opóźnienie spowodowane tymi cyklami zegara można uwzględnić inkrementując liczbę rozkazów. Podobna sytuacja następuje podczas zakończenia pracy zadania. Należy uwzględnić dodatkowy rozkaz, aby umożliwić stopniowe zwalnianie zasobów przez zadanie.

$$TF_i = \frac{C_i + M_i \cdot \left\lceil \frac{M_{dur}}{Min_{indistance}} \right\rceil + 2}{D_i} \tag{3}$$

gdzie:

TF_i[MHz] – Minimalna częstotliwość przetwarzania zadania, pozwalająca wykonać je przed czasem D_i

C_i – Liczba standardowych instrukcji, potrzebnych do wykonania zadania i (bez M_i)

M_i – Liczba instrukcji wymiany danych z innym wątkiem, potrzebnych do wykonania zadania i

M_{dur} – Liczba cykli zegara, potrzebnych do wykonania instrukcji wymiany danych z innym wątkiem

$Min_{indistance}$ – Minimalny przeplot systemu,

D_i[µs] – Maksymalny dopuszczalny czas wykonania zadania i w (deadline)

W odróżnieniu do jednej z pierwszych wersji systemu [62] autor zdecydował się też nie wprowadzać hierarchii zadań. W wymienionej wyżej wersji, do pracy systemu wymagane było zadanie zarządzające i kontrolujące prace całego systemu. W obecnej wersji systemu zadanie takie może wspomagać działanie systemu, ale nie jest wymagane do jego pracy.

4.3 Wykorzystanie przeplotu wątków

Jak już wielokrotnie wspominano, głównym założeniem projektowym jest przewidywalność czasowa systemu. Niestety wykorzystanie architektury procesora potokowego [63], [64], oprócz zalet pochodzących z zwiększonej wydajności, prowadzi do wielu problemów z przewidywalnością czasu wykonania instrukcji [65]. Aby całkowicie nie rezygnować z architektury potokowej, zdecydowano się wprowadzić przeplot wątków pozwalający zachować przewidywalność czasową wykonywanych instrukcji [34].

Każdy z rdzeni ma dedykowaną liczbę wątków sprzętowych, pracujących w przeplocie. Aby umożliwić jak najlepsze dopasowanie wykonywanych zadań do rdzenia, kolejność oraz częstotliwość przetwarzanych wątków jest konfigurowalna. Zadania wykonywane przez wątki są przełączane (przeplatane) zgodnie z kolejnością przechowywaną w rejestrze Cyklu Przeplotu (CP) z Rys. 10. Długość CP również jest konfigurowalna. Przy odpowiedniej konstrukcji CP, jedne wątki można przetwarzać częściej niż inne. Mechanizm taki pozwala tak dopasować rdzeń do zadań, że w jednym rdzeniu będą mogły pracować wątki z bardzo rygorystycznymi ograniczeniami czasowymi oraz te z mniej rygorystycznymi.

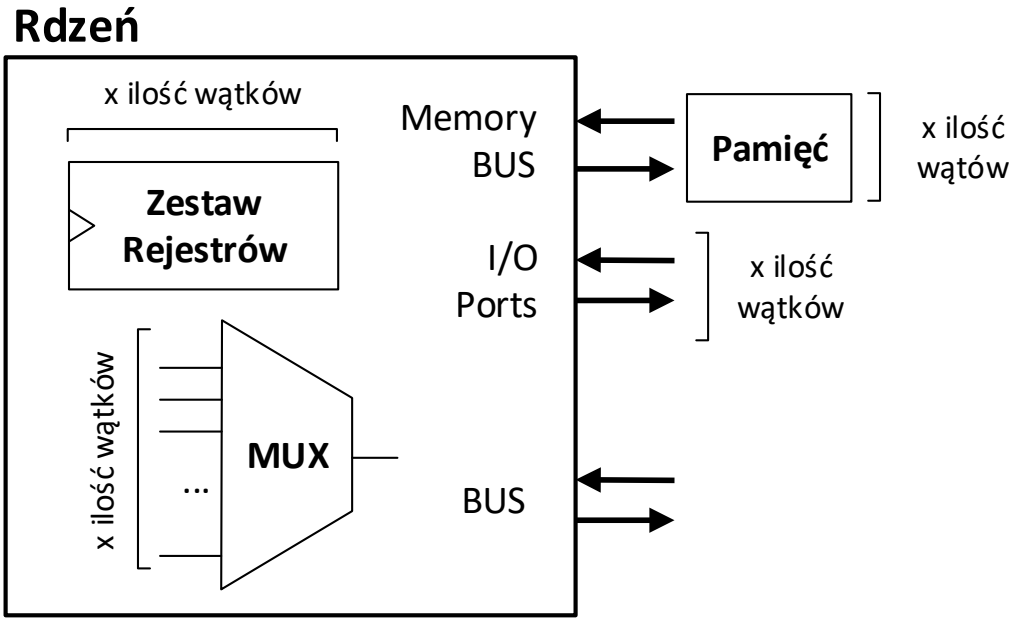
		Cykl przeplotu						Cykl przeplotu					
Cykl zegara		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Numer wątku	**DF**	1	2	3	1	4	5	1	2	3	1	5	4
	ID	-	1	2	3	1	4	5	1	2	3	1	5
	EXE	-	-	1	2	3	1	4	5	1	2	3	1
	MEM	-	-	-	1	2	3	1	4	5	1	2	3
	WB	-	-	-	-	1	2	3	1	4	5	1	2

Rys. 10. Cykl przetwarzania

Na Rys. 10 przedstawiono przykładową konfigurację CP. Ma ona długość 6 wątków oraz przetwarza wątki 1, 2, 3, 1, 4 i 5. Wątek 1 jest więc przeplatany dwukrotnie częściej niż pozostałe wątki 2, 3, 4 i 5.

4.4 Dopasowywanie struktury sprzętowej do zadań

Każdy z rdzeni ma konfigurowalną liczbę obsługiwanych wątków. Wątki te nie są zależne od siebie, więc muszą posiadać swoje niezależne zasoby takie jak pamięć oraz zestawy rejestrów. Aby rdzenie były jak najbardziej zwarte, tylko te niezależne zasoby będą dodatkowymi elementami przypadającymi na każdy wątek. Pozostała część rdzenia, jak np. potok przetwarzania, jest elementem wspólnym dla wątków pracujących w jednym rdzeniu. Na Rys. 11 pokazano schemat konfiguracji rdzenia. Jego rozmiar zależy od parametru oznaczającego maksymalną liczbę pracujących w nim wątków. Od wartości tego parametru zależy liczba zestawów rejestrów, liczba banków pamięci oraz wielkość multiplekserów przełączających zasoby wątków. Pozostała część rdzenia została zaprojektowana w taki sposób, aby nie była zależna od tego parametru. W ten sposób przyrost zapotrzebowania na zasoby wraz z wzrostem liczby obsługiwanych wątków jest minimalny.

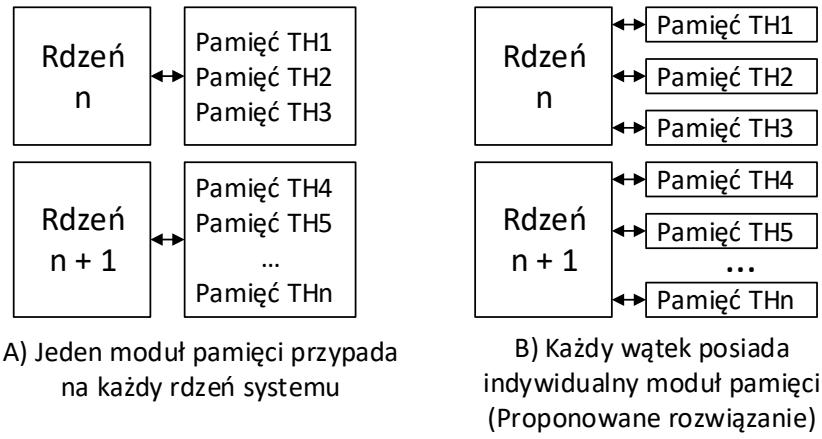


Rys. 11. Konfigurowalność rdzenia

Jeżeli w systemie pracuje kilka rdzeni to przetwarzają one wątki równolegle. Każdy z nich może także mieć inną strukturę. Liczba rdzeni pracujących w systemie również jest konfigurowalna. Projektowany system umożliwia zatem dopasowanie struktury sprzętowej, tj. rozmiarów rdzeni i ich liczby do zadań oraz ich wymagań czasowych.

4.5 Pamięć systemu oraz komunikacja wątków

Główną zmianą w proponowanej architekturze jest wykorzystywanie indywidualnych bloków pamięci dla każdego wątku sprzętowego pracującego w systemie. W standardowym przypadku odrębne moduły pamięci dla każdego rdzenia nie są wymagane, ponieważ podczas używania przeplotu wątków dla każdego cyklu zegarowego przetwarzany jest tylko jeden wątek. Zatem dla każdego taktu zegarowego konieczne jest tylko jedno odwołanie do pamięci wątku. Dlatego zarówno pamięć danych jak i pamięć programów wątków jednego rdzenia, mogą być przechowywane w pojedynczym bloku pamięci (Rys. 12A).



Rys. 12. Porównanie sposobów rozmieszczenia pamięci wątków

Podłączenie pamięci w ten sposób jest bardziej zwięzłe, ponieważ ogranicza liczbę połączeń rdzenia systemu z pamięcią. W przypadku 32-bitowego systemu, każda magistrala pamięci posiada 32-bity danych oraz podobną liczba bitów adresowych pamięci (wartość ta jest zależna od wielkości pamięci). Liczba magistral rośnie więc wraz z liczbą rdzeni, a nie tak jak w przypadku Rys. 12B proporcjonalnie do liczby wątków systemu. Rozwiązanie A jest więc bardziej optymalne pod względem liczby wykorzystanych sygnałów. Większe pamięci posiadają jednak dłuższy czas propagacji

sygnału, co oznacza, że pracują one z mniejszą maksymalną częstotliwością niż pamięci z mniejszą liczbą komórek danych. Wykonywanie instrukcji tego samego wątku co każdy takt zegara, wymaga także podziału pamięci na dwa bloki: blok pamięci danych oraz blok pamięci programu. Dzieje się tak dlatego, ponieważ taka konfiguracja wymaga dwóch odwołań do pamięci w jednym takcie zegara.

W systemach pozwalających na komunikację między zadaniami, rozwiązanie A posiada poważną wadę. W każdym cyklu zegarowym pracy takiego systemu, używana jest wspólna pamięć wątków. Oznacza to, że nie ma prostej metody na wymianę danych między zadaniami. W celu umożliwienia komunikacji wątków, trzeba wykorzystać dodatkowe sprzętowe układy wprowadzające dodatkowe ograniczenia. Jednym z rozwiązań jest wprowadzenie dwubramowych pamięci [66], pozwalających na jednoczesny zapis oraz odczyt dwóch danych do dwóch różnych adresów pamięci. Niestety pamięci takie są drogie, zużywają dużo zasobów oraz wymagają rozwiązania kwestii jedoczesnego zapisu lub odczytu pod tym samym adresem pamięci, co może wprowadzać problem ze spójnością danych systemu. Drugim rozwiązaniem takiego problemu jest cykliczne wprowadzanie przedziałów czasowych, w których rdzeń nie przetwarza rozkazów wątków, tylko wykonuje transfer danych między wątkami. W tym wypadku system jest spowalniany, co nie jest tak dużym problemem jak przewidywalność czasowa takiego rozwiązania. Przerw pracy rdzenia nie można wykonywać zbyt wiele, aby system posiadał dostateczną wydajność. Równocześnie przerw musi być wystarczająco dużo, aby wykonać wszystkie wymagane wymiany danych. Dodatkowo trzeba przewidzieć, które banki pamięci będą wykorzystywane częściej, a które rzadziej. Nie mogą być one jednak wprowadzane tylko wtedy, gdy jest takie zapotrzebowanie. System musi być przewidywalny czasowo, a takich sytuacji nie da się przewidzieć na etapie planowania systemu, gdyż są one zależne od programów oraz od danych wejściowych.

Zaproponowany podział pamięci wątków (Rys. 12B) umożliwia rezygnację z powyżej omawianych dodatkowych układów, kosztem dodatkowych magistral pamięci. Podczas takiego podłączenia pamięci wątków, w każdym cyklu zegara tylko dwie magistrale są zajęte, tzn. te które obsługują zadania znajdujące się w danym momencie w fazach potoku: Instruction Fetch (IF) (z pamięci tego wątku pobierana jest instrukcja do wykonania) oraz Memory Access (MA) (odczyt lub zapis danej do pamięci). Pamięci pozostałych wątków nie są wtedy używane. Można je więc

wykorzystać do wymiany danych z innymi wątkami. Przeplot odbywa się w sposób przewidywalny i zaplanowany przed syntezą układu. Dlatego w prosty sposób można przewidzieć, kiedy do takiej wymiany może dojść. Wymiana danych wykorzystująca taki mechanizm nie zaburza też w żaden sposób pracy rdzenia systemu i jest dla niego transparentna.

Przykład przetwarzania wątków umieszczono na rysunku Rys. 13. Dla uproszczenia schematu umieszczono na nim tylko 5 podstawowych etapów potoku oraz ukazano na nim przeplot 5 wątków.

	Cykl zegara	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Numer Wątku	**Instruction Fetch**	1	2	3	4	5	1	2	3	4
	Instruction Decode (Pamięć używana przez rdzeń)	5	1	2	3	4	5	1	2	3
	Execute	4	5	1	2	3	4	5	1	2
	Memory Access (Pamięć używana przez rdzeń)	3	4	5	1	2	3	4	5	1
	Write Back	2	3	4	5	1	2	3	4	5
	Wątki, których pamięć jest nieużywana przez rdzeń	5	1	2	3	4	5	1	2	3
		4	5	1	2	3	4	5	1	2
		2	3	4	5	1	2	3	4	5

Rys. 13. Okresy używania pamięci wątków przez rdzeń

Analizując Rys. 13 należy zauważyć, że jeżeli w danym takcie zegara (np. 3) dana pamięć jest używana przez rdzeń (wątki 2 i 5), to w następnym takcie (4) jest zawsze wolna. Wyjątkiem od takiej sytuacji jest zdarzenie, w którym fazy ID oraz MA przetwarzają te same wątki w kolejnych taktach zegara. Jest to możliwe tylko w pojedynczych konfiguracjach pokazanych w Tab. 1. Konfiguracje poniżej 12-tu wątków nie mogą być jednak przetwarzane (dla 12-to etapowego systemu minimalna liczba przetwarzanych wątków to 9, więcej w rozdziale 5.1.2.1 Ograniczenia przeplotu wątków), ponieważ będą dla nich występować hazardy danych i sterowania. Problem ten więc w praktycznych konfiguracjach systemu nie występuje.

Tab. 1. Przypadki szczególne używania pamięci przez rdzeń

Długość potoku	Liczba wątków
12	7
11	6
10	5
9	4
8	3

Jeśli przyjmiemy, że na wymianę danych potrzebne są dwa takty zegara, to uzyskamy wydajny, przewidywalny układ wymiany danych w systemie, ponieważ takie rozwiązanie daje pewność, że w którymś z tych taktów pamięć wątku będzie nieużywana. Poprzez przydzielenie każdemu wątkowi indywidualnego modułu pamięci, można uniknąć stosowania wymagających wielu zasobów logicznych pamięci dwubramowych lub skomplikowanych nieprzewidywalnych czasowo układów arbitrażu pamięci.

4.6 Przewidywalność czasowa systemu

Projektując system przyjęto założenie, że każdy z wątków pracujących w systemie ma stały czas wykonania każdej realizowanej instrukcji. Takie rozwiązanie pozwoli uprościć proces harmonogramowania wątków. Czas ten będzie zależny od konfiguracji rdzenia oraz CP (Rys. 10).

Każde zadanie ma takie dane jak: liczba instrukcji potrzebnych do jego realizacji oraz maksymalny dopuszczalny czas jego wykonania. Dysponując tymi informacjami można obliczyć, jak często identyfikator danego wątku i w jakiej kolejności musi być zapisany w CP oraz jak długi musi być CP, aby dane zadanie było wykonane przed zadanym czasem. Upraszczając problem przydziału zadań do rdzeni, jeżeli w danym rdzeniu nie zmieszczą się już kolejne wątki, ponieważ nie ma już wolnych przedziałów czasowych w CP, dokłada się kolejny rdzeń do systemu. W kolejnych rozdziałach (6.3 Podział zadań - strona 90) zostaną przedstawione algorytmy zaprojektowane do przydzielania zadań do rdzeni.

Jedynymi instrukcjami, które są wykonywane dłużej niż te standardowe, są instrukcje wymiany danych z innymi wątkami. Jedną z możliwości wyrównania czasu wykonywania tych instrukcji, było by spowolnienie wykonywania instrukcji standardowych. Zakładając jednak, że statystycznie rzecz biorąc, w zadaniach dominują instrukcje standardowe, takie rozwiązanie powodowałoby znaczne zmniejszenie wydajności systemu.

Z tego powodu zdecydowano się rozgraniczyć instrukcje zadań na instrukcje standardowe (SI) oraz instrukcje wymiany danych z innymi wątkami (MI). Wszystkie instrukcje MI danego wątku będą posiadać stały czas wykonania, który będzie zależny od konfiguracji systemu oraz rdzenia.

Znając liczbę instrukcji SI oraz MI danych wątków, maksymalny czas wykonania zadań oraz czas wykonywania instrukcji SI i MI można zaproponować taką konfigurację systemu, która zapewnia wykonanie zaplanowanych zadań przed podanym limitem czasowym.

4.7 Energooszczędność systemu

Koncepcja konfigurowalności systemu powinna również wpłynąć na jego energooszczędność. W literaturze [67] znana jest zależność określająca zużycie energii dynamicznej w układach CMOS (4). Dysponując układem FPGA, można ograniczyć zapotrzebowanie na moc systemu poprzez ograniczenie aktywności przełączeniowej układu oraz ograniczając częstotliwość zegara systemowego.

$$P_{dynamic} = \alpha \cdot C_L \cdot {V_{DD}}^2 \cdot F_{clk} \tag{4}$$

gdzie:

$P_{dynamic}$ – Moc dynamiczna układu CMOS

α – Aktywność przełączeniowa układu

V_{DD} – Napięcie zasilania

F_{clk} – Częstotliwość zegara

Parametr α można ograniczyć poprzez taką konstrukcję sprzętową systemu, która minimalizuje przełączanie się części układu, która w danym momencie jest nieużywana. Niestety projektowany system jest systemem potokowym, co oznacza, że w każdym cyklu zegara używany jest każdy etap przetwarzania rozkazu. Tylko niezależne zasoby wątków, takie jak pamięć i banki rejestrów nie są używane w każdym cyklu zegara. Jednak dołożenie dodatkowych mechanizmów wyłączających te moduły, poprzez odcinanie od nich sygnału zegarowego, zwiększy skomplikowanie systemu oraz zmniejszy maksymalne, dopuszczalne taktowanie systemu. Głównym celem tej rozprawy nie jest minimalizacja energii, ale przewidywalność czasowa. Dlatego zdecydowano się nie stosować takiej strategii.

Drugim sposobem na ograniczenie mocy rozpraszanej w układzie jest ograniczenie parametru częstotliwości pracy zegara systemowego (F_{clk}) [68]. Ograniczenie tego parametru jest dużo prostsze. Przez to, że system jest konfigurowalny, można dążyć do takiej konfiguracji systemu, w której pracuje jak największa liczba rdzeni. W takim wypadku wątki wykonują swoją pracę równolegle. Pozwala to ograniczyć maksymalne taktowanie pojedynczych rdzeni, co prowadzi do redukcji mocy rozpraszanej w systemie.

4.8 Model harmonogramowania zadań

Zadania są harmonogramowane statycznie, przed właściwą pracą systemu. Umożliwia to planowanie i weryfikację czy system w wybranej konfiguracji jest przewidywalny czasowo. Po procesie harmonogramowania możliwe jest również obliczenie czasów wykonywania zadań jeszcze przed właściwą pracą systemu.

Harmonogramowanie zadań w systemie składa się z kilku etapów. W pierwszym z nich należy przygotować odpowiednie informacje o harmonogramowanych zadaniach (Rys. 14). Następnie, na podstawie wybranego algorytmu obliczana jest liczba rdzeni systemu oraz następuje przyporządkowanie zadań do rdzeni. Algorytmy te mogą dążyć do minimalizacji energii potrzebnej do funkcjonowania systemu, minimalizacji zasobów potrzebnych do jego implementacji lub dążyć do równowagi między powyższymi wymaganiami. W tym etapie, nazwanym podziałem zadań (Rys. 14), obliczana jest również przybliżona częstotliwość pracy każdego z rdzeni (F_{sys}).

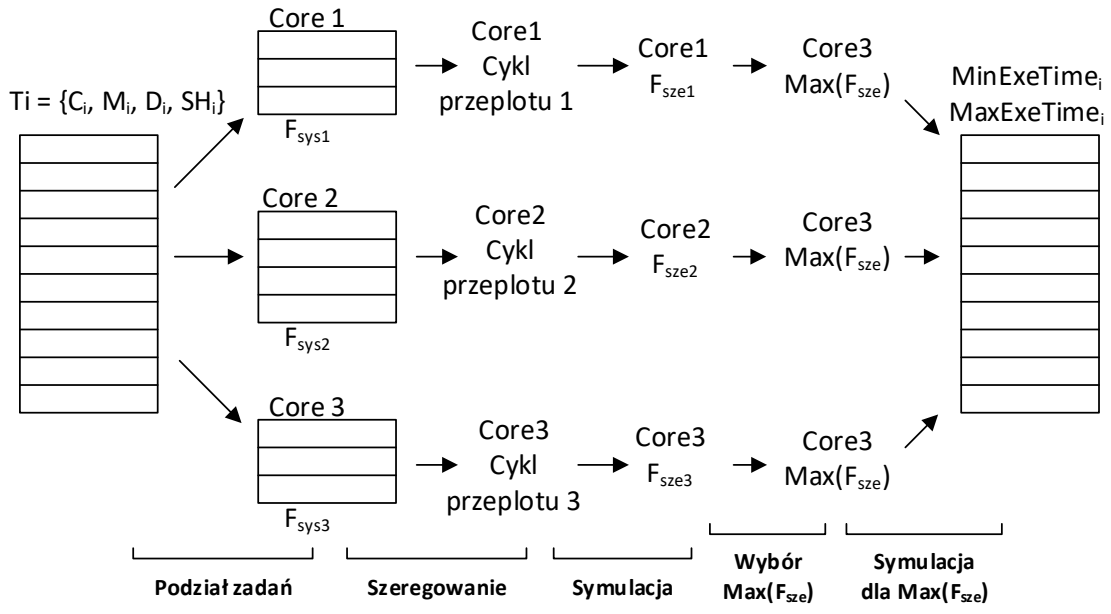
W następnym etapie, nazwanym szeregowaniem wątków, generowany jest cykl przeplotu (CP) (Wykorzystanie przeplotu wątków 4.3). Ustalana jest tu taka kolejność

przeplatania zadań wewnątrz rdzeni, aby wymagania czasowe wątków przy zadanej częstotliwości (F_{sys}) były spełnione.

W etapie symulacja, sprawdzane jest czy wygenerowany CP w każdym rdzeniu, spełnia wymagania czasowe wątków dla częstotliwości wykorzystanych na etapie szeregowania. Jeżeli nie, obliczana jest minimalna częstotliwość (F_{sze}), dla której wszystkie zadania zostaną wykonane przed swoim czasem *deadline*.

Aby rdzenie mogły się ze sobą komunikować, wszystkie z nich muszą pracować z tą samą częstotliwością. W kolejnym etapie, wybierana jest więc największa częstotliwość F_{sze}. Jeżeli wszystkie rdzenie będą pracować z tą częstotliwością, wszystkie zadania będą wykonane przed swoim czasem *deadline*.

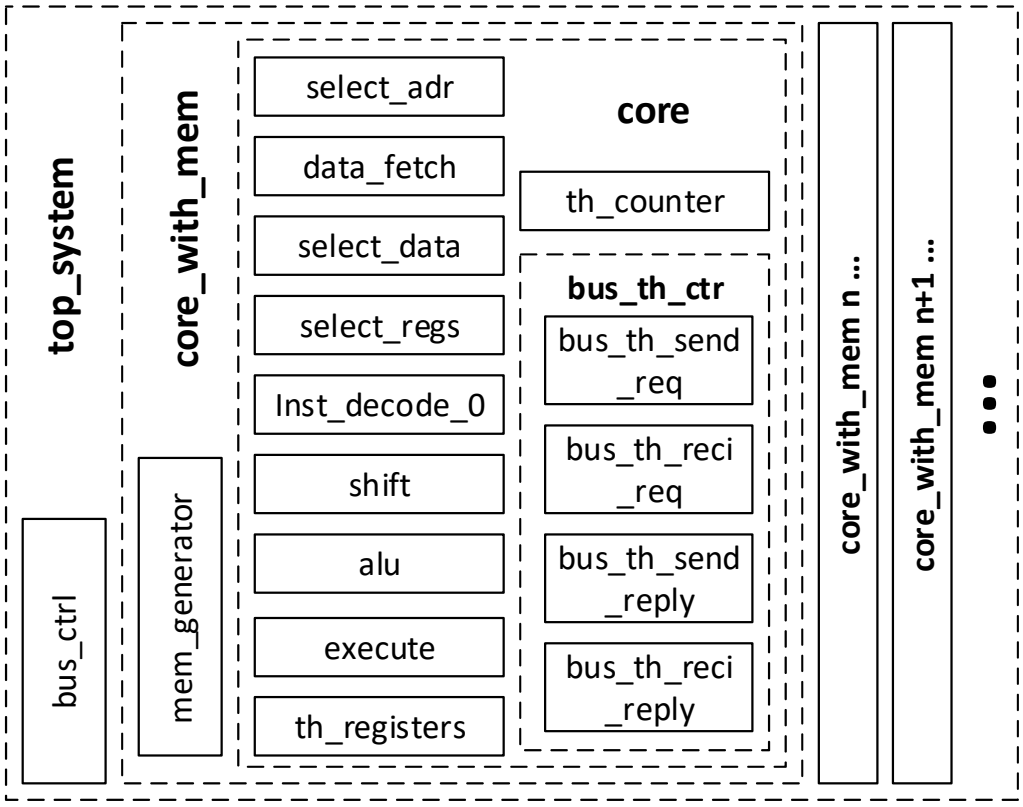
Ostatnim etapem jest finalna symulacja, przeprowadzana dla wszystkich rdzeni pracujących z tą samą częstotliwością (F_{sze}). Podczas symulacji, kolejny raz sprawdzane jest, czy wszystkie zdania zostaną wykonane przed swoim czasem *deadline*. Następnie obliczany jest minimalny i maksymalny czas wykonywania każdego harmonogramowanego zadania.



Rys. 14. Schemat metodologii harmonogramowania zadań

5 Budowa oraz implementacja systemu

W rozdziale tym zostaną omówione poszczególne części projektowanego systemu, takie jak jego architektura, sposób implementacji konfigurowalności rdzenia, czy sposób działania magistrali. Budowany system składa się z szeregu modułów zaprojektowanych za pomocą języka opisu sprzętu Verilog. Aby ułatwić rozeznanie w strukturze budowanego systemu na Rys. 15 pokazano schematyczną strukturę zaprojektowanych modułów. Każdy z nich znajduje się pliku verilog (*.v), o takiej same nazwie co nazwa modułu. Wszystkie moduły „*core_with_mem”* posiadają taką samą strukturę.



Rys. 15. Struktura modułów

5.1 Architektura systemu

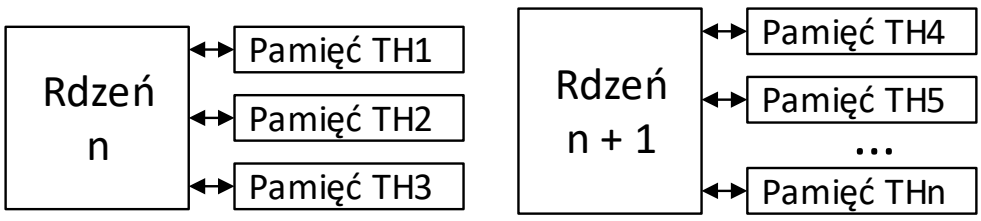
Projektowany system bazuje na architekturze procesorów ARM [69], ale jego mikroarchitektura jest inna. Omawiany system wykorzystuje przeplot wątków. Jest on skonstruowany tak, aby każdą instrukcję zakończyć po okresie jednego cyklu zegara od

zakończenia poprzedniej. Dzięki czemu można efektywnie stwierdzić, że wykonuje on jedną instrukcje na każdy cykl zegara. Wykonanie każdej instrukcji (oprócz instrukcji wymiany danych z innym wątkiem) trwa tyle samo. Struktura instrukcji oraz ich kod maszynowy bazują na ARM ISA [70]. Nie są one jednak zaimplementowane wszystkie oraz nie są one zgodne w stu procentach. Różnice to np. dodatkowe instrukcje pozwalające na kontrolę przewidywalności czasowej lub czas wykonania instrukcji wynikający z mikroarchitektury systemu. W zaimplementowanym systemie, w odróżnieniu do ARM ISA, czas wykonania każdego rozkazu jest taki sam (nie licząc instrukcji komunikujących się z innymi wątkami).

5.1.1 Pamięć oraz rejestry systemu

Każdy z wątków pracujących w systemie posiada swój własny, niezależny zestaw 16 rejestrów. Numerowane są one od R0 do R15 oraz każdy z nich jest 32-bitowy. Rejestr R15 jest rejestrem, który jest licznikiem rozkazów wątku, do którego należy dany bank rejestrów.

Drugim niezależnym zasobem, który posiada każdy wątek jest pamięć. Każdy z nich posiada swój niezależny moduł pamięci. Oznacza to, że jeżeli w danym rdzeniu pracuje 10 wątków, do rdzenia jest podłączonych 10 niezależnych 32-bitowych pamięci (Rys. 16).



Rys. 16. Podłączenie pamięci

Maksymalna szerokość szyny adresowej każdej pamięci to 24-bity, każdy wątek może więc zaadresować ponad 16 tysięcy 32-bitowych komórek. Na potrzeby tej rozprawy korzystano z makra BRAM_SINGLE_MACRO[50] zalecanego przez dokumentacje Xilinx, dostępnego w wykorzystywanym oprogramowaniu. Tylko taka konfiguracja została przetestowana. Pamięci budowane przy pomocy powyższych makr

posiadają 32Kb pamięci, co przy organizacji 32-bitowej oznacza, że każdy z bloków takiej pamięci posiada 1024 komórki. Zdecydowano, że na potrzeby tej rozprawy jest to wystarczające. Możliwe jest połączenie równoległe takich pamięci zwiększające dwukrotnie ich pojemność, jednak obniży to ich maksymalne dopuszczalne taktowanie.

Omawiane wyżej moduły pamięci, są jednocześnie pamięciami programu oraz danych. Aby nie dochodziło do sytuacji w których jedne dane nadpisują instrukcje programów, zaprojektowano parametryczny podział przestrzeni adresowej. W najbardziej standardowej konfiguracji (32Kb pamięci, podział 50% na 50%) podzielono przestrzeń adresową według poniższych adresów:

- Pamięć danych – od 0x001 do 0x100
- Pamięć programu – od 0x101 do 0x1FF

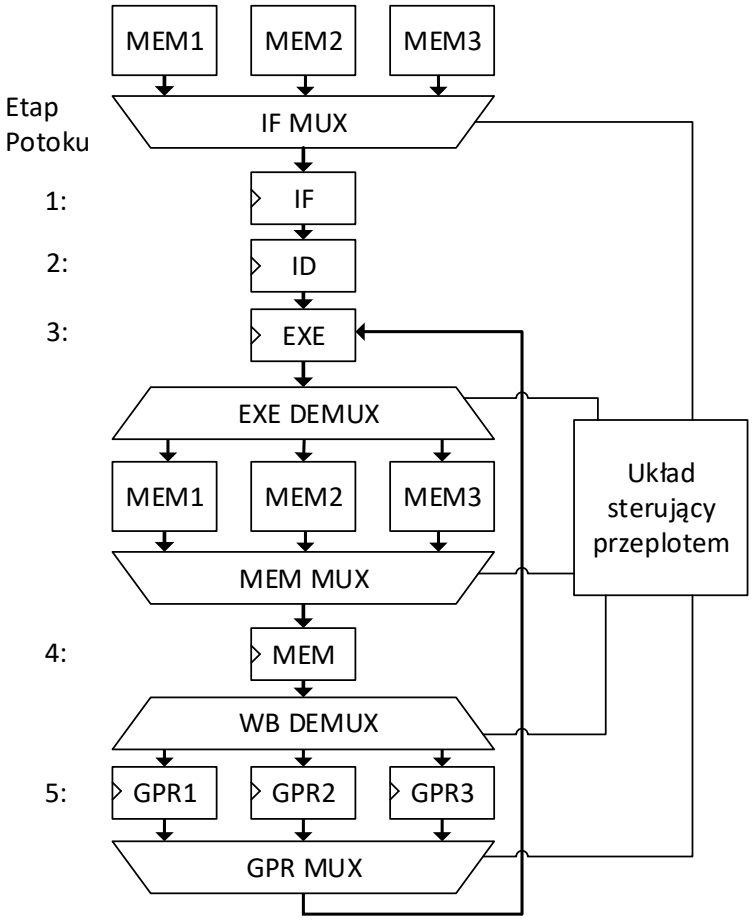
5.1.2 Potok przetwarzania danych

Potok przetwarzania bazuje na klasycznym pięcioetapowym potoku, wykorzystywanym np. w procesorach ARM [69]. Standardowy potok przetwarzania został wzbogacony o przeplot wątków [34]. Każdy z wątków posiada swój niezależny bank rejestrów oraz fizyczny moduł pamięci.

Na etapie projektowania systemu, bardzo trudno jest podzielić etapy potoku przetwarzania procesora tak, aby każdy z nich był przetwarzany podobną ilość czasu. Dzięki takiej filozofii potok będzie najbardziej optymalny. Najlepszą metodą na taki podział jest metoda prób i błędów. Kolejne implementowanie i sprawdzanie najdłuższej ścieżki propagacji danych. Niestety projektując system rekonfigurowalny, takie podejście jest niemożliwe.

Dużym problemem jest przewidzenie czasu propagacji danych w multiplekserach i demultiplekserach przedstawionych na rysunku Rys. 17. System ten jest rekonfigurowalny. Liczba wątków, które są przetwarzane przez ten rdzeń nie jest znana. Prowadzi to do sytuacji, w której nie znana jest wielkość zasobów przełączających dane, a więc i czasu ich propagacji. Gdy wątków przetwarzanych jest mało, czas propagacji WB DEMUX (Rys. 17) może być na tyle niski, że można skleić go z fazą 5 WB. W sytuacji odwrotnej, gdy wątków jest bardzo dużo, takie sklejenie może spowodować duży wzrost czasu propagacji fazy WB doprowadzając do znaczącego spadku częstotliwości taktowania całego systemu. Aby do takiej sytuacji nie dopuścić,

zdecydowano się podzielić przetwarzanie danych na elementarne etapy pokazane na Rys. 18.



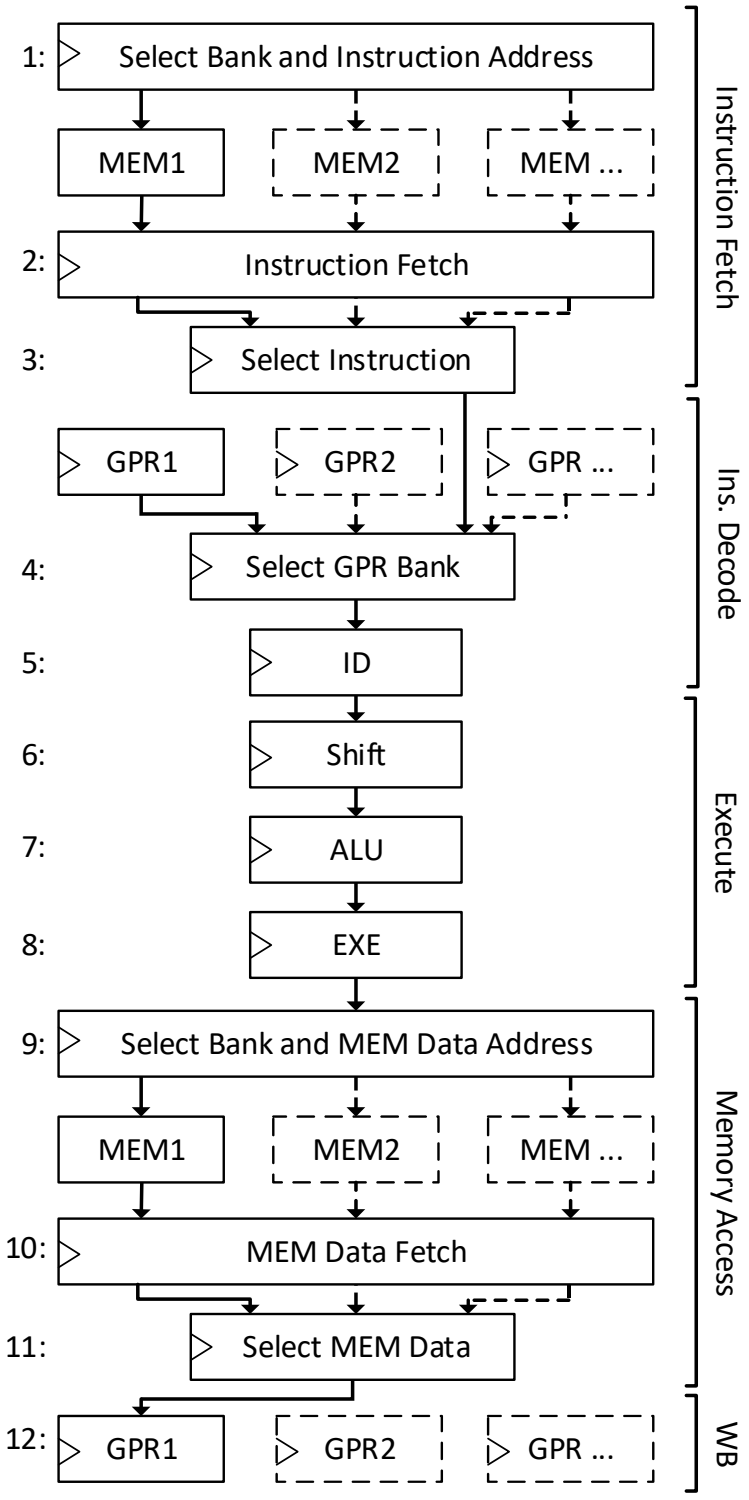
Rys. 17. Podstawowy pięcioetapowy potok wraz z przelotem wątków

Każdy z multiplekserów i demultiplekserów został podzielony na osobny etap przetwarzania. Do ich rozróżnienia każda z takich faz posiada przedrostek „Select”.

Dodatkowo zdecydowano się również podzielić fazę EXE na etap przesuwania bitowego argumentów (faza SFT), wykonywania obliczeń (faza ALU) oraz właściwą fazę EXE (w której podejmowane są decyzje).

Każda z zaprojektowanych faz może być zakończona rejestrem lub nie, w zależności od wartości parametrów podanych podczas syntezy. Gdy sygnały takiego modułu zatrzaskiwane są w rejestrach, moduł ten staje się etapem potoku. Obszerniejszy opis w jaki sposób został zrealizowany ten mechanizm, zostanie omówiony w następnym

podrozdziale 5.2.2 Konfigurowalność potoku. Takie rozwiązanie pozwoli dopasować strukturę potoku do liczby wykonywanych przez niego zadań.

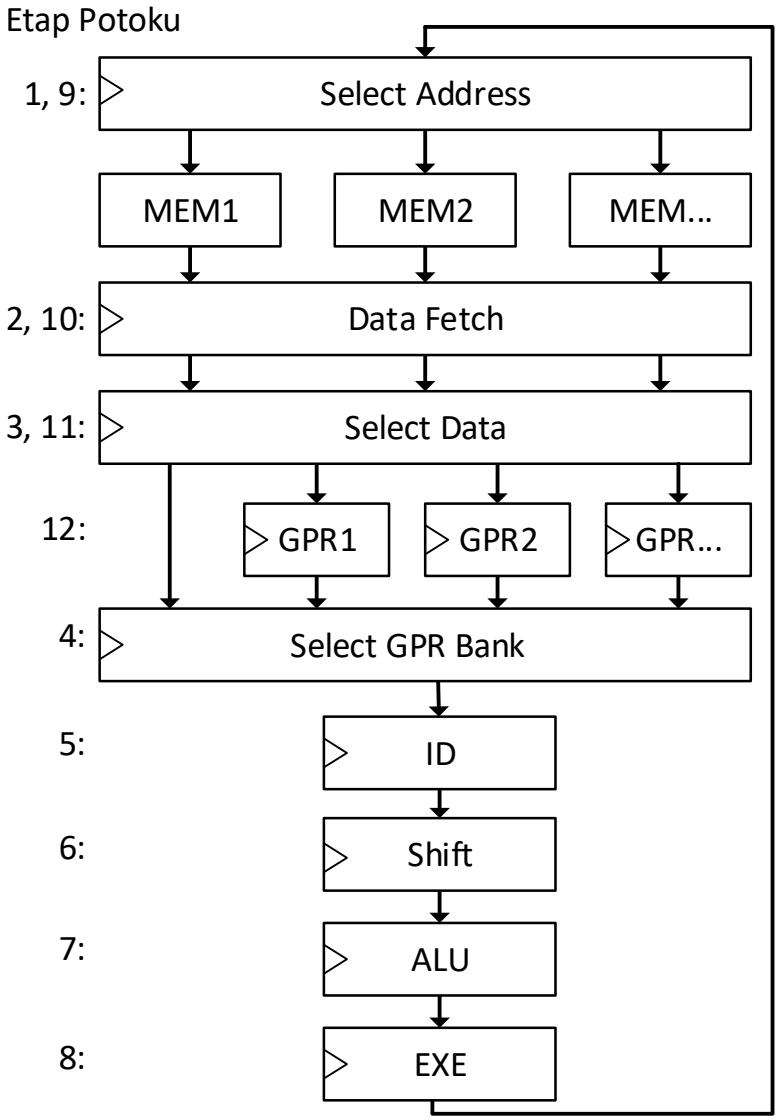


Rys. 18. Podział przetwarzania danych na elementarne etapy

Każdy z wątków pracujących w systemie posiada jeden wspólny fizyczny moduł pamięci, podzielony na pamięć danych i programu. Oznacza to, że podczas wykonywania jednej instrukcji może być konieczne podwójne odwołanie do tej samej pamięci, na Rys. 18 są to fazy 1, 2 oraz 9 i 10. Każda z pamięci posiada tylko jeden interfejs zapisu lub odczytu danych. Nie może być więc ona wykorzystywana w dwóch miejscach potoku przetwarzania naraz. Aby wyeliminować ten problem, zdecydowano się połączyć fazę 1 z 9 oraz 2 z 10.

Ostateczną wersję potoku przedstawiono na Rys. 19. Przetwarzanie danych zaczyna się od fazy pierwszej SA. Sygnały wyjściowe tego etapu są podłączane na linie adresowe wszystkich pamięci pracujących w tym rdzeniu. Zadaniem fazy SA jest podanie odpowiedniego adresu instrukcji na bank pamięci, który należy do przetwarzanego wątku. W fazie drugiej DF dane wyjściowe wszystkich komórek pamięci pracujących w rdzeniu zatrzaskiwane są w rejestrach wyjściowych tej fazy. W fazie trzeciej SD wybierany jest odpowiedni rejestr wyjściowy (fazy DF), w którym jest pobrana instrukcja, której adres był podany w fazie SA. W kolejnej fazie SR pobierane są używane w instrukcji wartości rejestrów z banku rejestrów należącego do przetwarzanego wątku. W fazie ID rozkaz jest dekodowany. Faza SFT opcjonalnie przesuwa dane pobrane z rejestrów. Faza ALU dokonuje obliczeń arytmetycznych oraz logicznych. W etapie EXE podejmowane są decyzje na podstawie wcześniejszych obliczeń. Takie jak: czy rozkaz ma być wykonany, gdzie mają się zapisać dane itp. W 9 etapie potoku wrócono do fazy SA. Tym razem faza ta obsługuje zapis i odczyt danych z pamięci. Na sygnały adresowe banku pamięci, należącego do obsługiwanego wątku, podaje teraz ona adres odczytu komórki pamięci danych lub dane do zapisu i adres, pod którym te dane mają być zapisane. Faza DF realizuje dokładnie to samo zadanie co w fazie 2, przepisuje zawartość sygnałów wyjściowych pamięci do swoich rejestrów. W fazie 11 SD z rejestrów fazy DF jest wybierana dana pobrana wcześniej z pamięci. Ustawiane są też sygnały wybierające odpowiedni bank rejestrów. Następnie w fazie 12 WB zapisywane są dane do właściwych rejestrów ogólnego przeznaczenia.

Należy zwrócić uwagę, że w każdym cyklu zegara faza SA, DF oraz SD obsługuje dwa wątki równocześnie. Jeden podczas pobierania instrukcji, a drugi podczas pobierania/zapisu danej do pamięci.



Rys. 19. Ostateczna wersja potoku przetwarzania

5.1.2.1 Ograniczenia przeplotu wątków

Przeplot wątków został wprowadzony, aby wyeliminować nieprzewidywalności czasowe wynikające z przetwarzania potokowego oraz stosowania mechanizmów i metod, które eliminowały hazardy danych i sterowania w takim przetwarzaniu (2.1 Przeplot wątków – strona 9). Aby takie mechanizmy, podczas używania przeplotu wątków nie musiały być stosowane, wymagana jest minimalna liczba wątków pracujących w przeplocie. W przypadku systemu, o zmiennej liczbie etapów potoku, minimalna liczba wątków nie jest taka oczywista, ponieważ zależy od liczby i rodzajów etapów potoku pracujących w rdzeniu.

Pierwsze ograniczenie pochodzi z potrzeby wyeliminowania hazardu sterowania. Hazard sterowania następuje wtedy, kiedy w następnym cyklu zegara ma nastąpić skok do innej instrukcji, której adres nie został jeszcze obliczony. Adres skoku zostaje obliczony dopiero w fazie EXE przetwarzania. Oznacza to, że kolejna instrukcja tego samego wątku może zacząć być przetwarzana dopiero gdy poprzednia opuści fazę EXE. W przypadku 12 etapowego potoku (Rys. 19) EXE jest 8 etapem potoku. Oznacza to, że minimalna liczba przetwarzanych wątków w takiej konfiguracji to 8.

Drugim ograniczeniem jest wyeliminowanie hazardu danych. Występuje on, gdy instrukcja pobiera zawartość rejestru, który nie został jeszcze nadpisany przez poprzednią instrukcję tego samego wątku. Banki rejestrów są zapisywane w fazie 12 WB. Z kolei zawartość rejestrów pobierana jest w fazie 4 SR (Select GPR Bank na Rys. 19). Oznacza to, że instrukcja tego samego wątku może być przetwarzana w fazie SR, dopiero gdy poprzednia opuści 12 fazę potoku. W przypadku 12 etapowego potoku z Rys. 19, minimalna liczba wątków to 8.

Trzecie ograniczenie pochodzi z architektury systemu i przechowywania pamięci programu oraz danych w tym samym bloku pamięci. Nie jest możliwy odczyt dwóch komórek pamięci w tym samym takcie zegara. System nie będzie pracował prawidłowo, jeżeli Faza SA będzie obsługiwała etap odczytu instrukcji oraz danych tego samego wątku (na Rys. 19 faza SA nie może przetwarzać fazy 1 i 9 tego samego wątku). Dwunastoetapowy potok z Rys. 19 nie może więc przetwarzać 8 wątków.

Za pomocą wzoru (5) można obliczyć minimalną liczbę wątków, które może przetwarzać projektowany system. Wynika to z podsumowania powyższych ograniczeń oraz z faktu, że niektóre fazy potoku muszą występować podwójnie (SA, DF oraz SD).

$$Min_{indistance} = SA_EXE_{num} + 1 \tag{5}$$

gdzie:

$Min_{indistance}$ – Minimalna liczba wątków, które może przetwarzać system, by uniknąć wystąpienia hazardów w potoku

SA_EXE_{num} – Liczba etapów potoku między faza SA a EXE

5.1.3 Sterowanie oraz obsługa przeplotu wątków

W każdym rdzeniu systemu znajduje się moduł „*th_counter”* (Rys. 20). Na wyjściu tego układu generowany jest lokalny numer wątku nazwany **TH_ID**, który ma być przetwarzany w następnym cyklu zegara (sygnał „*o_th_id”*). Kolejność oraz częstotliwość przetwarzania wątków ustawiane są za pomocą parametrów pokazanych na Rys. 20.

```
th_counter #
(
    .TH_NUM                 (TH_NUM),
    .TH_CYCLE_DUR           (TH_CYCLE_DUR),
    .TH_CYCLE_WIDTH         (TH_CYCLE_WIDTH),
    .TH_CYCLE_DUR_WIDTH     (TH_CYCLE_DUR_WIDTH),
    .CP_BRAM_MEM            (CP_BRAM_MEM),
    .CP_BRAM_NUM            (CP_BRAM_NUM),
    .TH_ID_WIDTH            (TH_ID_WIDTH),
    .CORE_TH_IDS            (CORE_TH_IDS     ),
    .CYCLE_IDS_ORDER        (CYCLE_IDS_ORDER)
)
th_counter_i
(
    .i_clk                  (i_clk),
    .i_rst                  (i_rst),

    .o_th_id                (th_id)
);
```

Rys. 20. Przykład modułu typu th_counter

Wartości wszystkich parametrów ustawia się w jednym miejscu, nagłówku modułu głównego „*core_with_mem”.* Na Rys. 21 pokazano fragment nagłówka tego modułu.

Parametr TH_NUM określa liczbę indywidualnych wątków pracujących w danym rdzeniu. Parametr TH_CYCLE_DUR (TH_CYCLE_WIDTH oznacza liczba bitów tego sygnału) określa długość cyklu przeplotu. CORE_TH_IDS przyporządkowuje globalne adresy wątków do danego rdzenia. W parametrze CYCLE_IDS_ORDER zapisana jest kolejność przetwarzanych wątków oraz częstotliwość ich przetwarzania. TH_NUM jest równe liczbie adresów w parametrze CORE_TH_IDS, z kolei TH_CYCLE_DUR liczbie adresów parametru CYCLE_IDS_ORDER. W przypadku kiedy CP jest bardzo długi, dużo wygodniej jest go zapisać pod postacią pamięci zaimplementowanej za pomocą makra BRAM_SINGLE_MACRO[50]. W takim wypadku za pomocą parametru

CP_BRAM_MEM, aktywowana jest opcja odczytu CP z pamięci. Jeżeli takich pamięci jest więcej, można ją wybrać za pomocą parametru CP_BRAB_NUM.

```
core_with_mem #
(
    .TH_NUM             (10),
    .TH_CYCLE_DUR       (11),
    .TH_CYCLE_WIDTH     (4 ),

    ...

    .CP_BRAM_MEM        (0),
    .CP_BRAM_NUM        (0),

    ...

    .CORE_TH_IDS        ({8'd10,8'd9,8'd8,8'd7,8'd6,8'd5,8'd4,8'd3,8'd2,8d'1}),
    .CYCLE_IDS_ORDER    ({8'd10,8'd9,8'd8,8'd7,8'd6,8'd1,8'd5,8'd4,8'd3,8'd2,8d'1}),

    ...
```

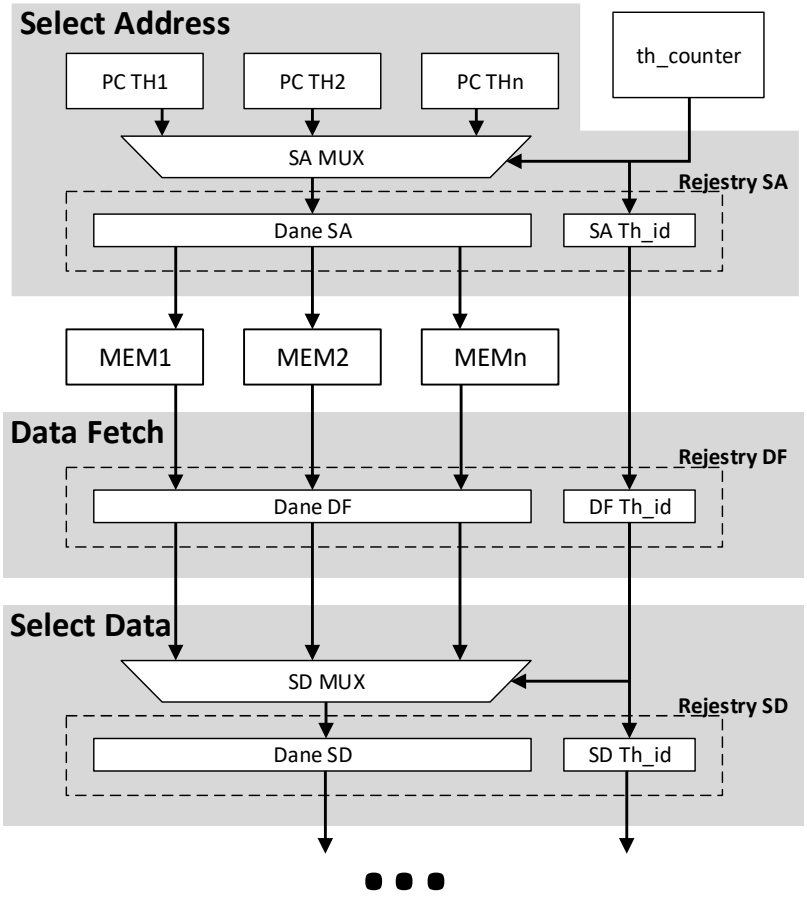
Rys. 21. Parametry przeplotu wątków

W przykładzie z Rys. 21 rdzeń przeplata 10 wątków, o adresach od 1 do 10. Wątki przeplatane są w kolejności od wątku 1 do 5. W kolejnym cyklu przeplatany jest wątek 1, a później wątki od 6 do 10. Taki cykl trwający 11 taktów zegara jest powtarzany w nieskończoność. W tym przykładzie wątek pierwszy jest przeplatany dwukrotnie częściej niż pozostałe. Taka struktura parametrów daje spore możliwości planowania przeplotu.

W każdym z rdzeni panuje podział na adresy wątków globalne i lokalne. Spowodowane jest to tym, że w standardowej konfiguracji globalny adres wątków posiada 8 bitów długości. Rozróżnianie wątków w rdzeniu tak długim sygnałem spowodowało by niepotrzebny spadek wydajności. Dlatego zdecydowano się w każdym rdzeniu nadawać adresy lokalne zaczynające się od 0. Oznacza to, że w systemie pracują wątki o takich samych adresach lokalnych (np. w każdym rdzeniu będzie pracował wątek o lokalnym adresie 0). Adresy te można odczytać z Rys. 21, z parametru CORE_TH_IDS. Pozycja adresu globalnego w tym parametrze nadaje adres lokalny. Na pozycji pierwszej znajduje się adres lokalny 1, na pozycji drugiej adres 2 itd. Na Rys. 21 adresy lokalne wątków są takie same jak adresy globalne.

Rys. 22 przedstawia sposób implementacji przeplotu wątków w trzech pierwszych etapach potoku systemu. Sygnał wejściowy, czyli numer wątku, którego przetwarzanie

ma się zacząć w następnym cyklu zegara, generowany jest w module „*th_counter”*. Następnie wartością tego sygnału sterowany jest SA MUX, który wybiera licznik programu odpowiedniego wątku (PC). Wartość sygnału *„o_th_id”* jest również zapisywana w rejestrach wyjściowych fazy SA. Każda faza potoku posiada sygnał wyjściowy nazwany „*o_N_th_id”* (gdzie N, to symbol fazy potoku), którego stan wskazuje numer lokalny wątku, który jest aktualnie w tej fazie przetwarzany.



Rys. 22. Fragment realizacji przeplotu wątków

Faza SD składa się z multipleksera, wybierającego odpowiednie rejestry wejściowe fazy DF. Są one wybierane na podstawie wątku jaki jest przetwarzany przez fazę SD (jeżeli przetwarzany jest wątek piąty wybierane są rejestry wyjściowe wątku piątego). Numer tego wątku jest zapisany w sygnale „*o_df_th_id”* (DF Th_id na Rys. 22) i tym sygnałem wysterowany jest SD MUX.

5.2 Konfigurowalność systemu

Projektowany system za pomocą parametrów w nagłówku modułu „*core_with_mem”* (który jest głównym modułem rdzenia) dopasowuje swoją strukturę do liczby rdzeni, które mają być w nim przetwarzane. Nagłówek ten został przedstawiony na Rys. 23. Za pomocą dodatkowych parametrów można również ustalić, które z etapów przetwarzania mają być etapami potoku. Niektóre z parametrów (TH_ADR_WIDTH, DATA_WIDTH oraz MEM_ADR_WIDTH), posiadają tak zwane wartości domyślne. Oznacza to, że podczas projektowania systemu przewidziano możliwość ich konfiguracji, ale ze względu na mnogość różnych scenariuszy ich wartości, konfiguracje inne niż domyślne nie zostały testowane. Wszystkie omawiane w tej pracy konfiguracje bazują na 32 bitowym systemie (DATA_WIDTH = 32), 24 bitowej szynie adresowej pamięci (MEM_ADR_WIDTH = 24) oraz 8 bitowych adresach globalnych wątków (TH_ADR_WIDTH = 8).

```
core_with_mem #
)(
    .TH_NUM             (10),
    .TH_CYCLE_DUR       (10),
    .TH_CYCLE_WIDTH     (4 ),
    .TH_ADR_WIDTH       (8),
    .DATA_WIDTH         (32),
    .MEM_ADR_WIDTH      (24),

    .SA_REGS_OUT        (1 ),
    .DF_REGS_OUT        (1 ),
    .SD_REGS_OUT        (1 ),
    .SR_REGS_OUT        (1 ),
    .ID0_REGS_OUT       (1 ),
    .ST_REGS_OUT        (1 ),
    .ALU_REGS_OUT       (1 ),
    .EXE_REGS_OUT       (1 ),
```

Rys. 23. Parametry konfigurujące rdzeń

5.2.1 Konfigurowalność struktury sprzętowej

Wybrany do implementacji systemu język opisu sprzętu Verilog, niestety nie jest zbyt elastyczny. Głównym problem przy implementacji systemu z konfigurowalną liczbą obsługiwanych wątków, było ograniczenie języka niepozwalające uzależnić liczby

portów wejściowych i wyjściowych rdzenia, od liczby wątków obsługiwanych przez rdzeń. Jest to wymagane, ponieważ nawet przy podłączaniu pamięci do rdzenia, portów do jej podłączenia musi być tyle ile wątków obsługiwanych przez rdzeń.

To ograniczenie zostało pominięte poprzez zastosowanie zmiennej szerokości portów w modułach. Sposób ten zostanie omówiony na przykładzie fazy SR, czyli fazy, w której jest wybierany bank rejestrów aktualnie przetwarzanego wątku. Moduł SR (Rys. 24) od linii 37 do 47 posiada porty wejściowe o konfigurowalnej szerokości. Są to sygnały zawierające wartości zapisane w bankach rejestrów wszystkich wątków pracujących w danym rdzeniu. Każdy z wątków posiada 15 rejestrów i dodatkowo rejestr flag (rejestr CPSR), ale w danej instrukcji wykorzystywane są tylko te, które mogą być zaadresowane w wykonywanej instrukcji. W liście rozkazów ARM ISA, adresy rejestrów występują zawsze na tych samych bitach rozkazu. Do fazy SR są więc tylko przekazywane te rejestry. Dla przykładu port wejściowy „*i_reg_1916”* (linia 37 Rys. 24) oznacza dane z rejestrów zaadresowanych na 16, 17, 18 i 19 bicie instrukcji.

```
30      input                                   i_clk,
31      input                                   i_rst,
32
33      input  [TH_CYCLE_WIDTH-1:0]             i_sd0_th_id,
34
35      input  [DATA_WIDTH-1:0]                 i_sd0_data,
36
37      input  [(TH_NUM*DATA_WIDTH)-1:0]        i_reg_1916,
38      input  [(TH_NUM*DATA_WIDTH)-1:0]        i_reg_1512,
39      input  [(TH_NUM*DATA_WIDTH)-1:0]        i_reg_1108,
40      input  [(TH_NUM*DATA_WIDTH)-1:0]        i_reg_0300,
41      input  [(TH_NUM*DATA_WIDTH)-1:0]        i_pc,
42      input  [(TH_NUM*32)-1:0]                i_cpsr,
43
44      output [DATA_WIDTH-1:0]                 o_sr_instruction,
45
46      output [DATA_WIDTH-1:0]                 o_sr_reg_1916,
47      output [DATA_WIDTH-1:0]                 o_sr_reg_1512,
48      output [DATA_WIDTH-1:0]                 o_sr_reg_1108,
49      output [DATA_WIDTH-1:0]                 o_sr_reg_0300,
50      output [DATA_WIDTH-1:0]                 o_sr_pc,
51      output [31:0]                           o_sr_cpsr,
52
53      output [TH_CYCLE_WIDTH-1:0]             o_sr_th_id
```

Rys. 24. Sygnały modułu SR

Ten port wejściowy („*i_reg_1916”*) posiada szerokość bitową zależną od dwóch parametrów, TH_NUM oraz DATA_WIDTH, który w testowanych konfiguracjach zawsze jest równy 32. Każdy z rejestrów tego systemu jest również 32 bitowy. TH_NUM określa liczba wątków obsługiwanych przez rdzeń (Rys. 23). Sygnał ten ma więc szerokość równą 32 bitom pomnożonym przez liczbę wątków pracujących w systemie.

Na każdych 32 bitach sygnału „*i_reg_1916”* występują dane rejestru innego wątku: od bitu 0 do bitu 31 wątek pierwszy, od 32 do 63 wątek drugi itd. W ten sposób można ominąć ograniczenia języka opisu sprzętu zachowując stałą liczba portów modułu.

Na Rys. 25 pokazano fragment realizacji przełączania banków rejestrów aktualnie przetwarzanego wątku w fazie SR. Sygnał „*reg_1916”* jest już wartością rejestru aktualnie przetwarzanego wątku. W linii 70 następuje przypisanie do niego odpowiedniej wartości. Najpierw (linia 70) sprawdzany jest warunek czy sygnał „*i_sd0_th_id”*, który zawiera adres lokalny wątku zatrzaśniętego w fazie SD, nie jest większy od liczby rdzeni przetwarzanych w systemie. Jest to zabezpieczenie przed złą konfiguracją systemu. Następnie (linia 71) z ciągu wektorów zawierającego dane wszystkich wątków („*i_reg_1916”*) wybierany jest ten, który znajduje się na pozycji „*i_sd0_th_id”,* czyli aktualnie przetwarzanego wątku. W ten sposób są realizowane wszystkie multipleksery wybierające dane przetwarzanego wątku

```
58 wire  [DATA_WIDTH-1:0]                    reg_1916;
59 wire  [DATA_WIDTH-1:0]                    reg_1512;
60 wire  [DATA_WIDTH-1:0]                    reg_1108;
61 wire  [DATA_WIDTH-1:0]                    reg_0300;
62 wire  [DATA_WIDTH-1:0]                    pc;
63 wire  [DATA_WIDTH-1:0]                    cpsr;
64 wire  [DATA_WIDTH-1:0]                    instruction;
65
66 //------------------------------------------------------------------------
67 // MUXY
68 //------------------------------------------------------------------------
69
70 assign reg_1916 = (i_sd0_th_id >= TH_NUM) ?
71   {DATA_WIDTH{1'b0}} : i_reg_1916[i_sd0_th_id*DATA_WIDTH +: DATA_WIDTH];
72
73 assign reg_1512 = (i_sd0_th_id >= TH_NUM) ?
74   {DATA_WIDTH{1'b0}} : i_reg_1512[i_sd0_th_id*DATA_WIDTH +: DATA_WIDTH];
75
```

Rys. 25. Realizacja konfigurowalnego multipleksera

Dla szybszego dekodowania, większość sygnałów sterujących w rdzeniu jest zapisana w kodzie ONEHOT. Niestety takie kodowanie uniemożliwia zaprojektowanie skalowalnego multipleksera, tak jak na Rys. 25. Przykładem takiego sygnału może być „*i_df_btc_get_acces”* w fazie potoku SD, który sygnalizuje, że magistrala uzyskała dostęp do pamięci wątku tego systemu oraz wskazuje, który to wątek. Każdy bit tego sygnału przyporządkowany jest do jednego wątku pracującego w rdzeniu, więc jego długość jest równa liczbie wątków obsługiwanych przez rdzeń. Sygnał ten został zainicjowany w linii 4 na Rys. 26.

Kolejnym ograniczeniem języka opisu sprzętu Verilog jest to, że operatorem operacji part-select („[]”) musi być sygnał kodowany binarnie. Aby obejść to ograniczenie, zaprojektowano funkcje „*oh_to_dec”* (linia 9, Rys. 26), która zamienia kod ONEHOT na kod binarny. W ten sposób wysterowano konfigurowalny multiplekser z linii 18.

```
1  //...
2  input [(DATA_WIDTH*TH_NUM)-1:0]    i_df_data;
3  //...
4  input [TH_NUM-1:0]                 i_df_btc_get_access,
5  //...
6
7  //...
8  //Funkcja onehot to dec
9  function integer oh_to_dec;
10     input grey_num;
11     integer grey_num,j;
12     for (j=0; j<32; j=j+1) if (grey_num[j]) oh_to_dec = j;
13 endfunction
14
15 wire [(DATA_WIDTH)-1:0]            btc_data;
16
17 assign btc_data = (i_df_btc_get_access) ?
18         i_df_data[(oh_to_dec(i_df_btc_get_access))*DATA_WIDTH +: DATA_WIDTH]:
19         i_df_btc_data;
20 //...
```

Rys. 26. Sterowanie konfigurowalnym multiplekserem za pomocą kodu ONEHOT

Każdy z wątków pracujących w systemie posiada swoje niezależne zasoby, takie jaki jak np. zestaw rejestrów. Moduły takie zdecydowano się zaprojektować pojedynczo, tak, aby obsługiwały tylko jeden wątek. Do każdego wątku zainicjowano oraz podłączono jeden taki moduł używając konstrukcji „*generate”* (Rys. 27). Pozwala ona wygenerować dokładne kopie fragmentu kodu oraz użyć do tego pętli „*for*”. Taka konstrukcja pozwala wygenerować konfigurowalną liczbę banków rejestrów zależną od parametru TH_NUM.

```
1 genvar i;
2
3  generate
4  for (i = 0; i < TH_NUM; i = i + 1)
5   begin:reg_bank
6    register_bank #
7     (
8         .DATA_WIDTH       (DATA_WIDTH),
9         .RD_SEL_WIDTH     (RD_SEL_WIDTH)
10    )
11   register_bank_i
12    (
13        .i_clk            (i_clk),
14        .i_rst            (i_rst),
15
16        //...
17
18        .i_sd0_rsel_1916 (i_sd0_rsel_1916),
19        .i_sd0_rsel_1512 (i_sd0_rsel_1512),
20        .i_sd0_rsel_1108 (i_sd0_rsel_1108),
21        .i_sd0_rsel_0300 (i_sd0_rsel_0300),
22
23        //...
24
25        .o_sr_reg_1916    (o_sr_reg_1916[i*DATA_WIDTH +: DATA_WIDTH]),
26        .o_sr_reg_1512    (o_sr_reg_1512[i*DATA_WIDTH +: DATA_WIDTH]),
27        .o_sr_reg_1108    (o_sr_reg_1108[i*DATA_WIDTH +: DATA_WIDTH]),
28        .o_sr_reg_0300    (o_sr_reg_0300[i*DATA_WIDTH +: DATA_WIDTH]),
29        .o_sr_reg_pc      (o_sr_reg_pc[i*DATA_WIDTH +: DATA_WIDTH]),
30        .o_sr_reg_cpsr    (o_sr_reg_cpsr[i*DATA_WIDTH +: DATA_WIDTH])
31
32   );
33   end
34 endgenerate
```

Rys. 27. Instancje banków rejestrów

Używając trzech powyższych konstrukcji zaprojektowano konfigurowalny rdzeń systemu czasu rzeczywistego, w którym liczba obsługiwanych wątków zależy od parametru TH_NUM (Rys. 23).

5.2.2 Konfigurowalność potoku

Każdy z etapów przetwarzania rdzenia może być fazą potoku. W takiej sytuacji etap taki musi posiadać swoje rejestry wyjściowe oraz zapisać w nich swoje sygnały wyjściowe. Aby umożliwić konfigurację czy dana faza ma być etapem potoku czy nie, wykorzystano konstrukcję „*generate”*. Na Rys. 28, na przykładzie fazy ALU pokazano fragment realizacji takiej konstrukcji. Przykładowy sygnał wyjściowy fazy ALU „*o_alu_log_alu_out”* zainicjowano w linii 3. Następnie zainicjowano sygnał jemu odpowiadający „*log_alu_out”* (linia 4). W linii 7 następuje wyliczenie wartości tego sygnału. Następnie rozpoczyna się konstrukcja „*generate”* (linia 11) zależna od

parametru REGS_OUT. Jeżeli jego wartość wynosi 1 tworzony jest rejestr „*r_log_alu_out”* (linia 14) i w następnym takcie zegarowym zostaje w nim zatrzaśnięta wcześniej obliczona wartość (linia 24). Na końcu rejestr jest łączony z sygnałem wyjściowym modułu (linia 28). Jeżeli ten etap przetwarzania nie ma być etapem potoku (REGS_OUT = 0) żadne rejestry nie są tworzone i obliczony wcześniej sygnał zostaje podłączony z kombinacyjnym wyjściem modułu *o_alu_log_alu_out* w linii 33.

Parametry wszystkich faz przetwarzania są przekierowywane do nagłówka modułu *core_with_mem* (Rys. 23). W tym miejscu, za pomocą parametru 1 lub 0, można ustawić czy dana faza przetwarzania ma być etapem potoku czy nie.

```
1       output [DATA_WIDTH-1:0]     o_alu_log_alu_out,
2       //...
3
4       wire [DATA_WIDTH-1:0]       log_alu_out;
5       //...
6
7       assign log_alu_out = (i_st_opcode == AND) ? op_and :
8                            (i_st_opcode == ORR) ? op_orr :
9       //...
10
11  generate
12      if (REGS_OUT)
13          begin : reg_outs_df0_th_id
14              reg [DATA_WIDTH-1:0]        r_log_alu_out;
15              //...
16              always @ (posedge i_clk)
17                  if (i_rst)
18                      begin
19                          r_log_alu_out <= {DATA_WIDTH{1'b0}};
20                          //...
21                      end
22                  else
23                      begin
24                          r_log_alu_out <= log_alu_out;
25                          //...
26                      end
27
28              assign o_alu_log_alu_out = r_log_alu_out;
29              //...
30          end
31      else
32          begin : comb_outs_df0_th_id
33              assign o_alu_log_alu_out = log_alu_out;
34              //...
35          end
36  endgenerate
```

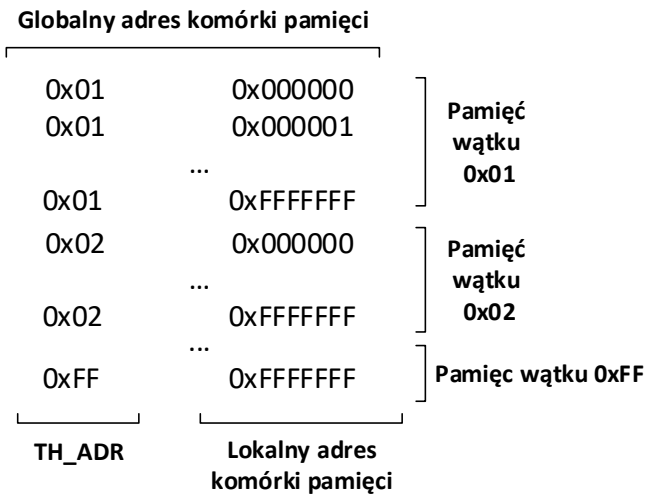
Rys. 28. Konfigurowalne wyjścia rejestrowe faz potoku

5.3 Magistrala wymiany danych między wątkami

W kolejnych podrozdziałach zostaną omówione dwie wersje magistral wykorzystujących koncepcję opisaną w rozdziale 4.5 Pamięć systemu oraz komunikacja wątków. Wersja jednokierunkowa powstała po zdobyciu doświadczenia podczas projektowania i implementowania magistrali dwukierunkowej. Jest ona bezpośrednio ulepszoną wersją pierwotnej koncepcji. W rozdziale tym zostanie również omówiony sposób adresacji wątków oraz mechanizm wysyłania zapytania wymiany danych.

5.3.1 Adresacja wątków

Do umożliwienia wymiany danych między wątkami niezbędna jest prosta identyfikacja wątków. Każdemu wątkowi pracującemu w systemie nadano indywidualny adres nazwany **TH_ADR**, który jest globalnym identyfikatorem danego wątku w całym systemie. Maksymalna liczba pracujących wątków w systemie jest nieograniczona (ustawiana za pomocą parametru), ale na potrzeby tej rozprawy oraz przez ograniczony czas na testy konfigurację ograniczono do 8-bitów adresu TH_ADR. Oznacza to, że można zaadresować maksymalnie 255 wątków, ponieważ adres 0x00 jest adresem zastrzeżonym. Każdy wątek w testowych konfiguracjach posiada maksymalnie 24-bity adresu pamięci adresu do zaadresowania. Po połączeniu 8-bitowego TH_ADR oraz 24-bitowego adresu komórki otrzymuje się globalny adres każdej komórki pamięci w systemie (Rys. 29).



Rys. 29. Adresacja wątków

5.3.2 Proces wysyłania zapytania wymiany danych

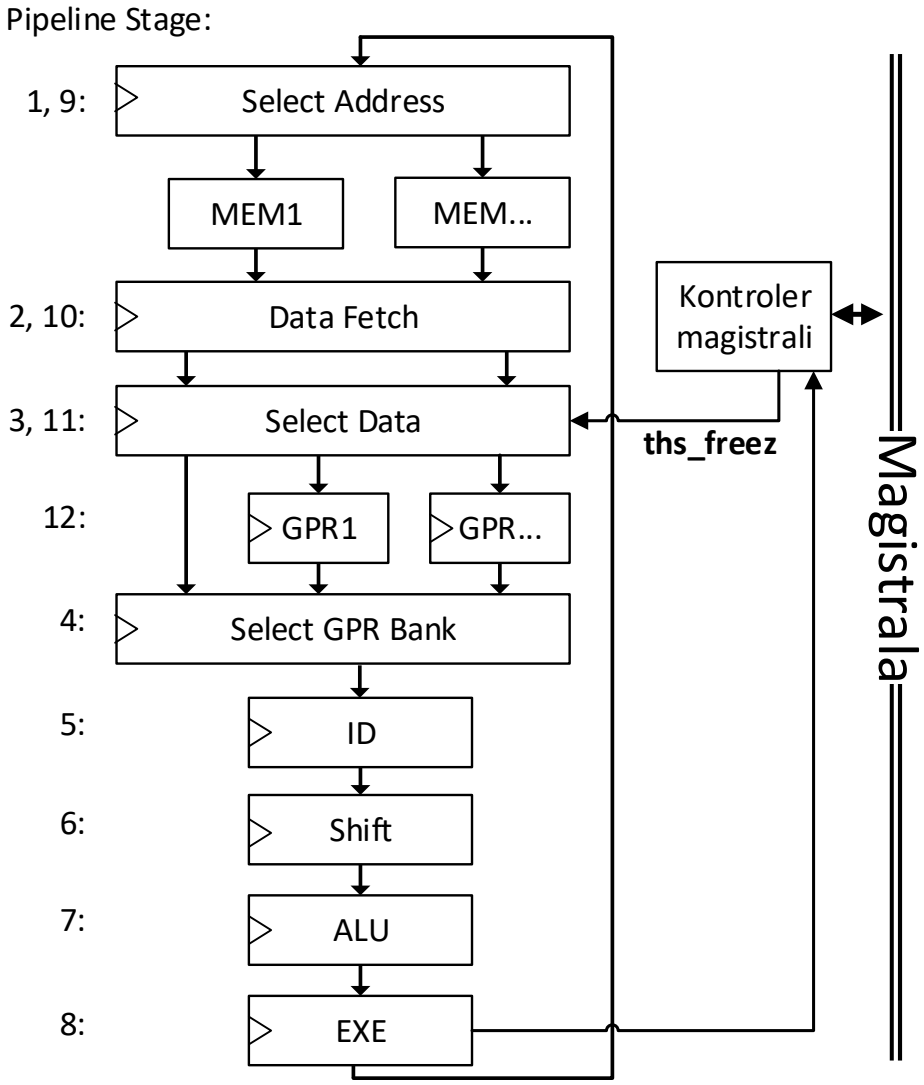
Gdy któryś z wątków wykonuje instrukcje wymiany danych z innym wątkiem, następuje zatrzymanie jego przetwarzania, aż do otrzymania odpowiedzi zwrotnej z magistrali wymiany danych. Jest to spowodowane hazardami wynikającymi z przetwarzania następnych instrukcji. Następny rozkaz takiego wątku może przetwarzać daną, która nie została jeszcze pobrana z pamięci innego wątku. Drugą sytuacją wymuszającą wstrzymywanie pracy wątku wysyłającego zapytanie, jest zdarzenie, w którym po wysłaniu zapytania następną instrukcją jest również wymiana danych z innym wątkiem. Kolejne zapytania nie mogą być odkładane na stos w nieskończoność. Z powodu powyższych sytuacji oraz przewidywalności czasowej zdecydowano się zatrzymywać pracę każdego wątku na czas wykonania wymiany danych. Czas ten jest przewidywalny i zostanie omówiony w kolejnych podrozdziałach (5.3.3.2 oraz 5.3.4.4).

Wstrzymywanie pracy wątku odbywa się poprzez odpowiednie wysterowanie sygnału *ths_freez* **(**Rys. 30**)**. Sygnał ten występuje w każdym z rdzeni pracujących w systemie i posiada tyle bitów, ile wątków przetwarza dany rdzeń (kod ONEHOT). Każdy z bitów odpowiada jednemu z wątków. Gdy bit ma stan wysoki, praca wątku do niego przyporządkowanego jest wstrzymana.

Sygnał *ths_freez* jest sterowany poprzez kontroler magistrali. Gdy faza EXE zgłasza do niego informacje o chęci nadania zapytania, ustawia on odpowiedni bit tego sygnału w stan wysoki. Gdy proces wymiany danych się zakończy, magistrala odeśle odpowiedź lub potwierdzenie wykonania operacji. Stan odpowiedniego bitu *ths_freez* wraca do poziomu niskiego.

Sygnał zatrzymujący pracę wątków (*ths_freez*) jest podłączony do fazy SD. W fazie tej są wybierane odpowiednie komórki pamięci z pobranych wcześniej z fazy DF. W trzeciej fazie potoku są to instrukcje procesora. Gdy praca któregoś z wątków jest wstrzymana, w fazie SD instrukcja tego wątku jest podmieniana na dodatkowo zaimplementowaną instrukcję WAIT. Instrukcja ta jest bardzo podobna do standardowej instrukcji NOP, również nie wykonuje żadnej operacji. Różnica polega na tym, że nie inkrementuje ona wartości rejestru PC. Procesor wykonujący instrukcje takiego wątku, blokuje się w nieskończonej pętli nie wykonując żadnej operacji. W momencie, którym sygnał *ths_freez* osiągnie wartość 0, instrukcja w fazie SD nie jest podmieniana na WAIT,

ale jest instrukcją, która miała być wykonana po rozkazie wymiany danych. W ten sposób procesor wraca do swojej normlanej pracy.

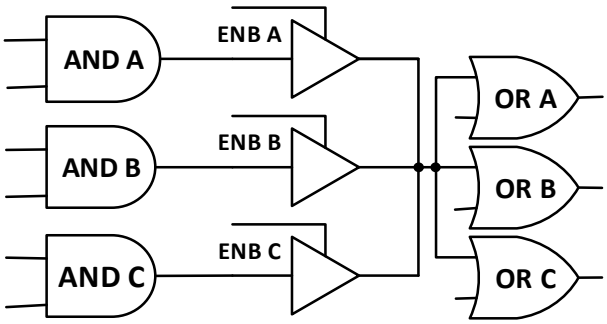


Rys. 30. Wstrzymywanie pracy wątku

5.3.3 Autorska implementacja magistrali dwukierunkowej

Pierwszy protokół wymiany danych zakładał użycie minimalnej liczby sygnałów magistrali. Wykorzystano więc sygnały dwukierunkowe, które do przesyłania danych w odpowiednim kierunku wykorzystują bramki trójstanowe. Bramka taka (oznaczana symbolem trójkąta z dwoma wejściami, Rys. 31), na swoim wyjściu oprócz dwóch standardowych stanów logicznych 0 i 1, posiada trzeci stan wysokiej impedancji

oznaczanej jako Z. Stan ten jest sterowany osobnym sygnałem ENB, a 1 oznacza stan wysokiej impedancji. Użycie takich buforów pozwala odciąć od magistrali wszystkie układy, poza tym, który w danej chwili korzysta z magistrali. Mechanizm ten pozwala również zapobiec efektowi „iloczynu na drucie". Przykład takiej transmisji przedstawiono na Rys. 31. Po założeniu, że bramka AND A oraz OR A znajduje się w rdzeniu A, bramka AND B oraz OR B w rdzeniu B itd. Można wysłać informacje z rdzenia A do rdzenia B poprzez załączenie sygnałów ENB B oraz ENB C. Tymi samymi sygnałami można również przesłać informacje z rdzenia B do rdzenia A załączając sygnały ENB A oraz ENB C. Używając takiej metody, można więc przesyłać informacje w dwóch kierunkach za pomocą jednego sygnału.



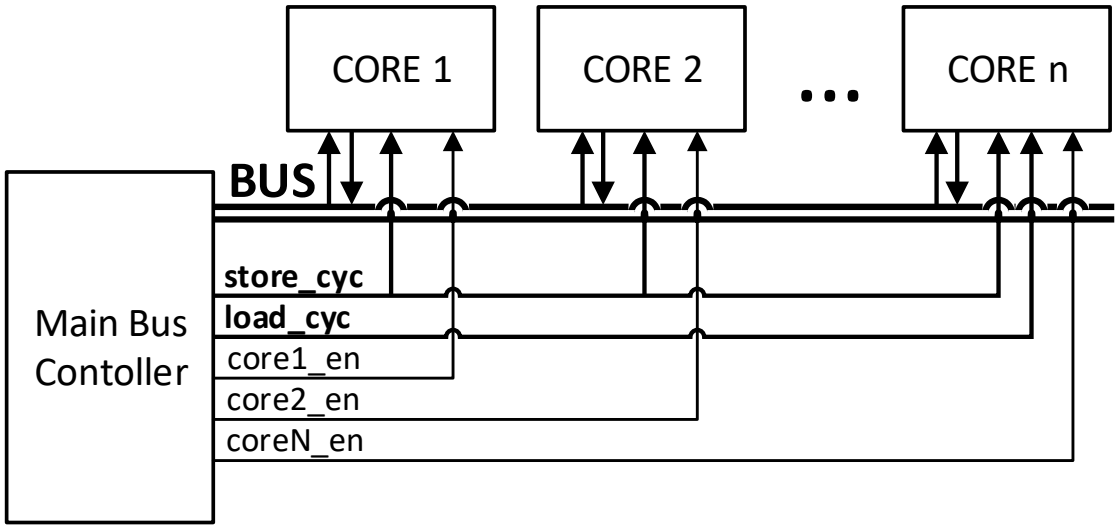
Rys. 31. Bramki trójstanowe

Projektowana magistrala powinna być przewidywalna czasowo. Każda operacja zapisu lub odczytu musi być wykonana w czasie przewidzianym wcześniej, przed uruchomieniem zadania. W tej wersji magistrali ograniczono się do przewidzenia, możliwie niskiego, maksymalnego czasu wykonania operacji wymiany danych. Wszystkie obliczenia przewidujące długość tego czasu powinny być stosunkowo proste i możliwe do uzyskania w szybki sposób.

5.3.3.1 Architektura magistrali dwukierunkowej

Do zapewnienia przewidywalności czasowej, każdy z wątków powinien mieć ustalony maksymalny czas, po którym uzyska dostęp do magistrali. Najprostszą metodą jest kolejne wykonywanie zapytań wymiany danych wszystkich wątków pracujących w systemie. W tym protokole zdecydowano się jednak na odpytywanie wątków blokowo, dzieląc je według rdzeni. Podejście takie pozwoli zaoszczędzić czas, ponieważ w czasie,

gdy pozostałe wątki wykonują zapytania, poprzednie z nich mogą być już przetwarzane. Wszystkie dane wysyłane są tymi samymi dwukierunkowymi sygnałami, więc informacje o zapytaniach czy danych zwrotnych muszą być gdzieś przechowywane. Blok zapytań nie powinien być więc zbyt długi. Z tego powodu zdecydowano się nie odpytywać wszystkich wątków naraz, a podzielić je na grupy zgodne z hierarchią rdzeni. Dzięki temu pamięci przechowywujące te dane nie muszą być duże.



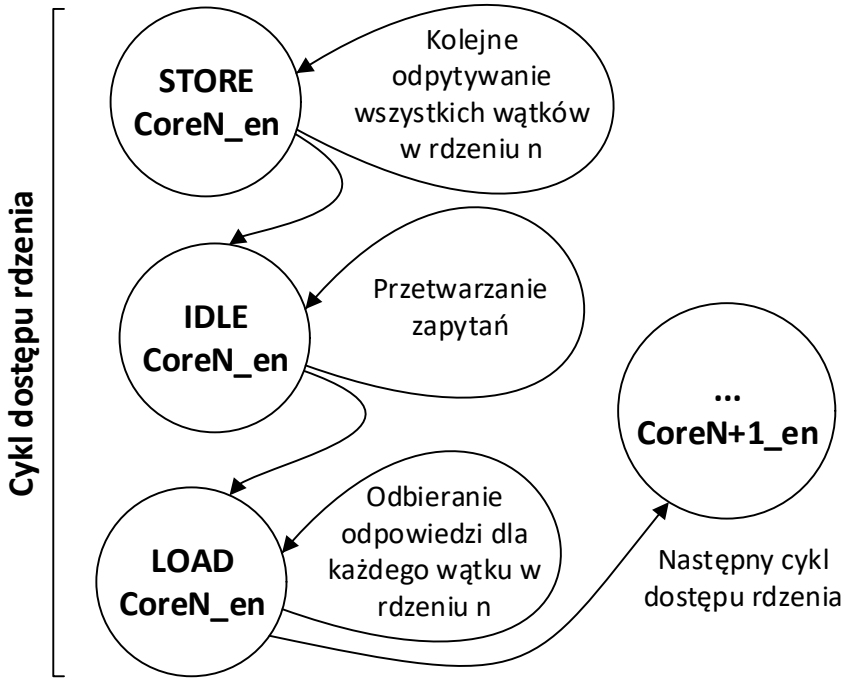
Rys. 32. Magistrala dwukierunkowa

Na rysunku Rys. 32 przedstawiono poglądowy schemat magistrali. Sygnały podzielono na sygnały dwukierunkowe którymi wysyłane są dane oraz zapytania (na Rys. 32 oznaczono je jako BUS) oraz jednokierunkowe sygnały sterujące. Do zarządzania magistralą zaprojektowano specjalistyczny kontroler nazwany **Main Bus Controller (MBC).** MBC generuje sygnały sterujące: „*store_cyc”* oraz „*load_cyc”.* Wskazują one w jakim cyklu jest magistrala. Kolejnymi sygnałami sterującymi są „*coreN_en”* (gdzie N – to numer rdzenia). Ich liczba zależy od liczby rdzeni pracujących w systemie. Pokazują one, który z rdzeni w systemie ma prawo do nadawania na magistrali w danym przedziale czasowym. Magistrala BUS składa się z następujących dwukierunkowych sygnałów:

- *mag_thadr* – adres wątku adresata zapytania
- *mag_memadr* – lokalny adres komórki pamięci zapytania
- *mag_data* – dane odczytane z komórki podczas zapytania odczytu

- *mag_store* – sygnalizator zapytania zapisu
- *mag_load* – sygnalizator zapytania odczytu

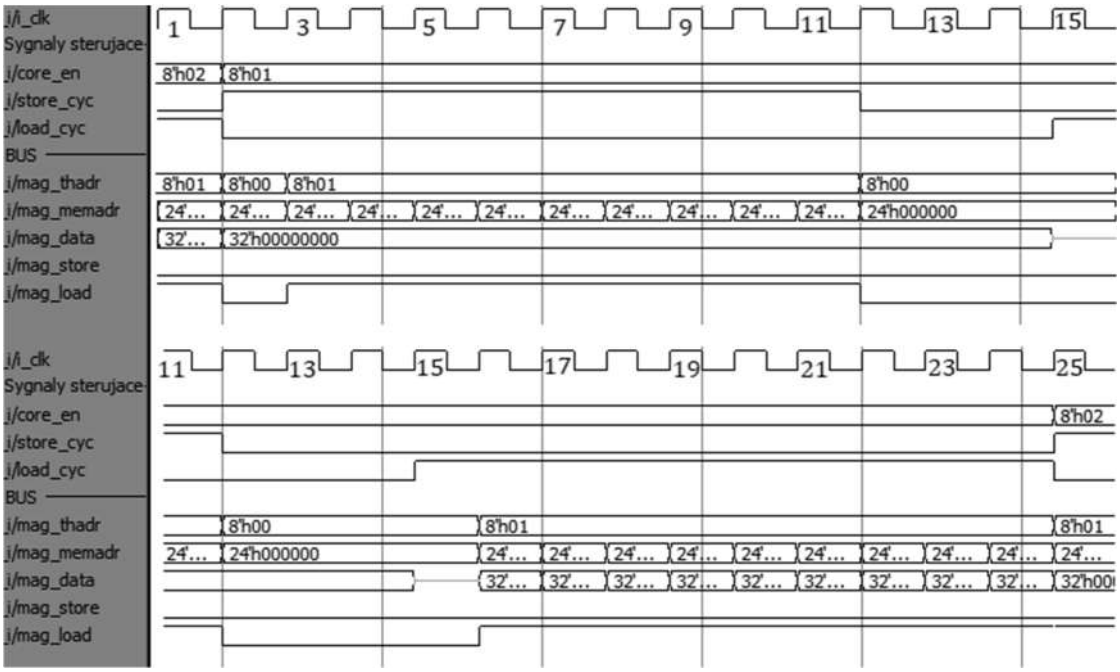
Do rozróżnienia rodzaju informacji i wprowadzenia organizacji, na magistrali wprowadzono 3 stany, w których może się ona znaleźć. Te cykle to **STORE (SC), IDLE (IC)** oraz **LOAD (LC)**. Kolejność ich przejść przedstawiono na Rys. 33. Odpowiadają one sygnałom sterującym z Rys. 32. Jeżeli oba sygnały „*store_cyc”* oraz „*load_cyc”* mają wartość logiczną 0, oznacza to, że aktywny jest stan IC.



Rys. 33. Cykl dostępu rdzenia

Przykładową transmisję danych przedstawiono na Rys. 34. Rysunek ten pochodzi z symulacji zaimplementowanego rozwiązania w programie QuestaSim. W systemie tym pracują dwa dziesięciowątkowe rdzenie. W pierwszym z nich znajdują się wątki od 1 do 10, w kolejnym od 11 do 20. Dla ułatwienia omawiania transmisji, takty zegara zostały ponumerowane liczbami od 1 do 25. Cykl dostępu rdzenia pierwszego zaczyna się w takcie 2 i trwa do taktu 24. MBC ustawia sygnały „*core_en”*, „*store_cyc”* oraz „*load_cyc”*, zaczynając cykl dostępu rdzenia 1 oraz fazę magistrali SC. W takcie 2 cyklu zegara nie następuje żadna transmisja zapytania. Oznacza to, że wątek 1 nie wykonuje

w tym momencie żadnej instrukcji komunikacji z innymi wątkami. Brak transmisji można rozpoznać po tym, że na sygnale magistrali „*mag_thadr'* występuje adres zastrzeżony (0x00) oraz oba sygnały „*mag_store"* oraz „*mag_load"* mają stan niski. Od taktu 3 do taktu 11 kolejne wątki znajdujące się w rdzeniu 1 (wątek 2, 3, 4, 5, 6, 7, 8, 9 oraz 10) wysyłają zapytania odczytu danych z pamięci wątku 1. Dokładne 24-bitowe adresy komórek, z których mają być pobrane dane, zapisane są w sygnale „*mag_memadr"*.



Rys. 34. Przebiegi sygnałów magistrali dwukierunkowej

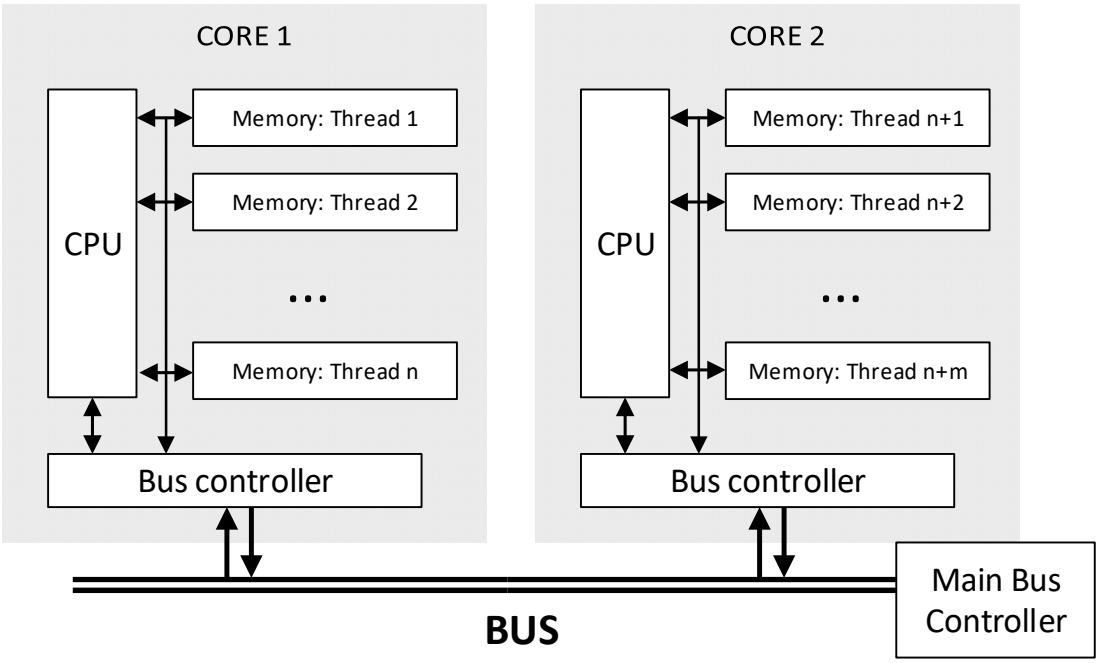
W takcie 12 zaczyna się cykl IC. Jego długość jest stała i zależna od konfiguracji systemu (w tej konfiguracji wynosi ona 2 cykle zegara). Jest on niezbędny do przetworzenia zapytań oraz zapobiegnięciu sytuacji, w której w czasie zapytania o pamięć, ta pamięć jest zajęta przez rdzeń systemu.

Cykl LC trwa między taktem 15 a 24. Podczas niego wysyłane są dane pobrane z komórek, o które wysyłane zostały zapytania w fazie SC. W takcie 15 występuje stan wysokiej impedancji oznaczony kolorem niebieskim. Pojawia się on, ponieważ w takcie 2 nie zostało wysłane żadne zapytanie, więc żaden z rdzeni na nie odpowiada. Jest to sytuacja normalna. W kolejnych taktach zegarowych (16-24) wysyłane są pobrane dane

na zapytania z taktów 3-11. Globalny adres komórki pamięci pobieranej danej można uzyskać po połączeniu sygnałów „*mag_thadr”* oraz „*mag_memadr”*. Wartość pobieranej danej przesyłana jest sygnałem „*mag_data”*. Od 25 taktu zegara zaczyna się cykl dostępu drugiego rdzenia.

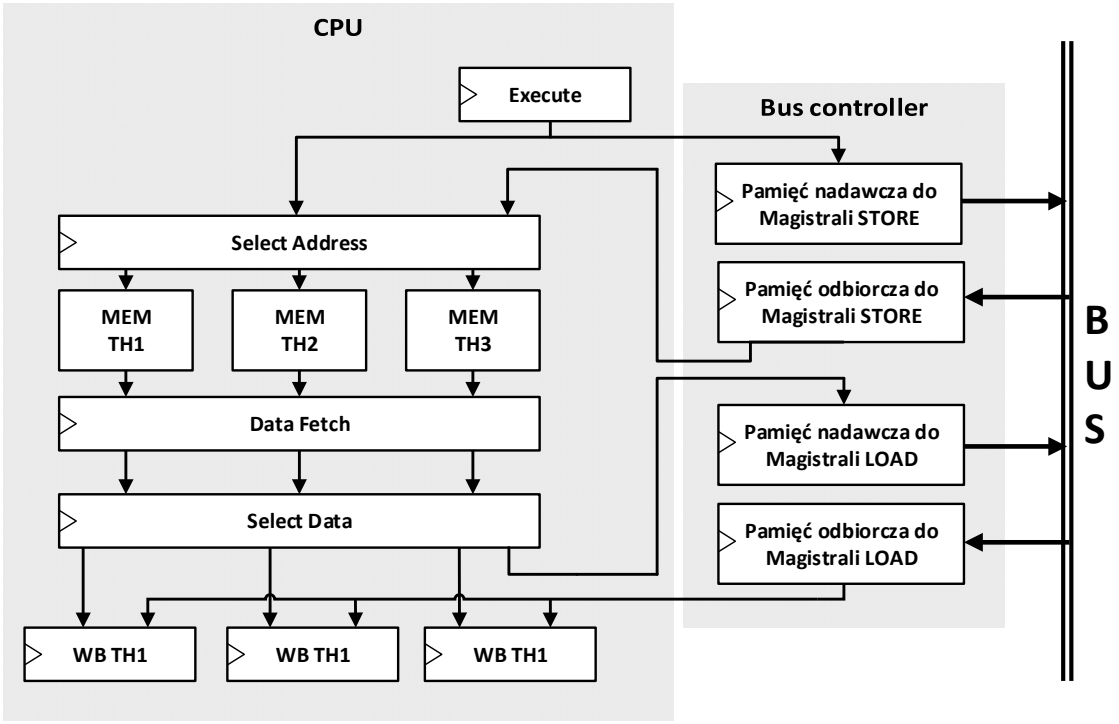
Wyżej omawiana sytuacja dotyczy zapytania pobrania danej z pamięci innego wątku. Instrukcja zapisu danej do innej pamięci jest bardzo podobna. Taka instrukcja jest oznaczana na magistrali wysokim stanem sygnału „*mag_store”*. Tego typu operacja nie wymaga informacji zwrotnej z pobranymi danymi. Dlatego w cyklu LC wysyłane są identyczne dane jak podczas cyklu SC jako potwierdzenie zapisanych danych.

W każdym z rdzeni znajduje się **Bus Controller (BC)**, pokazany na Rys. 35. Do niego bezpośrednio są podłączone sygnały magistrali oraz pamięci wątków. BC na bieżąco komunikuje się z CPU, aby pozyskać informacje, które z pamięci będą używane w kolejnym cyklu zegara. Pamięci, które nie będą używane, mogą być użyte do wymiany danych z magistralą. Operacja taka przebiega bez użycia CPU. BC ma swobodny dostęp do pamięci i może jej używać niezależnie od CPU, gdy jest ona wolna.



Rys. 35. Układy sterujące magistralą

Transmisja danych rozpoczyna się w fazie potoku Execute (EXE). W tej fazie podejmowana jest decyzja, czy przetwarzany rozkaz jest rozkazem wymiany danych z pamięcią oraz czy jest to pamięć wątku wysyłającego zapytanie (Rys. 36). Jeżeli ten rozkaz jest rozkazem komunikacji z innym wątkiem, takie zapytanie zapisywane jest w pamięci nadawczej do magistrali STORE. Pamięć ta musi posiadać tyle komórek pamięci, ile wątków jest przetwarzanych przez rdzeń, w którym się znajduje. Po zapisaniu zapytania do pamięci przetwarzanie wątku wysyłającego zapytanie jest zatrzymywane, aż do otrzymania odpowiedzi. Zapytanie przetrzymywane jest w pamięci, aż do momentu, w którym MBC przejdzie w fazę SC aktualnego rdzenia. Każdy z taktów zegara fazy SC jest przeznaczony do wysłania zapytania przez jeden wątek rdzenia. Przedziały te są przydzielane według kolejności, w której wątki zostały przydzielone do rdzenia. Gdy przyjdzie odpowiedni takt zegara, zapytanie jest wysyłane na magistralę.



Rys. 36. Kontroler magistrali

Zapytanie jest następnie zapisywane w pamięci odbiorczej magistrali STORE rdzenia, który jest adresatem zapytania. Jeżeli pamięć jest zajęta przez CPU, zapytanie oczekuje w pamięci, aż do jej zwolnienia. Co nie trwa dłużej niż jeden takt zegara (rozdział 4.5 Pamięć systemu oraz komunikacja wątków). Po przetworzeniu zapytania

przez fazy potoku SA, DF oraz SD odpowiedź zawierająca dane zapisywana jest w pamięci nadawczej LOAD i oczekuje tam, aż magistrala przejdzie w stan LC.

Zapytanie musi mieć odpowiednią ilość czasu na jego wykonanie, czyli pobranie odpowiednich danych z pamięci. Z tego powodu została wprowadzona faza magistrali IC. Do obliczenia czasu trwania tego stanu, użyto najgorszego przypadku, czyli rdzenia, w którym jest 9 wątków, najmniejszej możliwej liczby do poprawnej pracy systemu. W takim scenariuszu zakładając brak stanu IDLE, już w 10 takcie po pierwszym zapytaniu magistrala przechodzi w fazę LC. Oznacza to, że po 10 taktach zegara, na magistrali muszą pojawić się już dane pobrane z pamięci. Najbardziej niekorzystną sytuacją jest taka, w której wszystkie zapytania pytają o jedną pamięć. W przykładzie (Tab. 2) założono 9 zapytań (ZcX, X - oznacza takt zegara, w którym wpłynęło zapytanie) o dane z pamięci wątku 1.

Tab. 2. Czas trwania cyklu dostępu do magistrali

Takt zegara	**Faza**	**Pamięci zajęte przez CPU**	**Wykonane zapytania**	**Zapytania w kolejce**	**Uwagi**
1	SC	P1, P5	Brak	Zc1	Zajęta pamięć P1
2	SC	P2, P6	Zc1	Zc2	
3	SC	P3, P7	Zc2	Zc3	
4	SC	P4, P8	Zc3	Zc4	
5	SC	P5, P9	Zc4	Zc5	
6	SC	P6, P1	Brak	Zc5, Zc6	Zajęta pamięć P1
7	SC	P7, P2	Zc5	Zc6, Zc7	
8	SC	P8, P3	Zc6	Zc7, Zc8	
9	SC	P9, P4	Zc7	Zc8, Zc9	Koniec fazy SC
10		P1, P5	Brak	Zc8, Zc9	Zajęta pamięć P1
11		P2, P6	Zc9	Zc8	
12		P3, P7	Zc8	Brak	

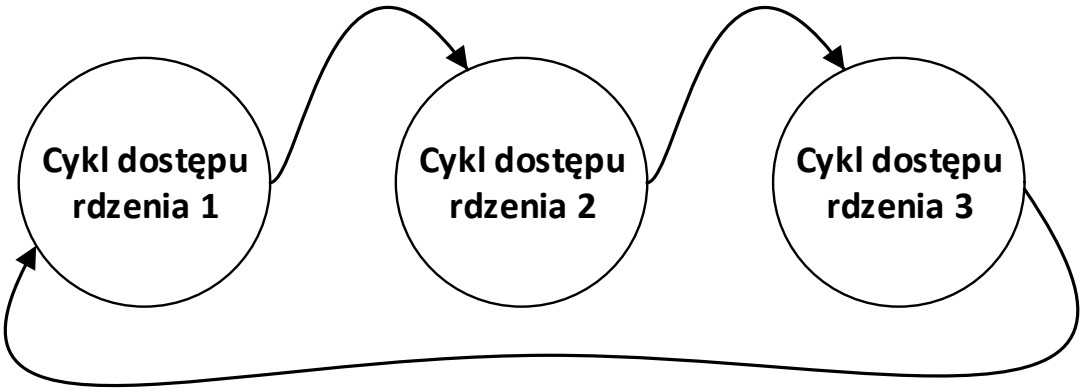
Z przytoczonego (Tab. 2) przykładu wynika, że minimalny czas dostępu do pamięci w najgorszym przypadku wynosi 12 taktów zegara. Czas wykonania zapytania to 5 cykli zegara, składa się na to: zapisanie zapytania w pamięci odbiorczej magistrali STORE, przetworzenie zapytania przez fazy potoku SA, DF, SD oraz zapisanie wyniku w pamięci nadawczej magistrali LOAD. Nadanie danych w fazie LC trwa dodatkowe 9 cykli zegara (dla 9 wątkowego rdzenia). Po zsumowaniu całości, cykl dostępu magistrali powinien wynosić 25 cykli zegara. Cykl LC można jednak zacząć wcześniej, już w 12 takcie zegara, ponieważ LC zaczyna wysyłanie odpowiedzi zapytań, które zostały wysłane najwcześniej. W 12 takcie zostały one przetworzone i znajdują się już w pamięci nadawczej magistrali LOAD. Po takim usprawnieniu cały cykl może trwać 20 cykli zegara: 9 cykli SC, 2 cykle IC oraz 9 cykli LC. Po takim rozważaniu przyjęto, że system będzie działał w każdej konfiguracji przy czasie trwania stanu IC wynoszącym 2 cykle zegara.

5.3.3.2 Przewidywalność czasowa wymiany danych

Protokół wymiany danych między wątkami był projektowany w ten sposób, aby zapewnić jak największą przewidywalność czasową wykonania operacji wymiany danych. Długość wykonania takiej operacji oznaczono jako *M_durI* (6) i wyrażane jest ono w liczbie taktów zegara systemu. W ten sposób można uniezależnić czas wykonania rozkazu wymiany danych od częstotliwości pracy systemu.

Maksymalny czas wykonania takiego rozkazu nastąpi, gdy zapytanie o wymianę danych przetwarzane będzie w momencie, w którym takt temu wątek wysyłający zapytanie znajdował się w fazie SC własnego rdzenia i miał prawo takie zapytanie wysłać. W takim wypadku rozkaz ten musi czekać cały cykl wymiany danych magistrali (Rys. 37 i Rys. 33), aby takie zapytanie zostało wysłane.

Każdy z cykli dostępu do rdzenia składa się z faz SC, IC oraz LC. Czas trwania faz SC oraz LC jest zależny od liczby wątków w danym rdzeniu i jest równy ich liczbie (Rys. 33). Po zsumowaniu czasu trwania faz SC oraz LC całego systemu jest on równy dwukrotności liczby wszystkich wątków pracujących w systemie. Dla każdego wątku występuje jedna faza SC oraz jedna faza LC trwająca dokładnie jeden takt zegarowy. Liczba faz IC jest równa liczbie rdzeni w systemie. W standardowych konfiguracjach rdzenia trwa ona zawsze 2 takty zegara



Rys. 37. Cykl wymiany danych magistrali

Po podsumowaniu powyższych rozważań, maksymalny czas dostępu do pamięci innego wątku jest zgodny ze wzorem (6). Czas wykonania instrukcji wymiany danych mieści się zawsze w przedziale od 1 cyklu zegara (w sytuacji, gdy zapytanie idealnie trafi w swój cykl SC magistrali) do liczby cykli zegara wynoszącej *M_durI* (6).

$$M_{durI} = \sum_{i=0}^{CN-1} Idle_{dur\ i} + 2 \bullet ThN \qquad (6)$$

gdzie:

CN – Liczba rdzeni systemu

$Idle_{dur\ i}$ – Czas trwania fazy IC dla rdzenia i

ThN - Liczba wątków w systemie

5.3.4 Autorska implementacja magistrali jednokierunkowej

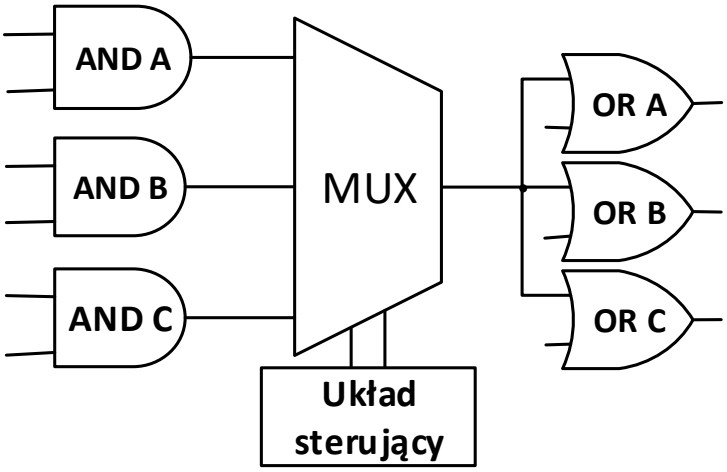
Magistrala jednokierunkowa powstała jako rozwinięcie wersji dwukierunkowej. Po licznych eksperymentach oraz doświadczeniu zdobytym podczas implementacji magistrali wykorzystującej stany wysokiej impedancji, zdecydowano się na zaprojektowanie nowej wersji wykorzystującej inną koncepcję.

Pierwszą zmianą było zrezygnowanie z idei stanów wysokiej impedancji. Po badaniach z nimi związanych okazało się, że wprowadzają one duże pojemności złączowe, co ograniczało maksymalne taktowanie systemu. Drugim problemem było to, że obecne układy FPGA nie posiadają już wbudowanych buforów trójstanowych (posiadają je tylko na wejściach/wyjściach układów), właśnie z powodu dużej wprowadzanej pojemności. Podczas syntezy takie bramki zostają zamienione na sygnały sterowane multiplekserami. Zdecydowano się więc na implementację właśnie w ten sposób, tj. wykorzystując multipleksery.

Wadami takiego rozwiązania jest to, że multipleksery są dużo bardziej złożone niż bufory trójstanowe, przez co zużywają więcej zasobów. Dodatkowo multiplekser przełączający sygnały, które mają być wysłane na magistralę musi znajdować się poza rdzeniami (podłączone są do niego wszystkie rdzenie). Oznacza to, że każdy z rdzeni musi posiadać swoją osobną linię danych. Skutkuje to kilkukrotnie (liczba ta rośnie wraz ze zwiększaniem się liczby rdzeni) większą liczbą sygnałów danych. Wielkość multipleksera rośnie również liniowo, z liczbą rdzeni pracujących w systemie.

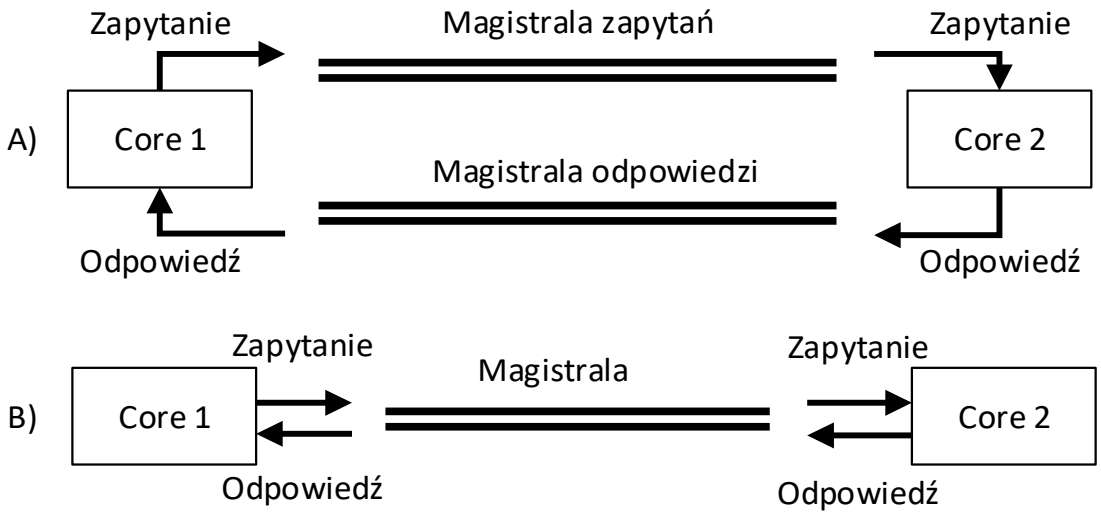
Sterowanie magistralą poprzez multiplekser pokazano na Rys. 38. Zakładając, że bramka AND A oraz OR A, znajdują się w rdzeniu A oraz AND B i OR B w rdzeniu B itd., można zapewnić komunikację każdego rdzenia z każdym poprzez odpowiednie wysterowanie multipleksera. Sterowanie to jest bardzo podobne do sterowania w sytuacji używania buforów trójstanowych (Rys. 31). W obydwu przypadkach wykorzystywany jest kod ONEHOT.

Drugą znacząca zmianą była rezygnacja z wysyłania zapytań oraz odpowiedzi tymi samymi sygnałami magistrali. Modyfikacja ta zwiększyła liczbę sygnałów magistrali dwukrotnie, w zamian zwiększając teoretycznie dwukrotnie przepustowość magistrali. Struktura magistrali stała się też znacznie prostsza, ponieważ nie ma już potrzeby stosowania różnych cykli lub stanów magistrali takich jak SC, LC czy IC.



Rys. 38 Sterowanie magistralą poprzez multiplekser

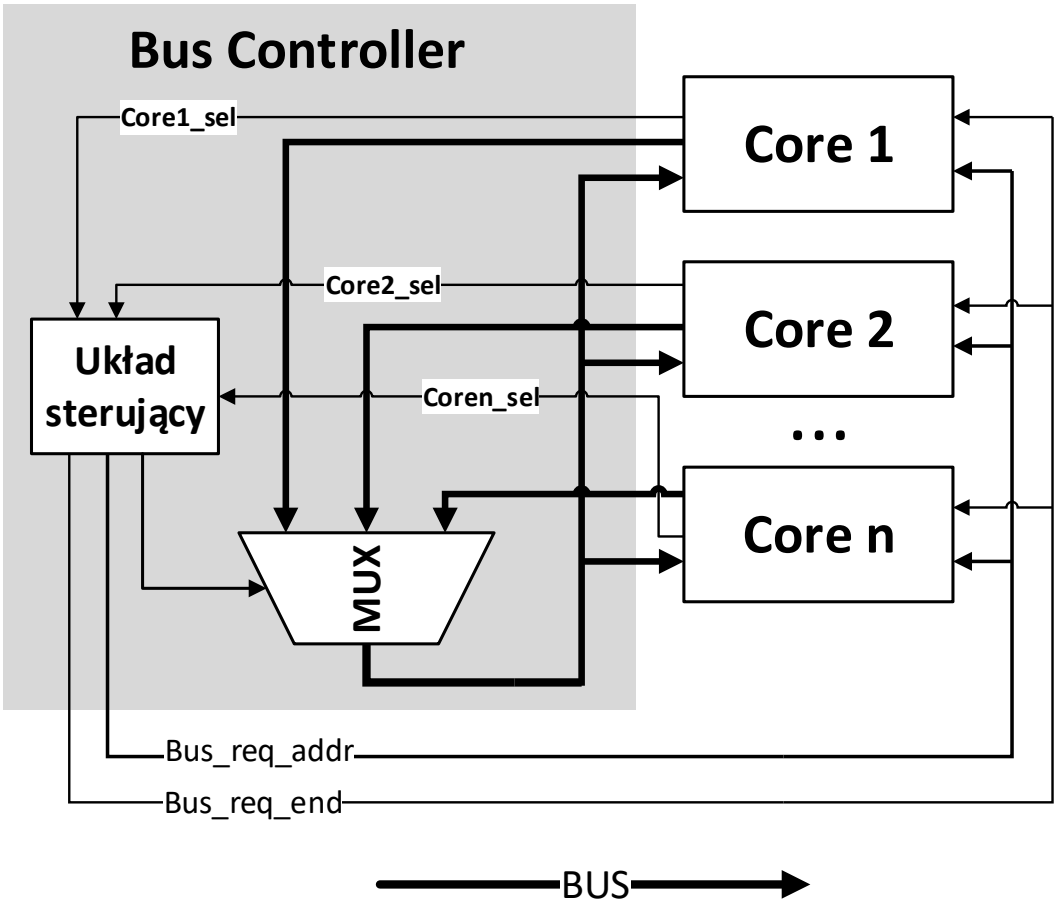
Omawiania magistrala (Rys. 39 A) składa się z dwóch osobnych magistral. Jedną z nich przesyłane są tylko zapytania o dane komórki pamięci, a drugą z nich przesyłane są odpowiedzi na poprzednie zapytania. W tym samym takcie zegara mogą być więc wysyłane zapytania jednego wątku i odpowiedzi dla innego. Z tego usprawnienia pochodzi wzrost transferu danych magistrali. W poprzednim rozwiązaniu wszystkie informacje wysyłane były tymi samymi sygnałami (Rys. 39 B).



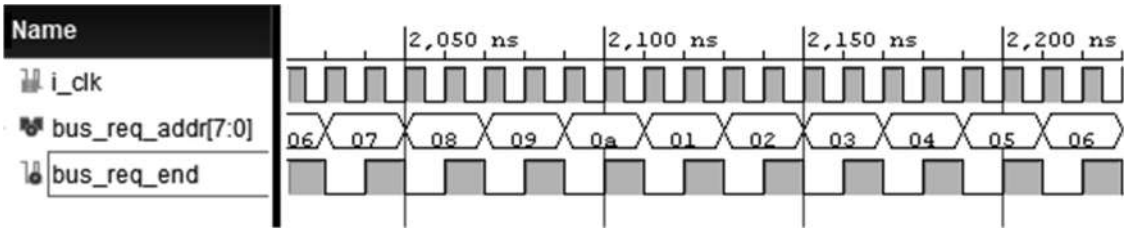
Rys. 39. Porównanie kierunku przesyłania danych w magistralach

5.3.4.1 Architektura magistrali jednokierunkowej

Głównym układem sterującym magistralą oraz zawierającym multiplekser przełączający dane, jest pokazany na Rys. 40, **Bus Controller (BC)**. Podobnie jak w poprzedniej wersji magistrali generuje on sygnały sterujące. Tymi sygnałami sterującymi są: „*bus_req_addr”* oraz *„bus_req_end”*. Zostały one pokazane na Rys. 41. Sygnał „*bus_req_addr”* podaje adres wątku, który w danej chwili może wysłać zapytanie o dane z pamięci innego wątku. W rozdziale 4.5 Pamięć systemu oraz komunikacja wątków ustalono, że każde zapytanie potrzebuje do dwóch taktów zegara, aby otrzymać dostęp do pamięci. Dlatego na każde zapytanie przewidziano dwa takty zegara. Sygnał „*bus_req_end”* sygnalizuje, że w następnym takcie zegara, prawo do nadawania danych na magistrali się zmieni. BC został napisany w taki sposób, aby długość okna nadawania można było zmienić za pomocą parametru. Dzięki temu w prosty sposób można badać różne konfiguracje.



Rys. 40. Sygnały sterujące magistralą



Rys. 41. Sygnały sterujące magistralą jednokierunkową

Dodatkowymi sygnałami sterującymi (Rys. 40) są sygnały „*CoreN_sel”* oraz „*CoreN_sel_rep”*. Liczba tych sygnałów jest równa liczbie rdzeni w systemie. Tylko pierwszy z nich został zaznaczony na Rys. 40, ponieważ są one podłączone w identyczny sposób. Sygnał „*CoreN_sel”* dotyczy magistrali zapytań, a „*CoreN_sel_rep”* magistrali odpowiedzi. Sygnały te są odbierane przez BC i pochodzą z każdego rdzenia w systemie. Uaktywniają się one, gdy któryś z rdzeni nadaje w systemie w którejś z magistral. Pozwalają one przestawić multiplekser do odpowiedniej pozycji oraz sprawdzić czy tylko jeden rdzeń nadaje w danym takcie zegara.

Układy z Rys. 40 są powtórzone dwukrotnie, raz dla magistrali zapytań i drugi raz dla magistrali odpowiedzi. Dla rozróżnienia sygnałów, sygnału pochodzące z magistrali zapytań posiadają przedrostek *„req”,* natomiast sygnały pochodzące z magistrali odpowiedzi *„rep”*. Sygnały magistrali pochodzące od rdzeni systemu oznaczone są poprzez przedrostek *„ibus”,* natomiast te pochodzące od BC poprzez *„obus”*. Przedrostek *„mux”* oznacza, że są to sygnały wszystkich rdzeni połączone w jednym wektorze.

Na Rys. 42 przedstawiono instancje modułu BC. Dzięki parametryzacji modułu, przed syntezą systemu można ustawić takie parametry jak: liczba wątków biorących udział w wymianie danych (REQ_TH_NUM), liczba taktów zapytania (REQ_CYCLES), liczbę rdzeni (CORE_NUM) oraz adresy i kolejność wątków, w której mają one otrzymać pozwolenie na wysyłanie zapytań przez magistralę *(„i_req_th_adress”*). Ostatni z atrybutów zaimplementowano jako sygnał, aby umożliwić jego zmianę w trakcie działania systemu.

```
 bus_ctrl #
(
    .REQ_TH_NUM        (10),
    .REQ_TH_NUM_WIDTH  (4),
    .REQ_CYCLES        (2),
    .REQ_CYC_WIDTH     (1),
    .MEM_ADR_WIDTH     (MEM_ADR_WIDTH),
    .TH_ADR_WIDTH      (TH_ADR_WIDTH),
    .CORE_NUM          (CORE_NUM)
)
 bus_ctrl_i
(
    .i_clk                      (i_clk),
    .i_rst                      (i_rst),

    // Sygnaly sterujące
    .i_req_th_address           ({8'd10,8'd9,8'd8,8'd7,8'd6,
                                  8'd5,8'd4,8'd3,8'd2,8'd1}),
    .o_bus_req_addr             (bus_req_addr),
    .o_bus_req_end              (bus_req_end),

    // Magistrala zapytan
    .i_core_sel                 (core_sel               ),
    .ibus_req_mux_req_thadr     (ibus_req_mux_req_thadr ),
    .ibus_req_mux_req_end       (ibus_req_mux_req_end   ),
    .ibus_req_mux_thadr         (ibus_req_mux_thadr     ),
    .ibus_req_mux_memadr        (ibus_req_mux_memadr    ),
    .ibus_req_mux_data          (ibus_req_mux_data      ),
    .ibus_req_mux_store         (ibus_req_mux_store     ),
    .ibus_req_mux_load          (ibus_req_mux_load      ),
    .ibus_req_mux_rd_sel        (ibus_req_mux_rd_sel    ),

    .obus_req_req_thadr         (obus_req_req_thadr     ),
    .obus_req_req_end           (obus_req_req_end       ),
    .obus_req_thadr             (obus_req_thadr         ),
    .obus_req_memadr            (obus_req_memadr        ),
    .obus_req_data              (obus_req_data          ),
    .obus_req_store             (obus_req_store         ),
    .obus_req_load              (obus_req_load          ),
    .obus_req_rd_sel            (obus_req_rd_sel        ),

    // Magistrala odpowiedzi
    .i_core_sel_rep             (core_sel_rep),
    .ibus_rep_mux_req_thadr     (ibus_rep_mux_req_thadr ),
    .ibus_rep_mux_data          (ibus_rep_mux_data      ),
    .ibus_rep_mux_store         (ibus_rep_mux_store     ),
    .ibus_rep_mux_load          (ibus_rep_mux_load      ),
    .ibus_rep_mux_rd_sel        (ibus_rep_mux_rd_sel    ),

    .obus_rep_req_thadr         (obus_rep_req_thadr ),
    .obus_rep_data              (obus_rep_data      ),
    .obus_rep_store             (obus_rep_store     ),
    .obus_rep_load              (obus_rep_load      ),
    .obus_rep_rd_sel            (obus_rep_rd_sel    )

);
```

Rys. 42. Przykład konkretnego modułu Bus Controller'a

Do BC podłączone są wszystkie sygnały magistrali. W porównaniu do poprzedniej wersji magistrali sygnałów tych jest znacznie więcej. W poprzedniej wersji dodatkowe informacje trzeba było zapisywać do nadplanowych pamięci. Taką informacją jest np. numer rejestru, do którego mają być zapisane pobrane dane (sygnał „*ibus_req_mux_rd_sel”*). Wiązało się to też z opracowaniem dodatkowych układów, które w odpowiednim momencie odczytywały te dane. Takie mechanizmy powodowały dodatkowe i niepotrzebne komplikowanie magistrali. Dlatego po doświadczeniach z poprzedniej implementacji zdecydowano się wszystkie informacje wysyłać magistralą. Takie rozwiązanie powoduje również dużo prostsze śledzenie działania magistrali, ponieważ wszystkie informacje o transferze danych znajdują się w jednym miejscu tj. w magistrali.

W skład magistrali zapytań wchodzą następujące sygnały:

- *req_req_thadr* – adres wątku wysyłającego zapytanie
- *req_req_end* – stan sygnału sterującego „*bus_req_end”* podczas wystąpienia zapytania
- *req_thadr* – adres wątku adresata zapytania
- *req_memadr* – adres lokalnej komórki pamięci występującej w zapytaniu
- *req_data* – dane do zapisania podczas wykonywania instrukcji STORE
- *req_store* – sygnalizator instrukcji STORE
- *req_load* – sygnalizator instrukcji LOAD
- *req_rd_sel* – numer rejestru, do którego ma być zapisana dana odczytana instrukcją LOAD

Magistrala odpowiedzi nie potrzebuje dokładnej kopii sygnałów magistrali zapytań. Sygnały „*req_thadr”* oraz „*req_memadr”* są już niepotrzebne, ponieważ podczas wysyłania odpowiedzi dane zostały już odczytane/zapisane z komórek o tym adresie. Sygnał „*req_req_end”* również jest już niepotrzebny, ponieważ używany jest on do ustalenia jak długo dana pamięć może być niedostępna. Magistrala odpowiedzi składa się więc z 5 sygnałów:

- *rep_req_thadr* – adres wątku wysyłającego zapytanie
- *rep_data* – dane pobrane po wykonaniu instrukcji LOAD
- *rep_store* – sygnalizator instrukcji STORE
- *rep_load* – sygnalizator instrukcji LOAD

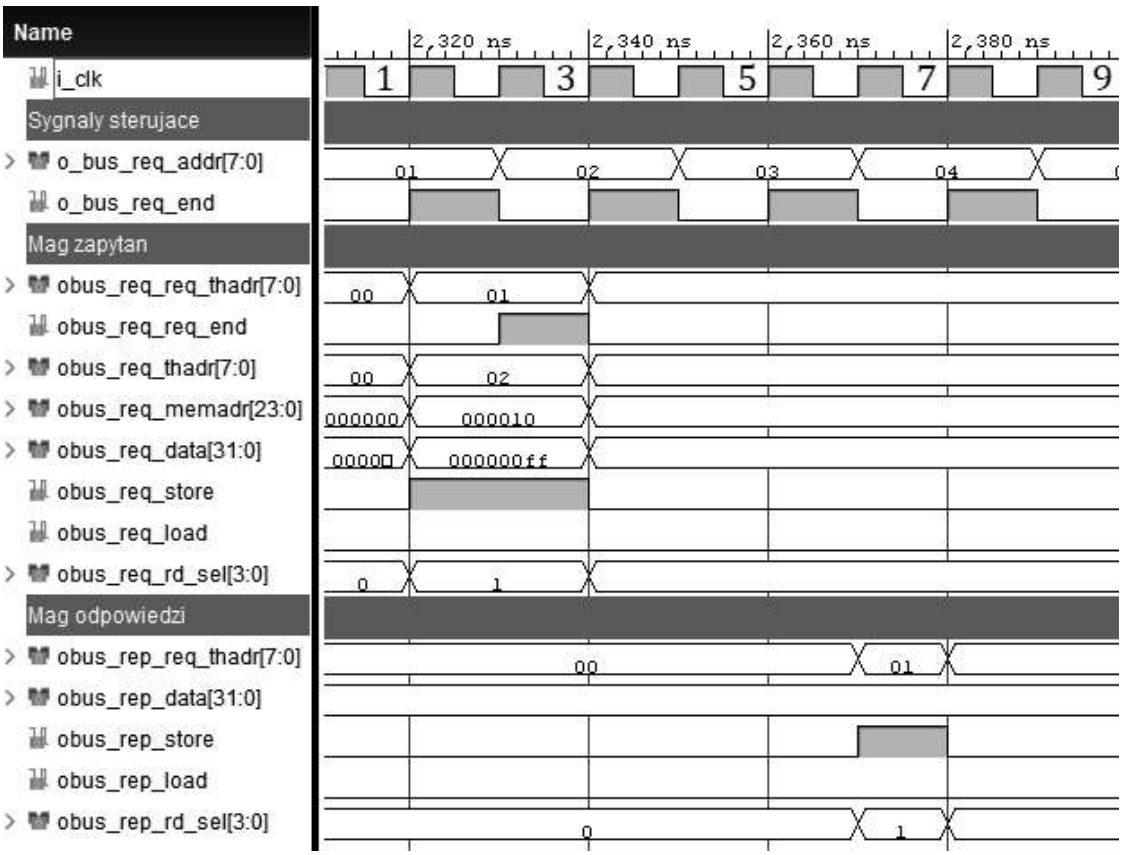
- *rep_rd_sel* – numer rejestru, do którego ma być zapisana dane odczytane instrukcją LOAD

Na Rys. 43 pokazano przykładowy program wykorzystujący wymianę danych między wątkami. Program ten pracuje w wątku pierwszym jednordzeniowego 10-cio wątkowego systemu. W linii drugiej (w pamięci lokalnej wątku) zapisano adres globalny komórki pamięci, w której będą zapisane, a następnie odczytywane dane. Pierwsze osiem bitów tego adresu to adres wątku. Pozostała jego część to adres komórki pamięci. Dane będą więc zapisywane w pamięci wątku drugiego (0x02) w komórce o adresie 0x00010. W linii 6 następuje zapisanie do rejestru zero wyżej omawianego adresu. Następnie w linii 7 następuje zapisanie zawartości rejestru zero (wartość 0xFF) do pamięci wątku drugiego. Przebiegi sygnałów magistrali takiej sytuacji pokazano na Rys. 44.

```
1  _mem:
2  0x01 = 0x02000010
3  _start:0x00000200
4     MOV  .R0, 0x01
5     MOV  .R1, 0xFF
6     LDR  .R0, [R0]
7     STR  .R1, [R0]
8     LDR  .R1, [R0]
9     MOV  .R14, #1
10 koniec:
11    B    .koniec
```

Rys. 43. Przykładowy program wykorzystujący wymianę danych między wątkami

W takcie 1 BC zezwolił wątkowi pierwszemu na wysłanie zapytania trwającego 2 takty zegara (sygnał „*o_bus_req_adr”*). W takcie drugim nastąpiło wysłanie zapytania. Przesunięcie o jeden takt wynika z potokowej idei magistrali, gdzie sygnały sterujące wyprzedzają sygnały magistrali o jeden takt zegara. Dzięki temu nie trzeba wykonywać wszystkich operacji podczas jednego cyklu zegara. Wątek pierwszy („*obus_req_req_thadr”*) wysyła zapytanie o komórkę pamięci o adresie „*obus_req_memadr”* wątku 2 („*obus_req_thadr’*). Zapytanie to wnosi o zapisanie danych („*obus_req_store”*) 0x000000FF („*obus_req_data”*).

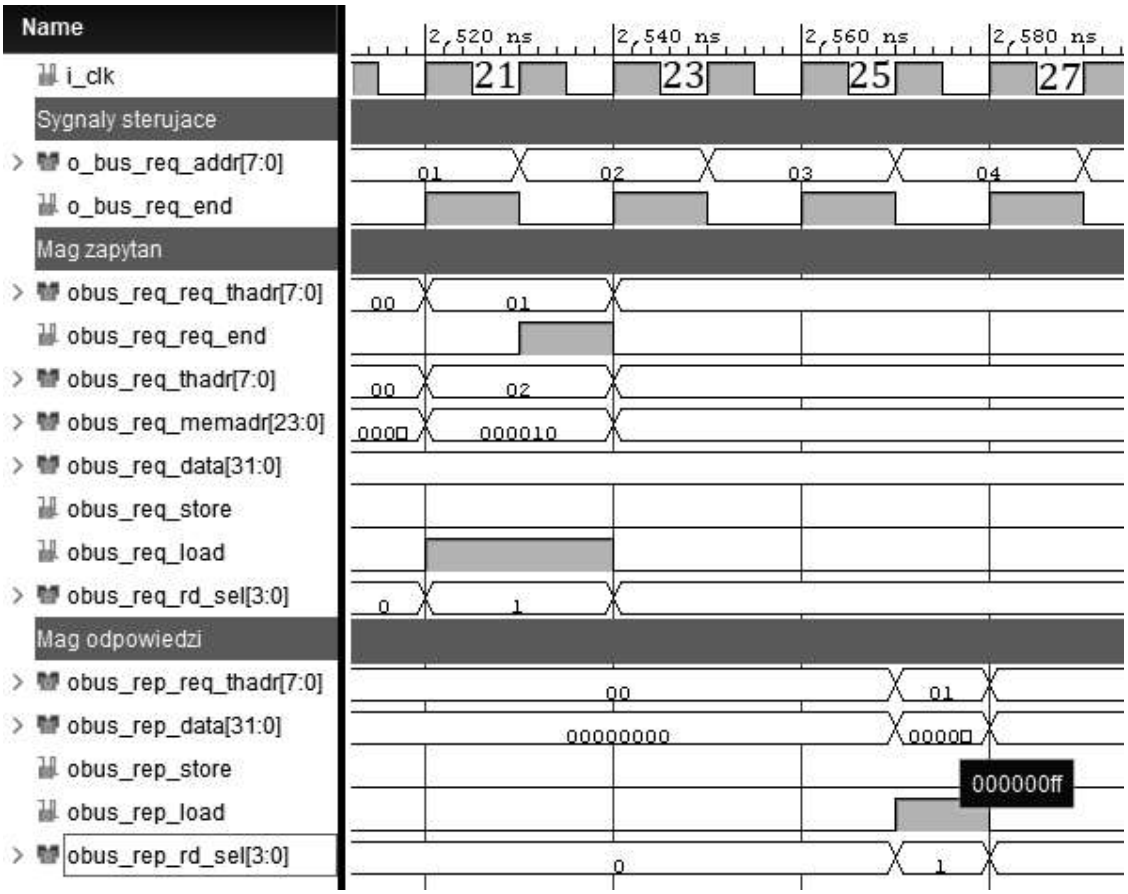


Rys. 44. Przykład wysyłania danych przez magistrale (instrukcja STORE)

W przypadku operacji zapisu (STORE) magistrala odpowiedzi przesyła tylko potwierdzenie zapisanych danych. Następuje to w takcie 7. Przysłane są tylko dane o numerze wątku przesyłającym zapytanie („*obus_rep_req_thadr”*), rodzaju operacji („*obus_rep_store”*) oraz o numerze rejestru wątku wysyłającego zapytanie, z którego były zapisywane dane („*obus_rep_rd_sel”*). Sygnały magistrali odpowiedzi są aktywne zawsze tylko podczas ostatniego taktu zapytania, ponieważ podczas tego taktu już na pewno dane zostały odczytane lub zapisane do pamięci. Przesunięcie 3 taktów między zapytaniem, a odpowiedzią wynika z przetwarzania zapytania i zostanie wyjaśnione w kolejnym rozdziale (Kontroler magistrali rdzenia).

Wykonanie instrukcji LOAD (Rys. 43 linia 8) jest bardziej skomplikowane. W poprzedniej instrukcji do adresu 0x02000010 (pamięć wątku 2) została zapisana wartość 0xFF. Instrukcja LOAD odczytuje z powrotem tą wartość. W takcie 21 rozpoczyna się zapytanie operacji odczytu (Rys. 45). W tym wypadku podczas zapytania nie są wysyłane żadne dane („*obus_req_data”*). Sygnalizator odczytu („*obus_req_load”*)

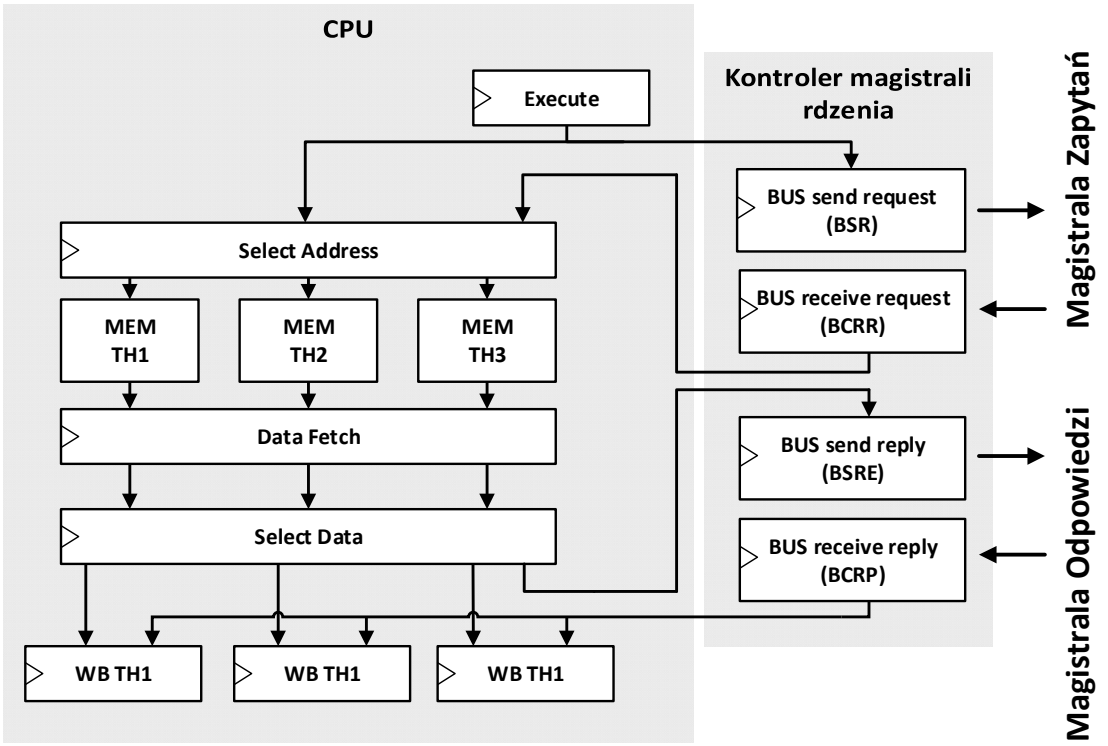
posiada poziom wysoki. Wartość sygnału „*obus_req_rd_sel”* ma tym razem znaczenie, ponieważ po odczytaniu danej to właśnie do tego rejestru wątku wysyłającego zapytanie („*obus_req_req_thadr”*) ma być ta dana zapisana. W takcie 26 jest odsyłana odpowiedź na zapytanie. Dana 0xFF („*obus_rep_data”*) została odczytana z pamięci i powinna być zapisana w rejestrze 1 („*obus_rep_rd_sel”*) wątku 1 („*obus_rep_req_thadr”*). Początek wykonania instrukcji LOAD dopiero w 21 takcie wynika z tego, że jest to system 10 wątkowy (każda instrukcja wykonuje się co 10 taktów). Zapytanie LOAD, wątku 1 musi również oczekiwać na prawo transmisji zapytania (stan 0x01 sygnału: *„o_bus_req_addr”*)



Rys. 45. Przykład odbierania danych przez magistralę (instrukcja LOAD)

5.3.4.2 Kontroler magistrali rdzenia

W każdym z rdzeni znajduje się zintegrowany z potokiem przetwarzania, **kontroler magistrali rdzenia (BTC)** pokazany na Rys. 46. W jego skład wchodzą 4 moduły zajmujące się poszczególnymi etapami transmisji danych.



Rys. 46. Kontroler magistrali rdzenia

Pierwszy z nich to **BUS send request (BSR).** Moduł ten przejmuje zapytania z fazy potoku EXE i przetrzymuje je, aż do momentu, gdy BSR dostanie prawo transmisji ich na magistrali. W momencie dostania zapytania wysyła też sygnał do CPU, aby zatrzymać przetwarzanie wątku, które wysłało zapytanie. Jako jedyny z modułów BTC, BSR musi posiadać dodatkowa pamięć przechowywującą te zapytania. Pamięć ta nie jest jednak duża i zawiera tyle zestawów komórek pamięci, ile rdzeni przetwarza dany rdzeń. Wątki, które wysłały zapytanie nie wyślą już kolejnego, ponieważ na czas jego przetwarzania wykonywanie ich jest zatrzymane.

Modułem odbierającym zapytania z magistrali jest **BUS receive request (BCRR)**. BCRR nasłuchuje magistrali i gdy zapytanie dotyczy wątku, który znajduje się w tym

samym rdzeniu co BTC, przesyła je do fazy potoku SA. Zapytanie to standardowo trwa 2 takty zegara (długość ta jest konfigurowalna: Rys. 42), ponieważ daje to pewność, że w którymś z tych dwóch taktów pamięć, do której odwołuje się zapytanie będzie nieużywana przez CPU. Następnie odczytane dane wraz z zapytaniem są przesyłane do fazy potoku DF i później SD.

W fazie SD dane odczytane z pamięci są przesuwane do ostatniego taktu zapytania (jeżeli zostały już odczytane podczas pierwszego taktu zapytania). Tak przygotowane dane trafiają do modułu **BUS send replay (BSRE)**. W następnym cyklu zegara BSRE przesyła odczytane dane wraz z dodatkowymi informacjami magistralą odpowiedzi. Odpowiedź jest nadawana tylko w ostatnim cyklu zapytania (w przypadku zapytania trwającego 2 takty zegara, w co drugim. Pokazano to na Rys. 44 oraz Rys. 45).

Nie ma potrzeby wprowadzania dodatkowych mechanizmów wprowadzających prawo transmisji na magistrali odpowiedzi. Zakładając, że przetwarzanie zapytań w każdym rdzeniu trwa tyle samo, odpowiedzi będą wysyłane kolejno na magistrale i taktowane przez zapytania pochodzące z magistrali zapytań.

Ostatnim modułem BTC jest **BUS receive reply (BCRP)**. Odbiera on odpowiedzi z magistrali. Po otrzymaniu takiej odpowiedzi, wysyła on sygnał do uruchomienia wątku, który wysyłał zapytanie. W przypadku odpowiedzi na instrukcje LOAD zapisuje on też odczytaną daną w odpowiednim rejestrze odpowiedniego wątku (informacje te są wysyłane magistralą odpowiedzi).

Długość przetwarzania zapytania oznaczono symbolem R_{dur} wzór (7). Jest ona zależna od liczby etapów potoku pomiędzy etapami SA oraz SD. W standardowej maksymalnej konfiguracji (gdy każdy etap SA, DF oraz SD jest etapem potoku) trwa ona 13 cykli zegara. Dwa cykle zajmuje uzyskanie prawa nadania zapytania. Kolejne cztery to przesłanie zapytania i odpowiedzi. Trzy kolejne to przetwarzanie przez fazy potoku SA, DF oraz SD. Ostatnie cztery to uruchomienie zatrzymanego wątku oraz zapisanie ewentualnej odpowiedzi do odpowiedniego rejestru.

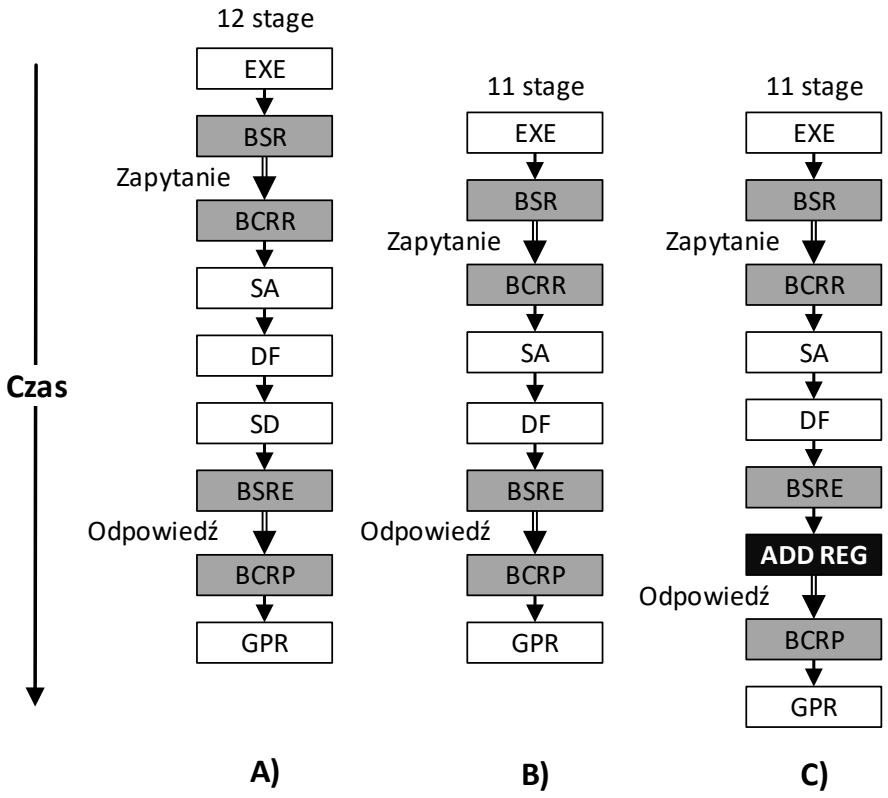
$$R_{dur} = Macces_{dur} + 10 \quad (7)$$

gdzie:

R_{dur} – Czas przetwarzania zapytania wymiany danych między wątkami

$Macces_{dur}$ – Liczba etapów potoku pomiędzy fazą SA, a SD (z tymi etapami włącznie)

W sytuacji, w której w systemie pracują rdzenie z różnym parametrem R_{dur}, do poprawnego działania magistrali trzeba dopasować przetwarzanie innych rdzeni do tego z największym R_{dur}. Zamiast spowalniać całe szybsze rdzenie systemu można wydłużyć czas przetwarzania etapu BSRE tak, aby dopasować transmisję danych szybszych rdzeni do wolniejszych. Sytuację taką przedstawiono na Rys. 47.



Rys. 47. Dodatkowe rejestry wyrównywujące w BSRE

W przypadku, gdy w systemie pracują 2 rdzenie o różnej długości potoku (Rys. 47 A i B), odpowiedzi pochodzące z rdzenia A i B zostają wysłane w tym samym momencie. Taka sytuacja jest niedopuszczalna. Aby uniknąć takiego zdarzenia wprowadza się dodatkowe rejestry w module BSRE. Opóźniają one wysyłanie odpowiedzi przez szybszy rdzeń. Na Rys. 47 C zaznaczono je czarnym kolorem. Rdzenie A i C będą pracować w takim systemie prawidłowo (rdzeń C to rdzeń B z dodatkowym rejestrem)

Liczbę dodatkowych rejestrów wprowadza się poprzez zmianę parametru REP_DELEY w nagłówku modułu rdzenia.

5.3.4.3 Typy obsługiwanych wątków

Doświadczenia zdobyte podczas projektowania oraz badań pierwszej wersji magistrali pozwoliły wprowadzić dodatkowe mechanizmy obsługi wątków. Podczas użytkowania poprzedniej wersji zauważono, że zazwyczaj nie wszystkie wątki pracujące w systemie potrzebują wymiany danych z innymi. Podzielono więc wątki na trzy typy:

- Wątek niezależny (NCT) – jest to wątek, który nie używa instrukcji LOAD i STORE do wymiany danych z innymi wątkami
- Precyzyjne wątki współpracujące (CTHM) – są to wątki, w których wykonanie instrukcji wymiany danych z innym wątkiem trwa dokładną liczbę taktów równą parametrowi M_{durII}
- Standardowe wątki współpracujące (CT) – są to wątki, w których czas wykonania instrukcji wymiany danych z innym wątkiem trwa nie większą liczbę taktów niż parametr M_{durII}, lecz zazwyczaj ta operacja będzie wykonana szybciej.

Rozróżnienie wątków na CT, CTHM oraz NCT pozwoli nie przydzielać przedziałów czasowych na nadawanie na magistrali wątkom NCT. Dzięki temu czas wykonania instrukcji wymiany danych przez wątki CT i CTHM będzie krótszy. W nagłówku modułu BTC (Rys. 42 oraz Rys. 48) występuje sygnał „*i_reg_th_address”*. Wartość tego sygnału oznacza do jakich wątków i w jakiej ich kolejności będą przydzielane prawa dostępu wysyłania zapytania przez magistralę. Przypisując adresy wątków do tego sygnału można pominąć wątki NCT. Dzięki temu czasy wykonywania zapytań wymiany danych będą krótsze.

```
89  bus_ctrl_i
90  (
91      .i_clk                      (i_clk),
92      .i_rst                      (i_rst),
93
94      // Sygnaly sterujące
95      .i_req_th_address           ({8'd10,8'd9,8'd8,8'd7,8'd6,
96                                    8'd5,8'd4,8'd3,8'd2,8'd1}),
```

Rys. 48. Parametry określające wątki zależne

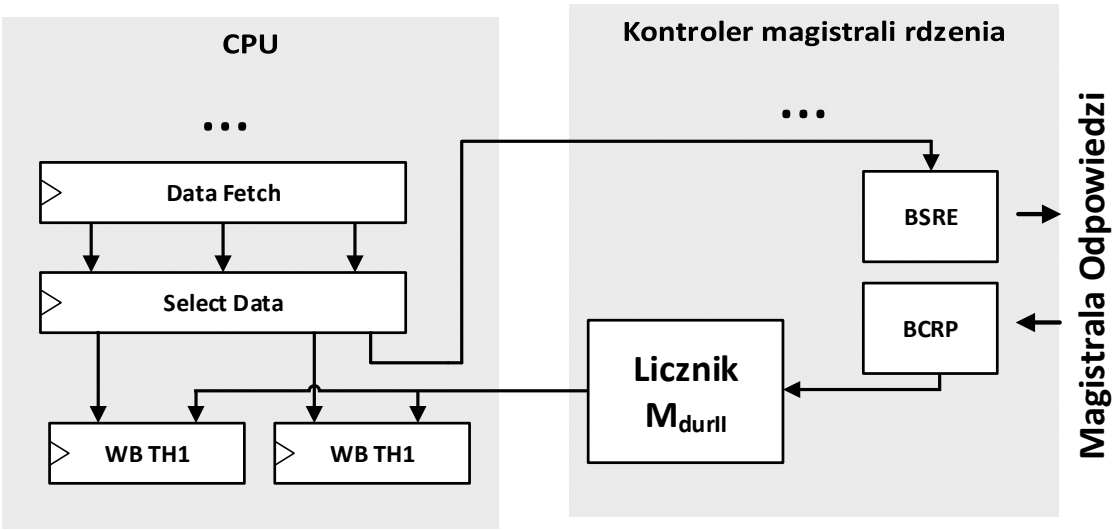
Moduły BSRE oraz BCRP wątków NCT pozostają bez zmian. Oznacza to, że mogą być one adresatami zapytań innych wątków. Istnieje więc możliwość odczytania lub zapisania pamięci wątku NCT przez wątek CT lub CTHM.

Modyfikacja magistrali pozwoliła na wprowadzenie wątków, w których czas trwania operacji wymiany danych wynosi dokładnie M_{durII}(9)(10) (pochodzenie tego parametru zostanie dokładnie wyjaśnione w następnym podrozdziale 5.3.4.4).

Podział na zadania CT i CTHM pozwoli wykonywać zadania, które są bardzo rygorystyczne czasowo i w których jest wymagana synchroniczna współpraca z otoczeniem (np. zadania SHT). Jednocześnie dzięki wprowadzeniu wątków CT możliwe jest szybsze wykonywanie zadań, w których synchronizacja z otoczeniem nie jest wymagana.

Dokładna synchronizacja czasowa wątków CTHM jest realizowana przez dodatkowy licznik taktów zegara pokazany na Rys. 49.

Gdy moduł BCRP odbierze odpowiedź na wcześniej odebrane zapytanie, sprawdza czy wątek je wysyłający był wątkiem CTHM. Jeżeli nie, wątek ten zostaje natychmiast uruchomiony. Gdy wątek wysyłający zapytanie to wątek CTHM, już w momencie wysłania zapytania Licznik M_{durII} (Rys. 49) zostaje wyzerowany i zaczyna zliczanie taktów zegara. Jak tylko moduł BCRP odbierze odpowiedź na wysłane zapytanie, zostaje ono zapisane do rejestru i przechowywane aż Licznik M_{durII} osiągnie wartość równą M_{durII}. Dopiero, gdy ta wartość zostanie osiągnięta, wątek wysyłający zapytanie zostaje uruchomiony.



Rys. 49. Licznik czasu wykonywania instrukcji wymiany danych z innymi wątkami

Przed syntezą systemu, w nagłówku modułu rdzenia ustawiane są parametry, które kontrolują podział na wątki CTHM i CT oraz kontrolują ich pracę. Parametry te pokazano na Rys. 50. Za pomocą parametru HARD_TH binarnie ustawiane są wątki, które pracują w tym rdzeniu jak wątki CTHM. Parametr CORE_TH_IDS przyporządkowuje wątki do rdzenia. Każdy z nich odpowiada jednemu bitowi parametru HARD_TH. Jedynka na odpowiednim bicie oznacza, że odpowiadający jej wątkiem jest wątkiem CTHM. Natomiast zero, że jest to wątek CT. W przykładzie z Rys. 50 tylko wątek o adresie 0x07 jest wątkiem CTHM. Parametr MDUR_TIME ustawia czas, po którym ma być wykonana wymiana danych (wartość, do której odlicza Licznik M_{durII}). Za pomocą parametru MDUR_COUNT_WIDTH ustawia się maksymalną liczbę bitów licznika M_{durII}.

```
.CORE_TH_IDS       ({8'd10,8'd9,8'd8,8'd7,8'd6,8'd5,8'd4,8'd3,8'd2,8'd1}),

.MDUR_COUNT_WIDTH  (MDUR_COUNT_WIDTH),
.MDUR_TIME         (MDUR_TIME     ),
.HARD_TH           (10'b0001000000)
```

Rys. 50. Parametry wątków CTHT i CT

5.3.4.4 Przewidywalność czasowa wymiany danych

Czas trwania wymiany danych w implementacji magistrali jednokierunkowej można podzielić na 2 etapy. Pierwszym z nich jest oczekiwanie na prawo wysłania zapytania na magistrali, a drugi to faktyczne wykonanie zapytania.

Czas pierwszego etapu jest zależny od długości trwania zapytania, czyli parametru REQ_CYCLES (Rys. 42). W standardowych testowanych konfiguracjach wartość tego parametru wynosi 2. Zapytanie przypada na każdy wątek współpracujący (CT i CTHM) pracujący w systemie. Sposób obliczania maksymalnego czasu dostępu do magistrali pokazano we wzorze (8).

$$Bacces_{dur} = CT_{num} \bullet REQ_CYCLES \tag{8}$$

gdzie:

$Bacces_{dur}$ – Maksymalny czas oczekiwania na dostęp do magistrali

CT_{num} – Liczba wątków współpracujących pracujących w systemie

REQ_CYCLES – Długość trwania zapytania

Czas oczekiwania na dostęp do magistrali w przypadku wątków ST nie jest jednak stały i zależy od momentu wysłania zapytania. W najlepszym możliwym przypadku (jeżeli zapytanie jest wysłane w momencie, w którym wątek wysyłający zapytanie w przyszłym takcie zegarowym będzie miał dostęp do magistrali) będzie ono wykonane podczas jednego taktu zegara. Maksymalny czas tej operacji wyniesie $Bacces_{dur}$ (8). W przypadku wątków CTHM ten czas jest stały i wynosi dokładnie $Bacces_{dur}$.

Drugą składową czasu wymiany danych jest R_{dur}(7). Czas ten odnosi się do samego przetwarzania zapytania. Jest on zależny od konfiguracji rdzenia (liczbie jego etapów potoku). Dla wątków należących do różnych rdzeni w systemie czas ten może być różny. Czas przetwarzania zapytania jest stały i ma taką samą wartość dla wątków CT i CTHM.

Maksymalny czas wykonania wymiany danych oznaczany jako M_{durII}(9), jest sumą $Bacces_{dur}$ oraz R_{dur}.

$$M_{durII} = Bacces_{dur} + R_{dur} \quad (9)$$

gdzie:

M_{durII} – Maksymalny czas wykonania wymiany danych

$Bacces_{dur}$ – Maksymalny czas oczekiwania na dostęp do magistrali

R_{dur} – Czas przetwarzania zapytania wymiany danych między wątkami

Poniżej przedstawiono również opis M_{durII}(10) zawierający wszystkie podstawowe składowe mające wpływ na jego wartość.

$$M_{durII} = CT_{num} \bullet REQ_CYCLES + Macces_{dur} + 10 \quad (10)$$

gdzie:

M_{durII} – Maksymalny czas wykonania wymiany danych

CT_{num} – Liczba wątków współpracujących pracujących w systemie

REQ_CYCLES – Długość trwania zapytania

$Macces_{dur}$ – Liczba etapów potoku pomiędzy fazą SA a SD (z tymi etapami włącznie)

5.3.5 Porównanie wersji magistral

Magistrala jednokierunkowa powstała jako ulepszona wersja magistrali dwukierunkowej. Głównym problem wynikającym ze stosowania jej pierwszej wersji był czas M_{durI}(6), który był uzależniony od liczby rdzeni systemu. Liczba stanów IC magistrali musiała być równa liczbie rdzeni systemu. Problemem z tego wynikającym okazał się fakt, że często wynikiem harmonogramowania zadań była struktura sprzętowa systemu, w której mieściła się liczba jej rdzeni. Do obliczenia czasu wymiany danych, jak i również harmonogramowania zadań, niezbędna była liczba rdzeni systemu. Taka sytuacja prowadziła do długotrwałych iteracyjnych obliczeń, ponieważ przy takiej

metodologii harmonogramowania, najlepszą metodą było zakładanie liczby rdzeni, harmonogramowanie ich i sprawdzanie czy mieszczą się w zakładanych ograniczeniach. W przypadku niespełnienia wymagań, zakładaną liczbę rdzeni zwiększano lub zmniejszano i całą operację powtarzano do skutku, aż ograniczenia zostały spełnione. Metodologia tego typu nie jest zbyt optymalna. W wersji magistrali jednokierunkowej czas $M_{durII}(10)$ zależy tylko od liczby wątków CT i CTHM pracujących w systemie oraz struktury rdzenia, w którym pracuje dany wątek (obie informacje są znane przed harmonogramowaniem zadań).

Drugim znaczącym usprawnieniem było zmniejszenie skomplikowania magistrali kosztem liczby sygnałów magistrali oraz sygnałów sterujących. Zmiany takie wyeliminowały potrzebę stosowania stanów i cykli magistrali (SC, IC i LC), co z kolei przełożyło się na szybszą pracę oraz intuicyjność działania magistrali. W obecnej wersji, po spojrzeniu na jeden takt sygnałów magistrali, można w prosty sposób sprawdzić jaka dana skąd i dokąd jest wysyłana. Po implementacji jednej z pierwszych wersji magistrali jednokierunkowej odczytano, że najdłuższy czas propagacji ścieżki wynosi 1,312ns (Rys. 51). Co oznacza, że teoretyczny maksymalna częstotliwość pracy zegara taktującego magistralę może wynosić 761 MHz. Wartość ta znacznie przewyższa maksymalne taktowanie rdzenia systemowego.

Name	Path 1
Slack	0.518ns
Source	test_bus_i/rst_clk_wiz_100M/U0/EXT_LPF/POR_SRL_I/CLK (rising
Destination	test_bus_i/rst_clk_wiz_100M/U0/EXT_LPF/lpf_int_reg/D (rising edge
Path Group	clk_out1_test_bus_clk_wiz_0
Path Type	Setup (Max at Slow Process Corner)
Requirement	2.000ns (clk_out1_test_bus_clk_wiz_0 rise@2.000ns - clk_out1_test_bu
Data Path Delay	1.313ns (logic 1.043ns (79.436%) route 0.270ns (20.564%))
Logic Levels	1 (LUT4=1)
Clock Path Skew	-0.145ns
Clock Un...rtainty	0.060ns

Rys. 51. Najdłuższy czas propagacji ścieżki w magistrali jednokierunkowej

W poprzedniej wersji (magistrala dwukierunkowa) to właśnie magistrala ograniczała maksymalne taktowanie rdzenia systemowego. Spowodowane to było symulowanymi stanami wysokiej impedancji (synteza zamieniała stany wysokiej

impedancji na sterowanie multiplekserem według swojego algorytmu, użytkownik nie miał nad tym kontroli) oraz dużą liczbą pamięci, którą wykorzystywała magistrala.

Magistrala jednokierunkowa wprowadziła również podział na wątki CT, CTHM oraz NCT, co pozwoliło przeprowadzić dodatkowe badania. Umożliwiło to również lepsze dopasowanie systemu oraz sprzętu do wykonywanych zadań. Porównanie magistral pokazano w Tab. 3.

Tab. 3. Porównanie magistrali jednokierunkowej oraz dwukierunkowej

	Magistrala dwukierunkowa	**Magistrala jednokierunkowa**
Zależność czasu M_{dur}	Liczba rdzeni, liczba wątków, struktura rdzenia	Liczba wątków, struktura rdzenia
Sposób przechowywania zapytań (na jeden wątek)	4 czteropoziomowe zestawy pamięci	1 zestaw rejestrów (2 zestawy rejestrów w przypadku wątku CTHM)
Skomplikowanie protokołu wymiany danych	Większe	Mniejsze
Liczba sygnałów magistrali	8	13 na każdy rdzeń systemu + 2
Obsługa zadań NCT, CT i CTHM	Nie	Tak

5.4 Kompilator zadań

Wszystkie wykonywane instrukcje przez system zapisywane są do pamięci programu w kodzie maszynowym. Aby ułatwić proces implementacji programów zdecydowano się zaprojektować i zaprogramować program kompilatora. Projektowany system posiada dodatkowe instrukcje rozszerzające standardową listę ARM ISA oraz posiada możliwość sprzętowej wymiany danych z innymi wątkami. Dlatego korzystanie z gotowych kompilatorów było niemożliwe.

Kompilator (Rys. 52) został zaprogramowany w języku C# [71] (C Sharp) za pomocą programu Visual Studio 2017. Pozwala on na kompilację wczytanego programu zapisanego w asemblerze (klawisz „Read ASM") na kod maszynowy („Hex File"). Odczytany program znajduje się w oknie „Assembler Code". W okienku „Start address (HEX)" można wybrać od której komórki pamięci zostanie zapisany kompilowany program. Spowoduje to również wpisanie do pierwszej komórki pamięci programu bezwarunkowego skoku do tego adresu, tak, aby program został wykonany poprawnie. Omawiany program pozwala także na proste debugowanie programu. W oknie „Address Labels" pokazane są wszystkie znalezione w kodzie etykiety skoków oraz przypisane im adresy. Natomiast w oknie „Address" nadane są adresy każdej linii kodu występującego w programie. W oknie „Errors" pojawiają się błędy składniowe kodu.

Program można skompilować do pliku wsadowego pamięci przyciskiem „Save Hex". Niestety po dłuższej pracy z kompilatorem okazało się to niewygodne. Każdy nowy plik wsadowy pamięci musiał być dodatkowo dołączany do bibliotek projektu programu Vivado, co wiązało się z niewydajną pracą. Dlatego zdecydowano się zaprojektować nową funkcjonalność kompilatora zadań. Polega ona na tworzeniu modelu pamięci wraz z zawartością w języku Verliog za pomocą makra BRAM_SINGLE_MACRO [50]. Kompilator tworzy plik o takiej samej nazwie i takiej samej nazwie modułu, co wskazany podczas jego zapisu (klawisz „Save BRAM"). W ten sposób moduł z zapisanym skompilowanym programem podmieniany jest z wcześniejszym modułem pracującym w systemie, co usuwa niedogodność dodawania nowych plików wsadowych pamięci do bibliotek projektu. Przyciski „Read, Compile and SaveHEX" oraz „Read, Compile and SaveBRAM" pozwalają wykonać wszystkie trzy operacje za pomocą jednego przycisku, jeżeli już wcześniej wskazana była ścieżka odczytu i zapisu pliku. Takie rozwiązanie znacznie przyspieszyło wprowadzanie poprawek do programu.

Rys. 52. Kompilator zadań

6 Metodologia harmonogramowania zadań

Harmonogramowanie zadań jest pierwszym etapem projektowania systemu. Po jego przeprowadzaniu uzyskane będą niezbędne dane potrzebne do implementacji sytemu, takie jak liczba rdzeni systemu, częstotliwość jego pracy oraz sposób przyporządkowania zadań do rdzeni. Wynikiem harmonogramowania są również dane pochodzące z symulacji. Zawierają one między innymi minimalny i maksymalny czas wykonania każdego harmonogramowanego zadania, w zaproponowanej strukturze sprzętowej.

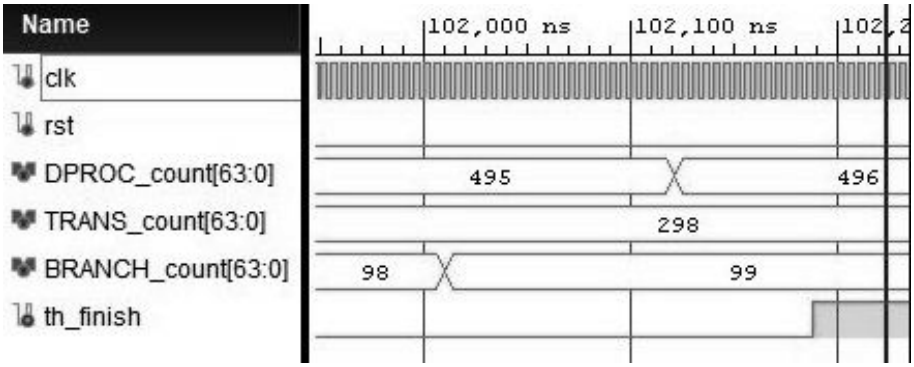
Harmonogramowanie składa się z kilku części. Schemat całego procesu został przedstawiony i omówiony w rozdziale „4.8 Model harmonogramowania zadań" na stronie 35. Model zadania wraz z opisem parametrów został przedstawiony i omówiony w rozdziale „4.2 Model zadania" na stronie 23.

6.1 Pozyskiwanie parametrów zadań

Pierwszym etapem procesu harmonogramowania jest pozyskanie wartości parametrów C_i, M_i oraz D_i. Wartość parametru D_i jest założonym maksymalnym czasem wykonania zadania, jest więc znany wprost od początku harmonogramowania. Problemem może być pozyskanie dwóch pozostałych parametrów. Docelowo pozyskiwanie wartości tych parametrów powinno odbywać się już na poziomie kompilacji programu, za pomocą wbudowanego symulatora w kompilatorze. Wraz z wykorzystaniem technik programowania „Single path programing" [72], będzie możliwe uzyskanie parametrów takich zadań. Jednak celem tej pracy nie było skupienie się na technikach programowania. W przypadku zadań NCT możliwe jest również uzyskanie parametru C_i, przy pomocy maksymalnego czasu wykonania takiego zadania, przy określonej częstotliwości pracy systemu [73].

Na potrzeby tej pracy parametry zadań pozyskiwano na dwa sposoby. W pierwszym z nich parametry Ci zadań pozyskano za pomocą symulacji, którą umożliwia narzędzie Vivado. Bazując na zaproponowanej architekturze (5.1 Architektura systemu, strona 27), zaprojektowano dedykowany model jednozadaniowego systemu, przeznaczonego do analizy zadań. W modelu tym zaimplementowano dodatkowe rejestry zliczające rodzaj wykonywanych instrukcji oraz sygnał oznaczający zakończenie pracy zadania. Przykładowe przebiegi sygnałów z symulacji pokazano na Rys. 53. W modelu tym

uruchomiono zadania NCT wzorowane na programach pochodzących z zestawu testów, zaprojektowanych do sprawdzenia wydajności i przewidywalności czasowej systemów czasu rzeczywistego [74], [75]. Instrukcje „DPROC" są instrukcjami przetwarzania danych, „TRANS" to instrukcje przesyłania danych, a „BRANCH" to instrukcje warunkowe. Suma wartości tych rejestrów w momencie zakończenia pracy zadania (zmiany stanu sygnału „th_finsh" z niskiego na wysoki) jest szukaną wartością C_i.



Rys. 53. Sposób pozyskiwania wartości C_i zadań

Drugą metodą pozyskiwania parametrów zadań pozwalającą zweryfikować prace systemu było stworzenie syntetycznych programów, w których parametry C_i oraz M_i zależą tylko i wyłącznie od parametrów zawartych w programach. Sposób taki pozwolił uzyskać również uzyskać parametry zadań CT i CTHM oraz znacznie przyśpieszył proces przygotowania systemu, co pozwoliło zwiększyć liczbę eksperymentów. Zadania wygenerowane w taki sposób działają tak samo jak standardowe zadania zaprogramowane wg techniki „Single path programing".

6.2 Przykładowe zadania

Przed omawianiem procesu harmonogramowania przygotowano zestaw 60 testowych zadań. Pozwoli to dokładniej opisać tę metodologię. Jako przykład wygenerowano syntetycznie 60 zadań o losowym typie oraz losowych parametrach C_i, M_i oraz D_i. Zadania były losowane wg następujących ograniczeń:

- Maksymalne $C_i = 3000$
- Maksymalne $M_i = 100$

- Maksymalne $TF_i = 5$
- Prawdopodobieństwo wylosowania zadania typu SHT = 5%
- Prawdopodobieństwo wylosowania zadania typu CT = 25%
- Prawdopodobieństwo wylosowania zadania typu CTHM = 15%

Ograniczając maksymalną wartość TF_i, losowano D_i w taki sposób, aby zadania nie miały zbyt dużego zapotrzebowania na moc obliczeniową. W Tab. 4 przedstawiono wylosowane parametry zadań wraz z typem zadań oraz obliczonym parametrem TF_i (M_{dur} = 61, $Min_{indistance}$ = 9). Zostało wylosowanych 36 zadań typu NCT, 17 zadań typu CT oraz 7 zadań typu CTHM. Cztery zadania spośród nich są zadaniami typu SHT.

Tab. 4. Parametry zadań realizacji systemu

ThID	C_i	M_i	D_i [ns]	SH	CTMH	TF_i	ThID	C_i	M_i	D_i [ns]	SH	CTMH	TF_i
1	1919	50	1902114	Nie	Tak	1,1939	31	2930	0	1442498	Nie	Nie	2,0326
2	2975	93	1448928	Nie	Nie	2,5039	32	2513	0	1034294	Nie	Nie	2,4316
3	626	0	1280929	Nie	Nie	0,4903	33	2210	0	1415204	Nie	Nie	1,5630
4	242	67	855631	Nie	Nie	0,8333	34	404	73	1034817	Tak	Tak	0,8861
5	2099	0	1922519	Tak	Nie	1,0928	35	1892	0	974619	Nie	Nie	1,9433
6	1979	0	1581164	Nie	Nie	1,2529	36	1346	0	1162232	Nie	Nie	1,1598
7	194	0	1980426	Nie	Nie	0,0990	37	1301	91	1250616	Nie	Nie	1,5512
8	2060	0	1612365	Nie	Nie	1,2789	38	959	85	1378492	Nie	Nie	1,1288
9	851	0	1088925	Nie	Nie	0,7833	39	134	0	187292	Nie	Nie	0,7261
10	401	22	1878523	Nie	Nie	0,2965	40	1424	55	1927832	Nie	Nie	0,9394
11	2363	0	1311330	Nie	Nie	1,8035	41	1253	39	1553470	Nie	Nie	0,9836
12	974	0	599793	Nie	Nie	1,6272	42	785	0	1094883	Nie	Nie	0,7188
13	506	54	1882301	Nie	Tak	0,4707	43	374	74	1834138	Nie	Tak	0,4874
14	953	0	720337	Nie	Nie	1,3258	44	1286	0	487666	Nie	Nie	2,6412
15	635	0	367229	Nie	Nie	1,7346	45	1466	81	543915	Nie	Tak	3,7414
16	1892	69	989144	Nie	Nie	2,4031	46	1847	10	492035	Nie	Nie	3,9001
17	2777	17	1251289	Nie	Nie	2,3160	47	2117	0	1267142	Nie	Nie	1,6723
18	2180	0	1225434	Nie	Nie	1,7806	48	1415	100	1551826	Nie	Nie	1,3642
19	2327	0	1016979	Nie	Nie	2,2901	49	2660	0	987253	Nie	Nie	2,6964
20	2915	0	1602334	Nie	Nie	1,8205	50	1685	0	1821961	Tak	Nie	0,9259
21	1898	0	769698	Tak	Nie	2,4685	51	2300	0	817140	Nie	Nie	2,8171
22	1016	0	1099207	Nie	Nie	0,9261	52	1460	50	1624262	Nie	Nie	1,1156
23	1601	0	1256100	Nie	Nie	1,2762	53	1691	0	731547	Nie	Nie	2,3143
24	275	21	902913	Nie	Nie	0,4696	54	2009	0	1003412	Nie	Nie	2,0042

ThID	C_i	M_i	D_i [ns]	SH	CTMH	TF_i	ThID	C_i	M_i	D_i [ns]	SH	CTMH	TF_i
25	602	0	422956	Nie	Nie	1,4280	55	1229	97	1725713	Nie	Tak	1,1068
26	1628	0	1900332	Nie	Nie	0,8577	56	734	0	990527	Nie	Nie	0,7430
27	1811	84	1891852	Nie	Nie	1,2691	57	2804	37	1446128	Nie	Nie	2,1195
28	2642	58	1924117	Nie	Nie	1,5851	58	665	56	768467	Nie	Tak	1,3781
29	317	0	1192492	Nie	Nie	0,2675	59	2159	35	1573557	Nie	Tak	1,5290
30	1352	0	989968	Nie	Nie	1,3677	60	590	0	1813628	Nie	Nie	0,3264

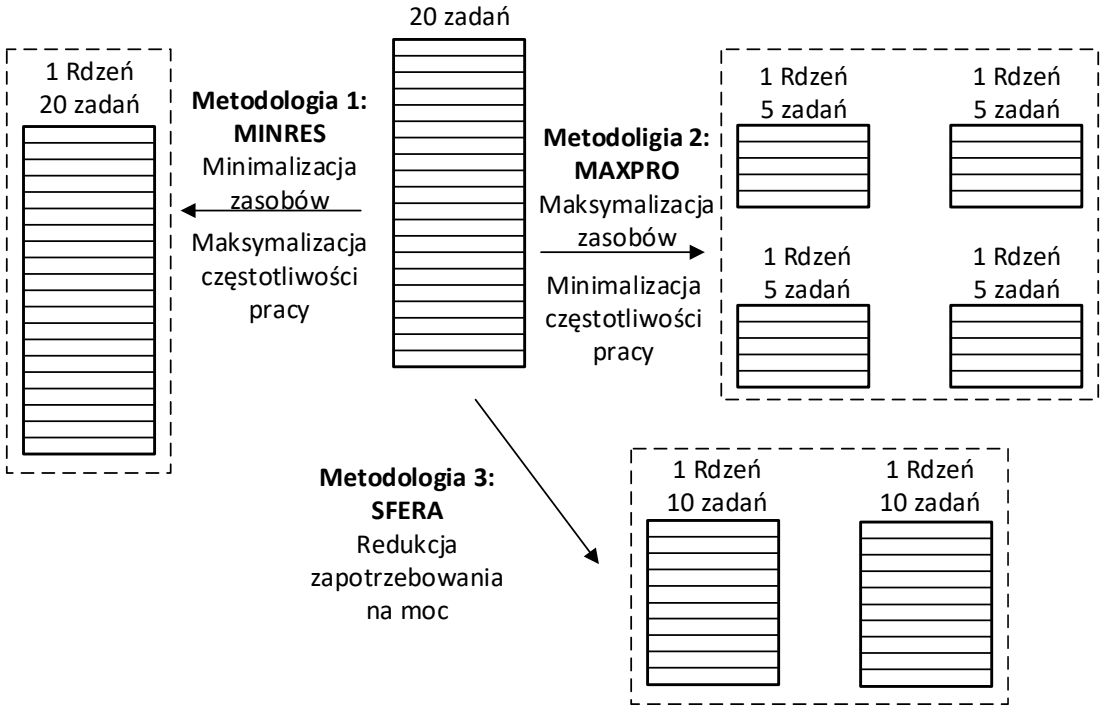
6.3 Podział zadań

Podział zadań jest pierwszym etapem harmonogramowania. Polega on na wybraniu odpowiedniej liczby rdzeni systemu oraz rozdysponowaniu harmonogramowanych zadań między tymi rdzeniami. Zadania powinny być przyporządkowane do rdzeni w taki sposób, aby możliwe było ich uszeregowanie w następnym etapie harmonogramowania oraz spełnienie ich wymagań czasowych. Rdzenie muszą pracować z taką samą częstotliwością, aby umożliwić ewentualną komunikację między nimi. Powoduje to, że przybliżona częstotliwość pracy rdzeni (F_{sys}), obliczona na tym etapie, powinna być jak najbardziej zbliżona dla każdego z rdzeni.

Istnieje wiele strategii omawianego podziału zadań. W przypadku potrzeby minimalizacji zasobów potrzebnych do implementacji systemu należy dążyć do minimalnej liczby rdzeni (Rys. 54 – Strategia 1). W takim wypadku liczba zasobów będzie najmniejsza, ponieważ w systemie będzie pracował tylko jeden potok przetwarzania (5.1.2 Potok przetwarzania danych, strona 39). Niestety spowoduje to wysoką częstotliwość pracy zegara systemowego, ponieważ wątki nie będą przetwarzane równolegle. Wysoka częstotliwość pracy systemu spowoduje większe zużycie energii przez system (4.7 Energooszczędność systemu, strona 34).

Drugą strategią (Rys. 54 – Strategia 2) jest dążenie do minimalizacji energii zużywanej przez system. W takim przypadku rdzeni powinno być jak najwięcej, ponieważ równoległe przetwarzanie zadań zmniejszy częstotliwość pracy zegara systemowego. Niestety w takim przypadku potok przetwarzania systemu jest zwielokrotniony, co powoduje zwiększenie zapotrzebowania na zasoby. Minimalna liczba zadań w rdzeniu jest również ograniczona przez kilka czynników. Pierwszym z nich jest minimalny przeplot systemu ($Min_{indistance}$, wzór (5) strona 44). Drugim

algorytm szeregowania (6.4 Szeregowanie zadań, strona 101), który działa wydajniej, jeżeli w rdzeniu jest więcej zadań. Dlatego należy znaleźć optymalną, minimalną liczbę rdzeni systemu.



Rys. 54. Metodologie podziału zadań

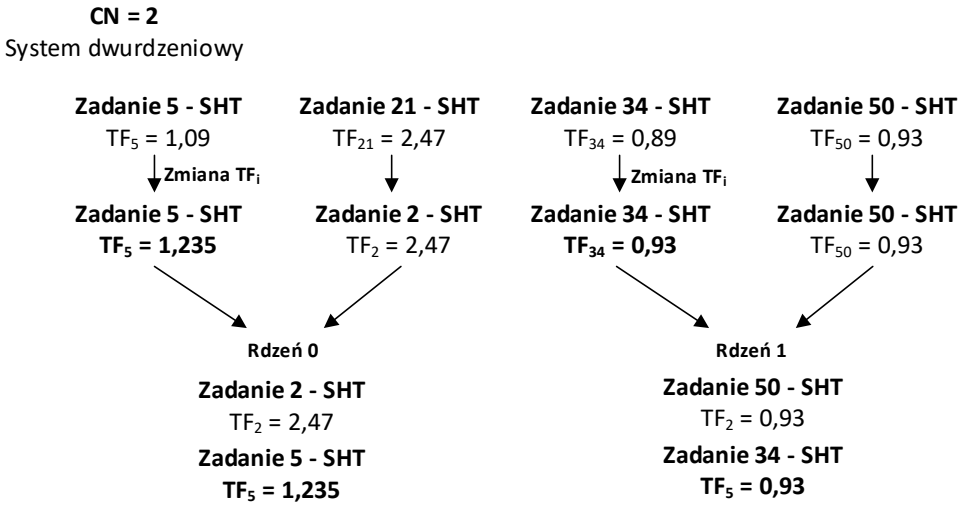
Pozostałe strategie dążą do uzyskania kompromisu pomiędzy zużyciem energii, a zapotrzebowaniem na zasoby. Wybrane strategie zostaną zaprezentowane w następnych podrozdziałach.

6.3.1 Algorytm BLTS

Algorytmy przedstawione w kolejnych podrozdziałach są ulepszoną wersją algorytmów BLIS, COTAS, BLIS II, COTAS II oraz metodologii SFERA i MAXPRO [53], [75]. Algorytmy te zostały przypomniane w Dodatku A – „Opublikowane ” na stronie 201. W aktualnej wersji systemu wartość parametru M_{dur} jest niezależna od liczby rdzeni systemu, nie jest więc wymagany podział na algorytmy BLIS i COTAS (podział zadań NCT i CT). W stosunku do poprzedniej publikowanej wersji systemu, wprowadzono nowe typy zadań: CTHM oraz SHT, zatem do podziału zadań nie można stosować poprzednich wersji algorytmów. Po zdobytych wcześniej doświadczeniach

zaproponowano uniwersalny algorytm podziału używany przy każdym typie zadania. Pozwoli to znacznie uprościć ten proces. Drugim czynnikiem wpływającym na uproszczenie powyższych algorytmów jest opracowanie symulatora wykonywania zadań. Pozwoliło to na zrezygnowanie z parametrów wprowadzających dodatkowe marginesy częstotliwości (parametr F_{margin}) podczas podziału zadań.

Nowy zaproponowany algorytm BLTS (ang. Balanced load task scheduling) bazuje na algorytmach BLIS i COTAS. Został on zaprojektowany do podziału wszystkich typów zadań, również nowo wprowadzonych zadań typu SHT. Na podstawie takich informacji jak F_{max} (maksymalna częstotliwość pracy systemu), *CN* (ang. „Core Number" – liczba rdzeni systemu), parametrów TF_i zadań oraz ich typu, algorytm ten przyporządkowuje je do odpowiednich rdzeni. Parametry te obliczane są według różnych metodologii i sposób ich uzyskania zostanie wyjaśniony w następnych podrozdziałach. Zadania są przydzielane w taki sposób, aby wszystkie rdzenie były możliwie równo obciążone.



Rys. 55. Podział zadań typu SHT

Pierwszym krokiem algorytmu BLTS jest utworzenie zbioru nazwanego ***TFs***, zawierającego parametry TF_i wszystkich harmonogramowanych zadań. Dla danych z Tab. 4 zbiór ***TFs*** zawierałby 60 elementów. Następnie wyznaczana jest liczba rdzeni (parametr *CN*) w zależności od wybranej metodologii. W kolejnym kroku następuje przyporządkowanie zadań typu SHT do rdzeni. W najbardziej optymalnym przypadku w każdym rdzeniu powinno znajdować się maksymalnie jedno zadanie SHT. Gdy zadań

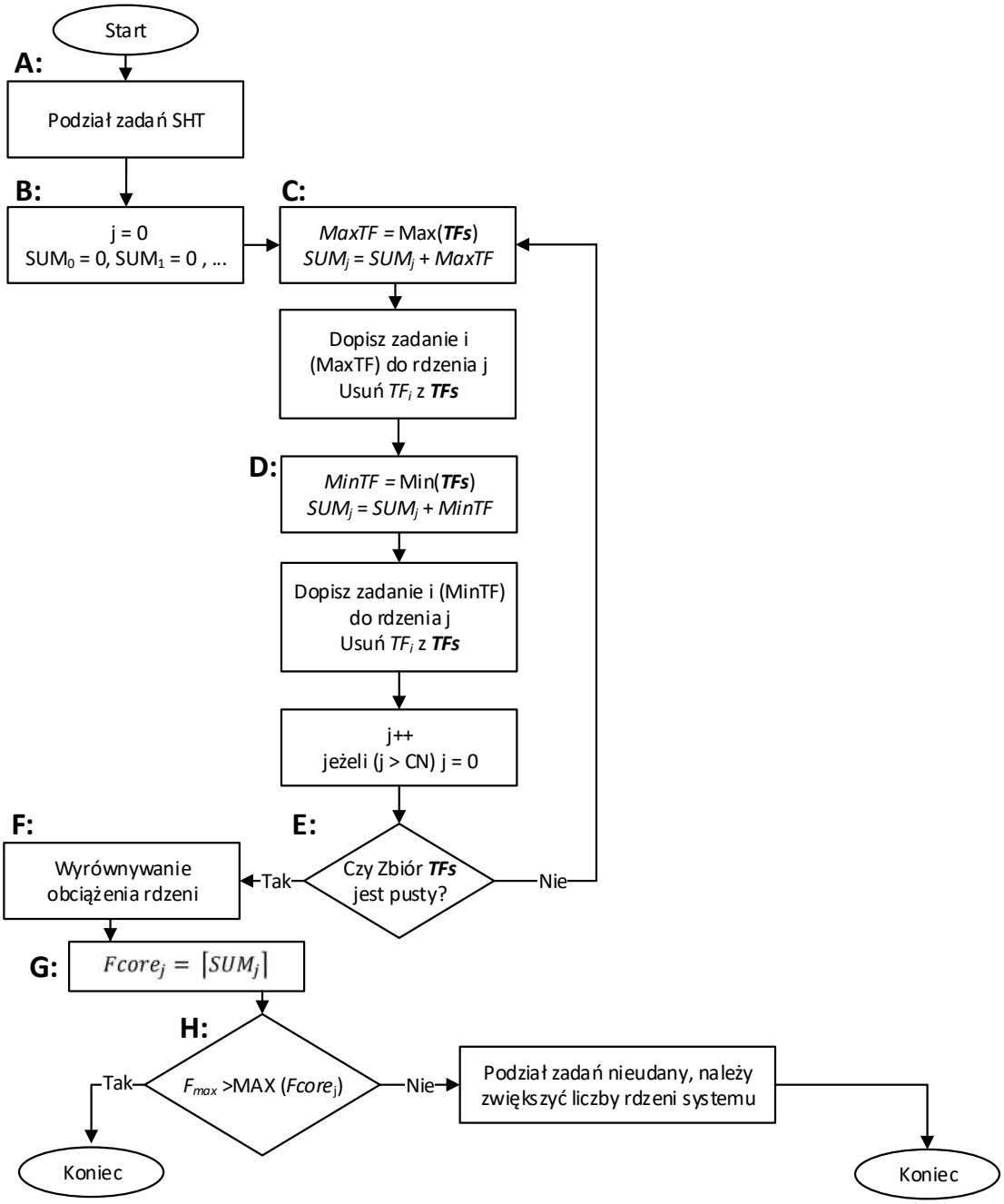
SHT jest więcej niż rdzeni (*CN*), należy tak dopasować i zwiększyć parametr TF_i zadań SHT, aby w każdym z rdzeni znajdowały się tylko wielokrotności TF_i zadań SHT lub aby wartości TF_i zadań SHT w rdzeniu były równe. Przykład takiego dopasowywania przedstawiono na Rys. 55. W przykładzie tym występują 4 zadania SHT natomiast system powinien być dwurdzeniowy. Niemożliwy jest więc przydział po jednym zadaniu SHT do rdzenia. Aby rozwiązać ten problem, TF_5 zostało tak zwiększone, żeby było połową TF_{21}, w ten sposób oba zadania SHT mogą pracować w jednym rdzeniu. Natomiast wartość TF_{34} została tak zwiększona, aby była równa TF_{50}. Powód takiego ograniczenia przydziału zadań SHT zostanie wyjaśniony w następnych podrozdziałach (Szeregowanie wątków precyzyjnych - SHT, strona 113).

Schemat ideowy algorytmu BLTS przedstawiono na Rys. 56. Po etapie podziału zadań SHT (Rys. 56A), następuje inicjalizacja zmiennych (Rys. 56B), zmienna *j* przyjmuje wartość 0 (będzie ona przechowywać numer aktualnie przetwarzanego rdzenia). Wszystkie zmienne SUM_j, (dla każdego rdzenia) również są zerowane. Będą one przechowywać sumę zadań TF_i poszczególnych rdzeni.

Następnie wyszukiwany jest element o maksymalnej wartości *TF* z zbioru ***TFs***, który zostaje zapisany do zmiennej *MaxTF* (Rys. 56C). Jednostką parametru *TF* jest liczba instrukcji w jednostce czasu jaką należy wykonać, by zadanie zostało wykonane na czas (4.2

Model zadania, strona 23). Przy przetwarzaniu potokowym wykonywanie każdej instrukcji kończy się co każdy cykl zegara. Dlatego w przybliżeniu można przyjąć, że rdzeń wykonuje jedną instrukcję na jeden cykl zegara. Suma *TFi* zadań pracujących w rdzeniu jest więc równoważna z minimalną częstotliwością z jaką musi pracować rdzeń, aby wszystkie zadania przez niego wykonywane zostały wykonane na czas. W kroku C dopisywane jest zadanie o maksymalnej wartości TF_i z listy ***TF***.

Na Rys. 56D, przedstawiono drugą część głównej pętli algorytmu. W części tej wyszukiwane jest zadanie o minimalnej wartości *TF*. Reszta algorytmu jest identyczna jak w punkcie B. Dodawanie do tego samego rdzenia na przemian zadania o maksymalnej i minimalnej wartości *TF*, pozwoli uzyskać zoptymalizowane pod względem obciążenia rdzenie.



Rys. 56. Algorytm BLTS

Kolejnym krokiem algorytmu (Rys. 56 F) jest ręczna optymalizacja obciążeń rdzeni, polegająca na wymianie zadań pomiędzy rdzeniami w taki sposób, aby suma parametrów TF_i zadań rdzeni była jak najbardziej zbliżona do siebie.

Na Rys. 56 H, obliczane są parametry $Fcore_j$. Są to zaokrąglone w górę do całości parametry SUM_j, czyli sumy parametrów zadań TF_i poszczególnych rdzeni systemu.

Ostatnim (Rys. 56 G) krokiem algorytmu jest sprawdzenie czy maksymalna wartość parametru SUM_j jest wyższa od maksymalnej częstotliwości pracy systemu F_{max}. Jeżeli tak jest, należy zwiększyć liczbę rdzeni systemu i uruchomić algorytm jeszcze raz.

6.3.1.1 Metodologia MINRES

Metodologia dążąca do minimalizacji zasobów powinna dążyć do minimalizacji liczby rdzeni systemu. Idealną sytuacją było by umieścić wszystkie harmonogramowane zadania w jednym rdzeniu. Niestety nie zawsze jest to możliwe, ponieważ istnieje maksymalna częstotliwość zegara systemowego, przy którym system działa poprawnie. Szeregowa praca zadań (podział na jak najmniejszą liczbę rdzeni) będzie wymagała zwiększenia mocy obliczeniowej systemu. Należy więc opracować taki podział zadań, aby rdzeni systemu było jak najmniej, ale teoretyczna częstotliwość pracy sytemu (F_{teo}), była mniejsza niż maksymalna (F_{max}). Dla aktualnej implementacji w używanym układzie Virtex-7 przyjęto F_{max} = 150 [MHz], wartość ta zawiera pewien margines bezpieczeństwa.

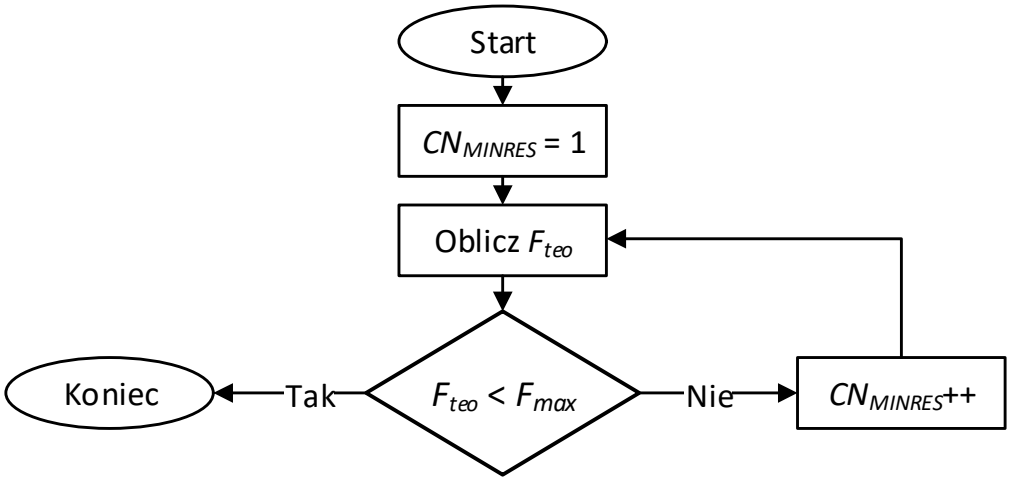
Aby uruchomić algorytm przydzielania zadań do rdzeni (Rys. 56) należy obliczyć wartości dwóch parametrów F_{teo} oraz CN_{MINRES}. Oba te parametry są zależne od siebie (wzór (11)). Należy więc wyznaczyć minimalne CN_{MINRES}, dla którego F_{teo} będzie mniejsze niż F_{max}.

$$F_{teo} = \left\lceil \frac{\sum_{i=1}^{N} TF_i}{CN_{MINRES}} \right\rceil \tag{11}$$

gdzie:

F_{teo}[MHz]	–	Teoretyczna częstotliwość pracy systemu
N	–	Liczba harmonogramowanych zadań
TF_i	–	Parametr *TF* zadania *i*
CN_{MINRES}	–	Liczba rdzeni systemu metodologii MINRES

Obliczana jest więc wartość F_{teo} dla minimalnego CN_{MINRES} równego 1. Gdy wartość ta jest za duża, następuje inkrementacja parametru CN_{MINRES}. Ideę algorytmu zaprezentowano na Rys. 57.



Rys. 57. Wyznaczanie parametru CN_{MINRES}

Dla przykładowego zestawu danych z Tab. 4 minimalne CN_{MINRES} wynosi 1 dla F_{teo} równego 88,23 [MHz]. Następnie dla tych parametrów został uruchomiony algorytm przydzielania zadań do rdzeni. Pierwszym etapem tego algorytmu jest podział zadań SHT. W konfiguracji tej jest tylko jeden rdzeń i cztery wątki typu SHT, więc wszystkie zadania typu SHT muszą pracować w jednym rdzeniu. Aby było to możliwe, zaktualizowano wartości TF_i zadań SHT wg Tab. 5.

Tab. 5. Podział zadań SHT - MINRES

ThID	**5**	**21**	**34**	**50**
Przydział	Rdzeń 0	Rdzeń 0	Rdzeń 0	Rdzeń 0
TF_i	1,092	2,468	0,886	0,925
Nowe TF_i	1,235	2,47	1,235	1,235

Po etapie podziału zadań SHT, wartość F_{teo} wzrosła do 89,06 [MHz]. W systemie pracuje tylko jeden rdzeń, nie ma więc potrzeby uruchamiania całego algorytmu podziału zadań. Wszystkie zadania pracują w jednym rdzeniu.

Wynikiem jego pracy jest podział zadań umieszczony w Tab. 6, parametry podziału to $Fcore_0$ = 90 [MHz] oraz CN_{MINRES} = 1;

Tab. 6. MINRES - podział zadań

Numer rdzenia	**Liczba zadań**	**ThID zadania**	**Suma TF_i**	**$Fcore_j$**
0	60	**21, 5, 34, 50,** 1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60	89,06	90

6.3.1.2 Metodologia MAXPRO

Metodologia ta powinna dążyć do maksymalizacji rdzeni systemu. Zadań w rdzeniu nie powinno być jednak zbyt mało, ponieważ znacznie ograniczy to możliwości szeregowania takich zadań. Po testach algorytmu szeregowania stwierdzono, że algorytm szeregowania jest mniej efektywny, gdy liczba zadań w rdzeniu jest mniejsza niż dwukrotność minimalnego przeplotu (parametr $Min_{indistance}$). Maksymalną liczbę rdzeni systemu można więc obliczyć za pomocą wzoru (12).

$$CN_{MAXPRO} = \left\lfloor \frac{N}{2 \cdot Min_{indistance}} \right\rfloor \tag{12}$$

gdzie*:*

CN_{MAXPRO} – Liczba rdzeni systemu w metodologii MAXPRO

N – Liczba harmonogramowanych zadań

$Min_{indistance}$ – Minimalny przelot systemu

Dla przykładowych zadań z Tab. 4, wartość parametru CN_{MAXPRO} wynosi 3 ($Min_{indistance}$ = 9). Ze wzoru (11) można również obliczyć parametr F_{teo}. Dla omawianych danych jego wartość wynosi 30 [MHz]. Po obliczeniu powyższych wartości można uruchomić algorytm przydzielania zadań do rdzeni (Rys. 56). Pierwszym jego etapem jest podział zadań SHT. W tej konfiguracji liczba rdzeni również jest mniejsza od liczby zadań SHT, w jednym rdzeniu muszą więc pracować dwa zadania SHT. Znaleziono więc dwa zadania o najmniejszej różnicy parametrów TF_i i wyrównano ich wartości poprzez zwiększenie mniejszego parametru. Podział zadań wynikający z tej operacji umieszczono w Tab. 7.

Tab. 7. Podział zadań SHT - MAXPRO

ThID	**5**	**21**	**34**	**50**
Przydział	Rdzeń 0	Rdzeń 1	Rdzeń 2	Rdzeń 2
TF_i	1,092	2,468	0,886	0,925
Nowe TF_i	1,092	2,468	0,925	0,925

Następnie uruchomiono drugą część algorytmu, która przydziela pozostałe zadania do rdzeni. Podział ten umieszczono w Tab. 8.

Tab. 8. MAXPRO - podział zadań

Numer rdzenia	**Liczba zadań**	**ThID zadania**	**Suma TF_i**
0	19	**5,** 58, 51, 60, 2 ,43, 17, 56, 57, 26, 57, 26, 35, 41, 18, 38, 12, 6, 37, 8, 30	27,56
1	19	**21,** 45, 10, 44, 13, 16, 39, 19, 4, 54, 40, 11, 52, 47, 1, 33, 23, 25, 48	30,23
2	22	**34, 50,** 3, 7, 46, 29, 49, 24, 32, 42, 53, 9, 31, 22, 20, 55, 15, 36, 28, 27, 59, 14	30,51

W kolejnym kroku uruchomiono proces wyrównywania obciążenia rdzeni. Zadanie 14 i 23 zostało przeniesione do rdzenia 0, natomiast zadanie 60 do rdzenia 2. Ostateczny podział przedstawiono w Tab. 9.

Tab. 9. MAXPRO - podział zadań po procesie wyrównywania obciążenia rdzeni

Numer rdzenia	**Liczba zadań**	**ThID zadania**	**Suma TF_i**	$Fcore_j$
0	22	**5,** 58, 51, ~~60~~, 2 ,43, 17, 56, 57, 26, 57, 26, 35, 41, 18, 38, 12, 6, 37, 8, 30, 14, 23	29,83	30
1	18	**21,** 45, 10, 44, 13, 16, 39, 19, 4, 54, 40, 11, 52, 47, 1, 33, ~~23~~, 25, 48	28,95	29
2	22	**34, 50,** 3, 7, 46, 29, 49, 24, 32, 42, 53, 9, 31, 22, 20, 55, 15, 36, 28, 27, 59, ~~14,~~ 60	29,51	30

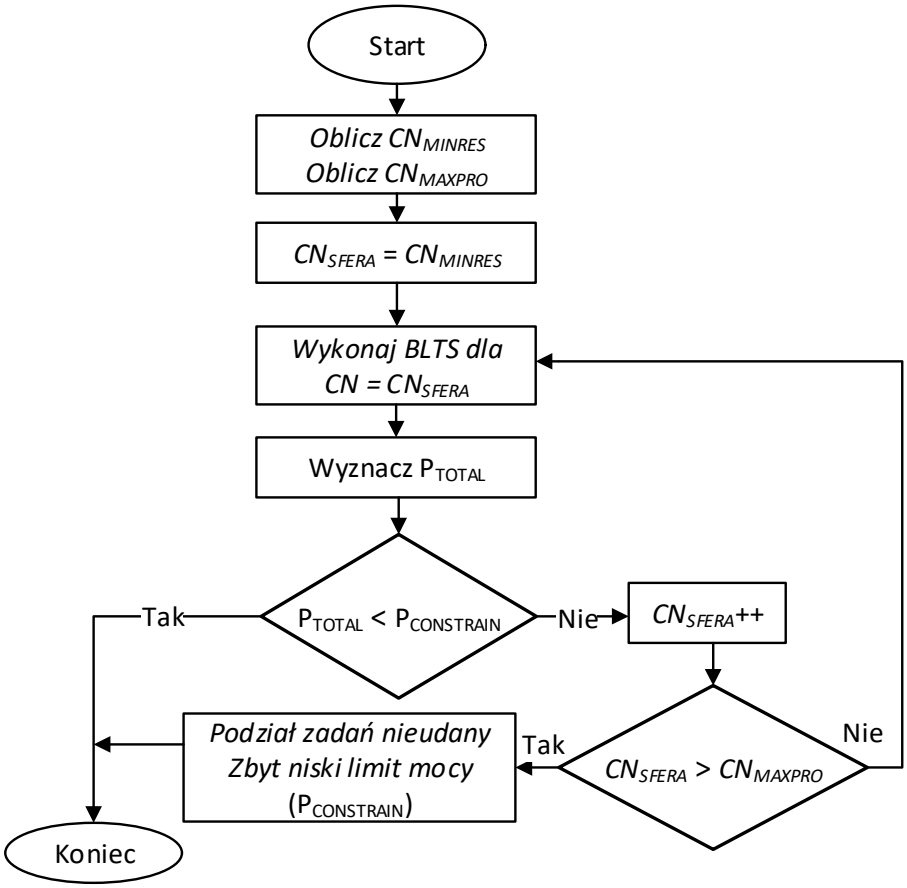
W ostatnim etapie sprawdzono czy minimalna liczba zadań w rdzeniu nie jest mniejsza niż dwukrotność $Min_{indistance}$ czyli wartości 18. W takiej okoliczności należałoby zmienić rozkład zadań lub zmniejszyć liczbę rdzeni systemu.

6.3.1.3 Metodologia SFERA

Metodologia SFERA pozwala na uzyskanie konfiguracji bardziej zbalansowanej niż w metodologiach MINRES oraz MAXPRO. Na początku zastosowania tej metodologii zakładany jest limit zapotrzebowania na moc systemu ($P_{CONSTRAIN}$). Zastosowany algorytm realizuje taki podział zadań, aby konfiguracja zużywała mniej mocy niż zakładane $P_{CONSTRAIN}$, ale wartość ta była jak najbliższa tej wartości. Schemat ideowy metodologii przedstawiono na Rys. 58. Algorytm ten polega na cyklicznym zwiększaniu liczby rdzeni systemu, aż zapotrzebowanie na moc będzie mniejsze niż założone. Wyznaczenie aktualnego zapotrzebowania na moc systemu (P_{TOTAL}) odbywa się przy użyciu narzędzi dostarczanych przez producenta układu FPGA.

Zakładając limit mocy całkowitej rozpraszanej w układzie ($P_{CONSTRAIN}$) na 1 [W], pętla algorytmu wykona się dwukrotnie. Dla $CN_{SFERA} = CN_{MINRES} = 1$, $P_{TOTAL} = 1{,}015$ [W]. Wartość ta jest zbyt duża, dlatego parametr CN_{SFERA} zostaje inkrementowany do 2.

Przy tym parametrze P_{TOTAL} jest równe 0,81 [W], więc mieści się w zakładanym limicie. Finalny podział zadań metodologii SFERA umieszczono w Tab. 10.



Rys. 58. Metodologia SFERA

Tab. 10. SFERA - podział zadań

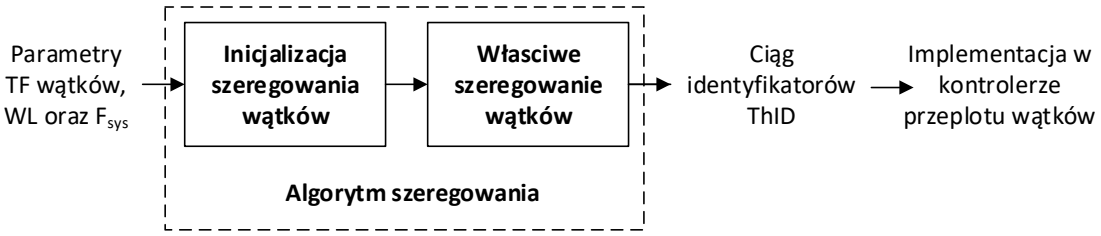
Numer rdzenia	**Liczba zadań**	**ThID zadania**	**Suma TF_i**	$Fcore_j$
0	29	**5, 21,** 46, 7, 51, 10, 44, 24, 32, 43, 17, 42, 19, 56, 31, 4, 35, 18, 41, 47, 52, 28, 36, 37, 6, 25, 23, 30, 14	44,22	45
1	31	**34, 50,** 20, 45, 29, 49, 60, 2, 13, 16, 3, 53, 39, 57, 9, 54, 26, 11, 40, 15, 55, 12, 38, 33, 1, 59, 27, 58, 8, 48, 22	44,23	45

6.4 Szeregowanie zadań

Szeregowanie zadań jest drugim etapem harmonogramowania. Polega ono na zaprojektowaniu takiego CP (Cykl Przeplotu - 4.3 Wykorzystanie przeplotu wątków strona 28), aby zadania przydzielone w etapie podziału zadań przetwarzane były z odpowiednia częstotliwością. Zadania o wyższym parametrze TF_i, muszą być przetwarzane częściej, jednak żadne zadanie nie może być przetwarzane szybciej niż co $Min_{indistance}$ (5.1.2.1 Ograniczenia przeplotu wątków, strona 43). W tym podrozdziale zostanie opisana metodologia szeregowania zadań.

6.4.1 Algorytm szeregowania zadań (wątków sprzętowych)

Algorytm uruchamiany jest przed implementacją oraz pracą systemu. Pozwala to na sprawdzenie, jeszcze przed jego uruchomieniem, czy zastosowana kolejność przeplatanych wątków w CP jest wystarczająca do zapewnienia przewidywalności czasowej systemu. Wynikiem pracy algorytmu jest cykl CP identyfikatorów ThID wątków. Cykl ten jest zapisywany pod postacią kodów ASCII w pliku tekstowym o rozszerzeniu „.il.txt". Posłuży on później do wygenerowania zawartości pamięci kontrolera przelotu wątków. Kontroler ten, według zawartości swojej pamięci, czyli kolejności identyfikatorów ThID wygenerowanych przez algorytm, będzie ustalał kolejność wykonywania zadań (Rys. 59). Aby nie doszło do sytuacji, w której wszystkie wątki z pamięci zostały już przetworzone, cała sekwencja identyfikatorów przechowywanych w CP jest zapętlana tak, aby długość pracy systemu nie była ograniczona długością sekwencji CP.

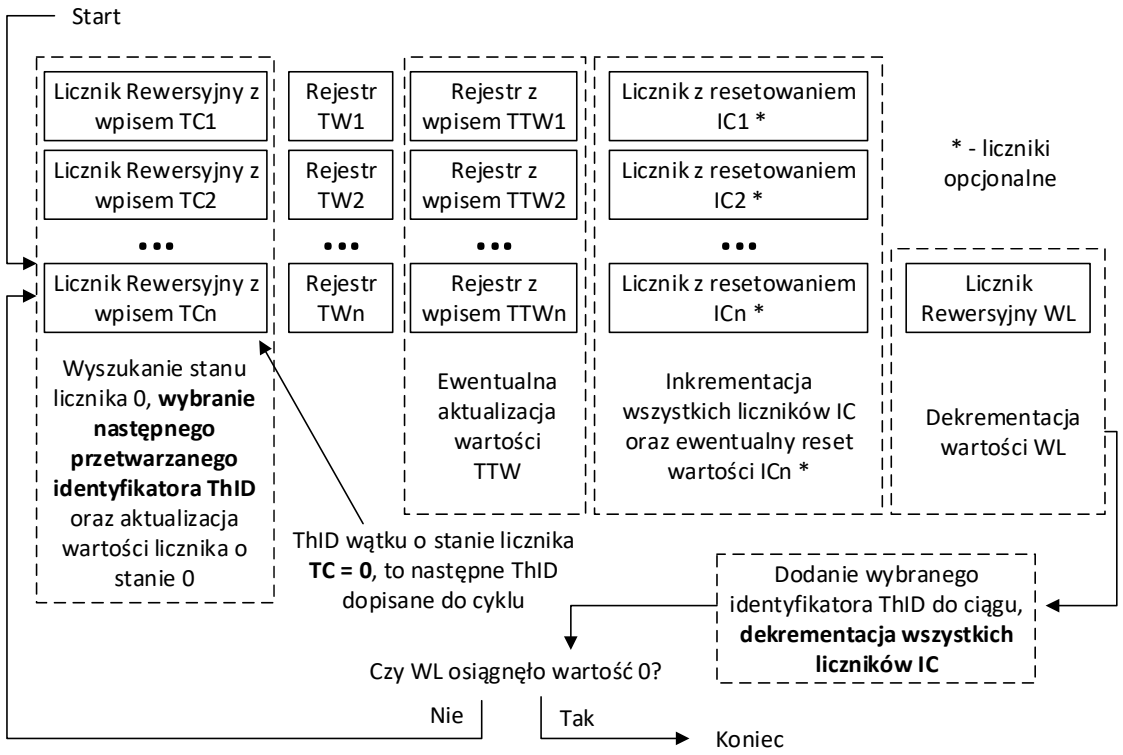


Rys. 59. Schemat metodologii szeregowania zadań

Każdy wątek sprzętowy obsługuje jedno zadanie, które charakteryzuje się parametrem TF. Kolejnym parametrem szeregowania jest WL („Window Length"),

według wartości którego, ustawiana jest długość generowanego cyklu CP (liczba identyfikatorów ThID w jednym cyklu). Niezbędnym parametrem do uruchomienia algorytmu jest jeszcze $Fcore_j$, czyli suma parametrów *TFi* zdań poszczególnych rdzeni (uzyskanych w poprzednim etapie podziału zadań). Jest to teoretyczna częstotliwość systemu, obliczona na etapie podziału zadań, dla której ma nastąpić szeregowanie zadań.

Szeregowanie zadań składa się z 2 etapów – inicjalizacji szeregowania wątków oraz właściwego szeregowania (Rys. 60). Inicjalizacja polega na obliczeniu i ustawieniu wstępnych parametrów algorytmu. Dokładniej zostanie to opisane w następnym podrozdziale.



Rys. 60. Schemat algorytmu szeregowania zadań

Szeregowanie właściwe polega na wypełnieniu liczników TC („Thread Counter") wartościami, które wskazują za ile cyklów zegara dany wątek ma być przetworzony, żeby spełnił swoje wymagania czasowe. Decyzja o tym, który wątek będzie przetwarzany przez system w następnym cyklu zegara, jest podejmowana na podstawie liczników TC. Gdy wartość któregoś z nich osiągnie 0, oznacza to, że taki wątek będzie przetwarzany w następnym cyklu zegara (Rys. 60) W każdym cyklu szeregowania liczniki TC są

dekrementowane. Dodatkowe rejestry oraz liczniki z Rys. 60 zostaną omówione w następnych podrozdziałach. Warto w tym miejscu zaznaczyć, że nowe wartości przypisywane są do liczników TC w taki sposób, aby nie doszło do sytuacji, w której dwa różne liczniki w tym samym cyklu posiadają tą samą wartość. Może się też zdarzyć, że w danym cyklu żaden z liczników nie osiągnął stanu zero, w takim wypadku do cyklu dopisywany jest wątek o ThID0. Adres ten oznacza, że w danym cyklu nie będzie przetwarzany żaden wątek. Następnie Wartość WL jest dekrementowana i cały cykl jest powtarzany, aż wartość licznika WL osiągnie wartość 0.

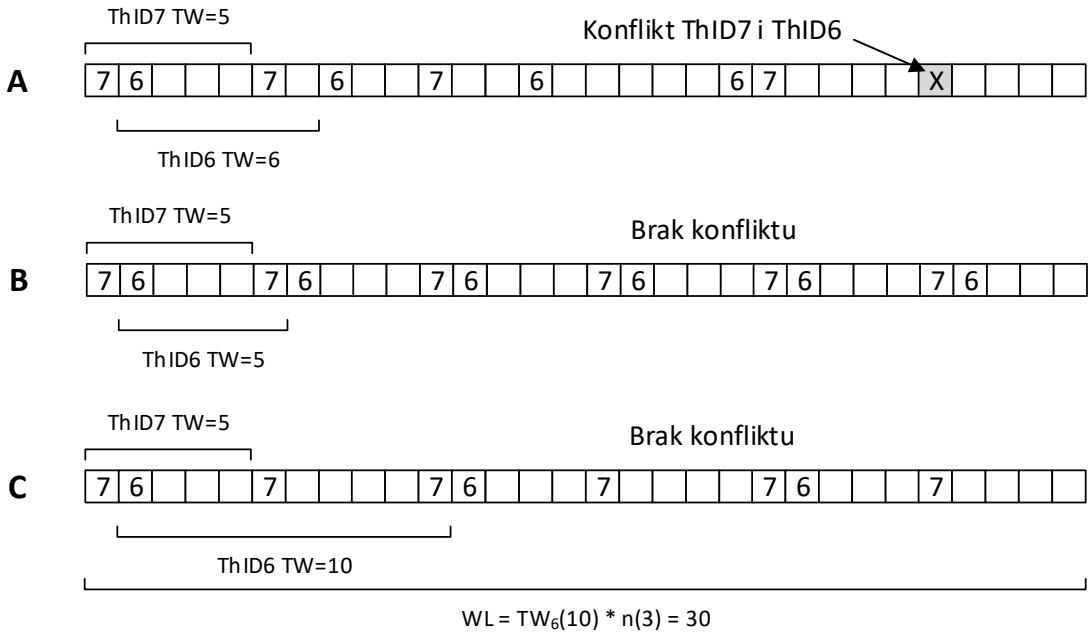
6.4.1.1 Wątki Precyzyjne - SHT

Wątki typu SHT (5.3.4.3 Typy obsługiwanych wątków strona 78) są to wątki sprzętowe, które wykonują instrukcje wymiany danych z innymi wątkami ze stałym czasem wykonania. Oznacza to, że w przypadku użycia wątków SHT eliminuje się wszystkie rozbieżności czasu wykonania zadania pochodzące z mikro architektury systemu. Wątki standardowe (TH) mogą posiadać odchylenie pomiędzy maksymalnym a minimalnym czasem wykonania, jeżeli maksymalny czas wykonania zadania jest mniejszy niż jego deadline (spowodowane jest to wykonywanie instrukcji wymiany danych). W przypadku szeregowania wątków SHT trzeba zadbać o stałą częstotliwość przetwarzania takiego wątku. Dzięki temu maksymalny i minimalny czas wykonania wątku SHT będzie taki sam.

Niestety szeregowanie wątków tego typu wiąże się z pewnymi ograniczeniami. Wątki typu SHT posiadają priorytet w szeregowaniu nad wątkami zwykłymi, ale nie można dopuścić do sytuacji, w której następuje konflikt w szeregowaniu dwóch wątków tego typu (SHT), ponieważ jeden z nich nie mógłby być wątkiem SHT. W kolejnym podrozdziale zostanie omówiony parametr TW („Time Window”), wyznaczający co ile taktów zegara dane zadanie musi zostać przetworzone, aby jego wymagania czasowe zostały spełnione. Aby nie dopuścić do takich konfliktów, parametry TW wątków typu SHT pracujących w jednym rdzeniu muszą być takie same lub muszą być wielokrotnością najmniejszego TW wątku typu SHT. TW obliczane jest liniowo za pomocą między innymi parametru TF, można więc stosować powyższe wymagania zamiennie do TF lub TW wątków.

Dzięki powyższemu założeniu odległości między identyfikatorami ThID wątków SHT w generowanym cyklu będą zawsze stałe, co nie pozwoli na konflikt wątków SHT

(Rys. 61B i C). W przeciwnym wypadku zawsze, przy odpowiednio długim cyklu, nastąpi taka sytuacja (Rys. 61A).



Rys. 61. Konflikty w szeregowaniu wątków SHT

Aby zapewnić stały czas wykonania wątków SHT nawet po *n*-tym wykonaniu cyklu CP, cykl musi mieć określoną długość (WL). W przypadku szeregowania wątków SHT, WL musi być równy lub być wielokrotnością największego parametru TW wątków SHT (Rys. 61B i C). Pozwoli to na wyeliminowanie konfliktów oraz spełnienie stałego przetwarzania co TW wątku SHT, nawet po zapętleniu cyklu CP (Po ostatnim identyfikatorze cyklu przetwarzany zostaje pierwszy identyfikator).

6.4.1.2 Inicjalizacja szeregowania wątków

Pierwszy etap szeregowania zadań - inicjalizacja, polega na obliczeniu wartości (stanów) początkowych liczników, rejestrów oraz obliczeniu wartości parametru *TW* („Time Window") charakteryzującego każde zadanie. Wartość *TW* jest obliczana według wzoru (13).

$$TW_i = \frac{Fcore_j - Fc_{marg}}{TF_i} \tag{13}$$

gdzie:

TW_i	–	Parametr *TW* zadania *i*
$Fcore_j$ [MHz]	–	Teoretyczna częstotliwość pracy systemu rdzenia *j*
Fc_{marg} [MHz]	–	Margines częstotliwości pracy szeregowania
TF_i	–	Parametr *TF* zadania *i*

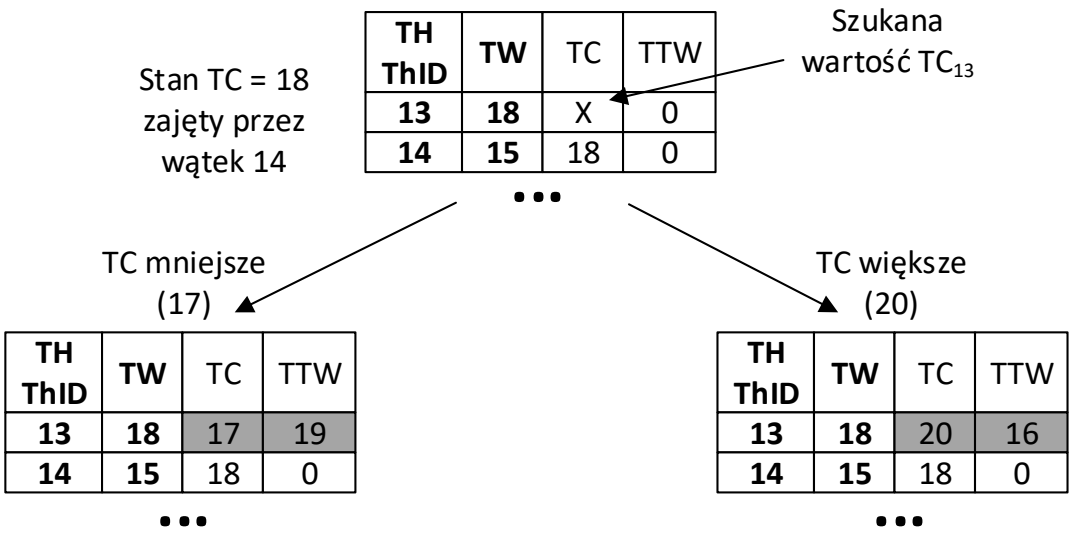
Wartość parametru TW oznacza wprost, co który takt zegara dane zadanie ma być przetwarzane, aby przy danej częstotliwości spełnić wymagania TF (czyli również nie przekroczyć czasu „deadline" zadania). Parametr ten nie może być również mniejszy niż minimalny odstęp przeplotu wątków w systemie ($Min_{indistance}$, wzór (5) strona 44), który odpowiada maksymalnej częstości powtarzania tego samego zadania (wątku) w potoku. Niespełnienie tego wymagania, czyli pojawienie się w potoku danego wątku częściej niż co $Min_{indistance}$ mogłoby doprowadzić do wystąpienia hazardów potoku [34]. Parametr Fc_{marg} jest marginesem częstotliwości pracy systemu, wcześniejsze eksperymenty pokazały, że najlepsze wyniki osiągane są przy wartości ok. 4-12[%] parametru $Fcore_j$[75]. Uruchomienie algorytmu szeregowania dla niższej częstotliwości pozwala na wyliczenie bardziej restrykcyjnych parametrów TW, co prowadzi do pewnego marginesu bezpieczeństwa. Parametr Fc_{marg} jest dobierany doświadczalnie. Powyższe założenie ogranicza minimalną częstotliwość zegara według wzoru (14):

$$F_{sys_{min}} = Min_{indistance} \cdot max(TF_i) + Fc_{marg} \tag{14}$$

gdzie:

$Min_{sys_{Freq}}$ [MHz]	–	Minimalna częstotliwość pracy systemu
$Min_{indistance}$	–	Minimalny przeplot systemu,
Fc_{marg} [MHz]	–	Margines częstotliwości szeregowania
TF_i	–	Parametr *TF* zadania *i*

Wartości TW_i posłużą do ustalenia początkowych stanów liczników TC (Rys. 60), czyli parametrów IV („Initial Value"). Intuicyjnie wydaje się, że najłatwiej można ten problem rozwiązać kopiując wartości TW_i zadań do odpowiednich liczników IV. Niestety wartości liczników TC w przeciwieństwie to parametrów TW_i nie mogą się powtarzać, a poza tym ich wartości powinny rozpoczynać się od 0. Aby rozstrzygnąć takie sytuacje wprowadzono nowy rejestr TTW („Temporary TW") dla każdego szeregowanego zadania. Jeżeli podczas szeregowania zdarzy się sytuacja, w której dane zadanie nie może być wykonane z częstością co TW cykli (w tym takcie może być już zaplanowane przetwarzanie innego zadania), to takie zadanie szeregowane jest wcześniej lub później niż TW, a TTW przyjmuje wartość różną od zera. Taką sytuację przedstawiono na Rys. 62.

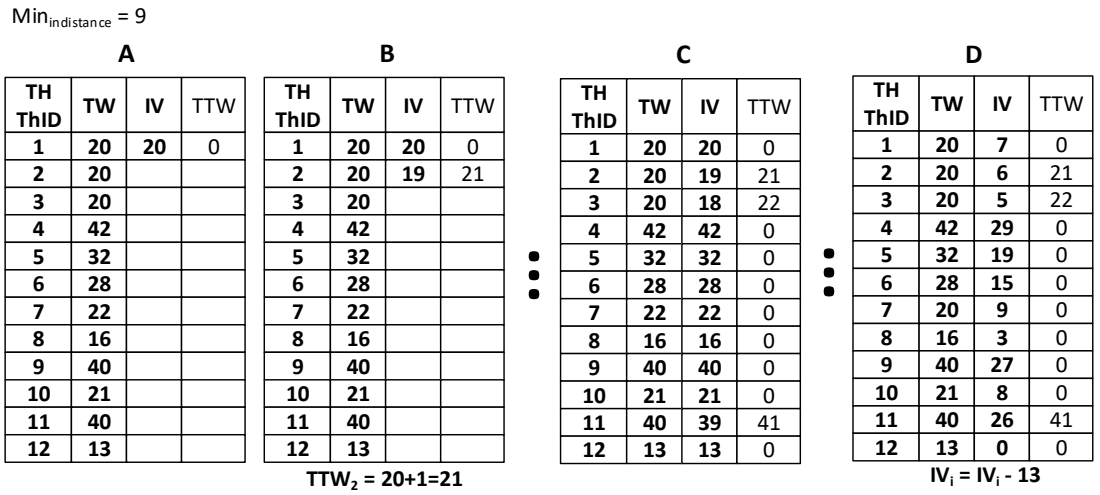


Rys. 62. Brak wolnego stanu TC

Wartość licznika TC równa 18 jest już zajęta przez wątek 14 (ThID 14). Możliwe jest więc tylko wpisanie wartości większej lub mniejszej niż 18 (wartość ta też musi być wolna). W przypadku wpisania wartości mniejszej, np. 17 TTW osiągnie wartość większą niż TW – 19. Podczas następnej iteracji wyliczania TC wątku 13, pod uwagę będzie brana wartość TTW zamiast TW. Jeżeli jedna z instrukcji wątku 13 została wykonana szybciej niż powinna, następna instrukcja tego wątku może pojawić się później w sekwencji, tzn. może zostać wykonana później. TTW jest więc wartością zapamiętującą odchylenia od standardowej wartości TW. Jeżeli wartość TTW jest nieużywana jej wartość wynosi 0.

Dokładny opis obliczania wartości TTW znajduje się w następnym podrozdziale (wzór (15)). Analogicznie, w przypadku wybrania większej wartości licznika (na Rys. 62, wartość 20), wartość TTW będzie mniejsza niż TW. Jeżeli instrukcja danego wątku została wykonana później niż powinna, to kolejna instrukcja zadania obsługiwanego przez ten wątek powinna zostać wykonana wcześniej.

Sposób wyznaczenia wartości początkowych (startowych) IV, TW oraz TTW przedstawiono na Rys. 63.



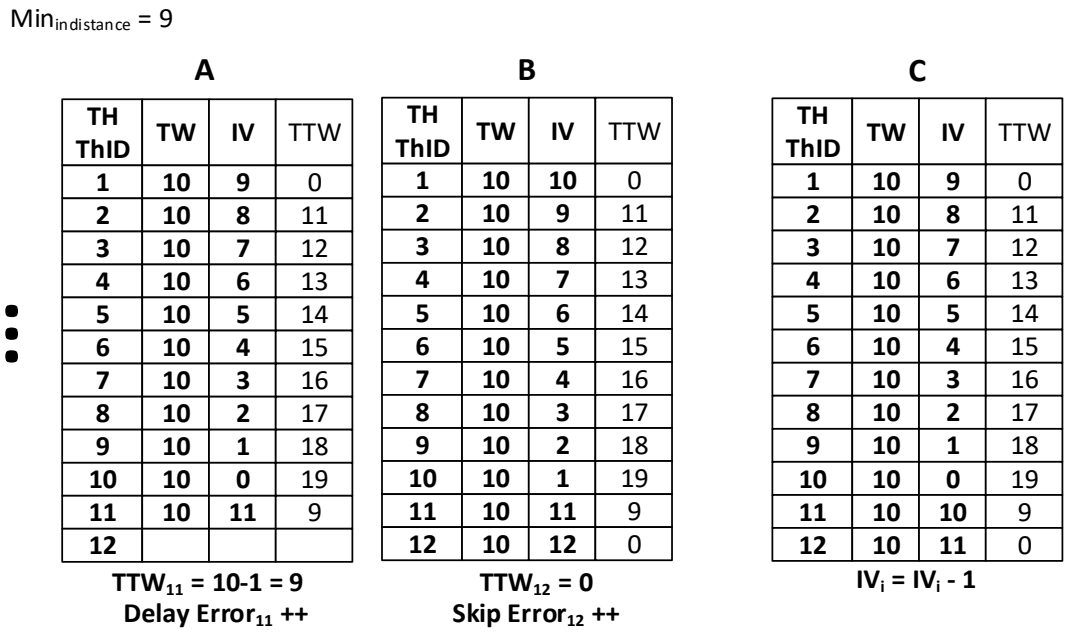
Rys. 63. Inicjalizacja szeregowania, przypadek standardowy

Przypisywanie rozpoczyna się od pierwszego zadania na liście (ThID 1). Wartość IV równa 20 jest wolna, więc następuje przypisanie stanu 20 (zawartość TW_1) do licznika IV_1 (Rys. 63A). TW_1 pasowało do IV_1 bez zmian, dlatego do TTW_1 zostaje przypisane zero. Następnie przetwarzane jest kolejne zadanie z listy (ThID 2), w tym wypadku wartość TW_2(20) jest już zajęta przez IV_1, dlatego szukana jest najbliższa niższa wartość – 19, która zostaje wpisana do IV_2. TTW_2 przyjmuje wartość TW_2, powiększoną o różnicę między TW_2 a IV_2, czyli 21 (Rys. 63B). Stan wszystkich wartości inicjalizujących pokazano na Rys. 63C. Mechanizm ten zostanie dokładniej opisany w następnym podrozdziale.

Podczas szeregowania może nastąpić również taka sytuacja jak na Rys. 64A, gdy nie ma niższego wolnego stanu IV. W takim wypadku (ThID 11), znajdowana jest najbliższa wyższa wartość (Rys. 62). TTW_{11} przyjmuje wartość TW_{11} pomniejszonego

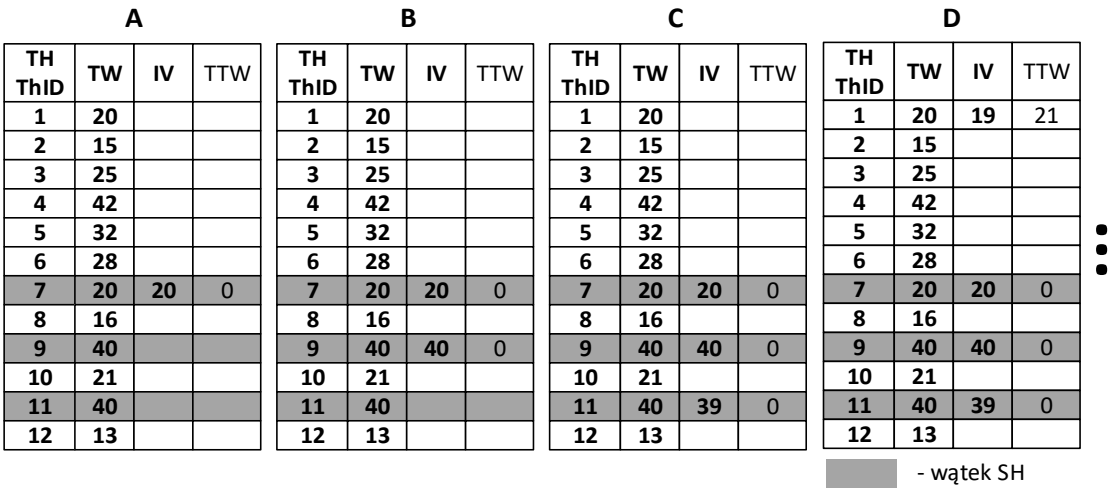
o różnicę między TW_{11} a IV_{11}, czyli 9. W takim przypadku inkrementowany jest również licznik błędów konkretnego wątku „Delay $Error_{11}$", ponieważ nie było możliwości prawidłowego uszeregowania zadań. Zadanie numer 11 występuje co najmniej raz w CP (cykl przeplotu), czyli rzadziej niż co minimalna liczba taktów zapisana w TW_{11}.

Na rysunku Rys. 64B przedstawiono zdarzenie, w którym wyliczone TTW_{12} (10-2=8) jest mniejsze niż $Min_{indistance}$ (9). W takiej sytuacji informacja o TTW_{12} jest tracona (zostaje jej przypisana wartość 0), a licznik błędów „Skip $Error_{12}$" wątku 12 zostaje inkrementowany. Mechanizm ten zostanie dokładniej opisany w następnym podrozdziale.



Rys. 64. Inicjalizacja szeregowania, brak wolnego stanu TC

W przypadku inicjalizacji szeregowania zadań, w których są zadania typu SHT (Wątki precyzyjne), ustalanie parametrów IV rozpoczyna się od pierwszego zadania SHT z listy (ThID7, Rys. 65A). Należy pamiętać, że TW wątków typu SHT pracujących w jednym rdzeniu musza spełniać określone ograniczenia: muszą być sobie równe lub być wielokrotnością najmniejszego TW wątku SHT (Zostało to wyjaśnione w poprzednim rozdziale, 6.4.1.1 Wątki Precyzyjne - SHT strona 103).



Rys. 65. Inicjalizacja szeregowania, wątki SHT

Następnie obliczane jest IV dla kolejnego zadania SHT z listy (ThID9, Rys. 65B) i kolejne (ThID11, Rys. 65C). W tym wypadku wartość ta jest jednak zajęta, zostaje więc znaleziona najbliższa niższa wartość 39. W momencie określania parametrów startowych wątków SHT nie są obliczane i przypisywane wartości TTW. Zadania te, są wykonywane dokładnie co TW, więc TTW nie jest potrzebne. Różnica w stanie IV wpływa tylko na przesunięcie wykonania, a nie jego długość. Po ustaleniu parametrów IV wątków SHT, następuje ustalenie parametrów pozostałych wątków (ThID1, Rys. 65D).

Ostatnim etapem inicjalizacji szeregowania jest przesunięcie stanów IV tak, aby najniższa wartość wynosiła 0. Dzięki tej zmianie, któryś z liczników po inicjalizacji będzie na pewno zawierał zero, czyli na pewno będzie przetwarzany jakiś wątek. W przeciwnym wypadku mogło by się zdarzyć, że jeden lub więcej początkowych adresów ThID w cyklu CP będą zerami. Operacja to odbywa się poprzez odjęcie najmniejszego stanu licznika od wszystkich pozostałych liczników. W przypadku Rys. 63D, od każdego IV odjęta by została wartość 13, a w przypadku Rys. 64C została odjęta wartość 1.

Parametry TW, IV, TTW oraz WL są przekazywane do następnego etapu właściwego szeregowania, gdzie inicjalizują początkowe stany liczników.

6.4.1.3 Szeregowanie właściwe

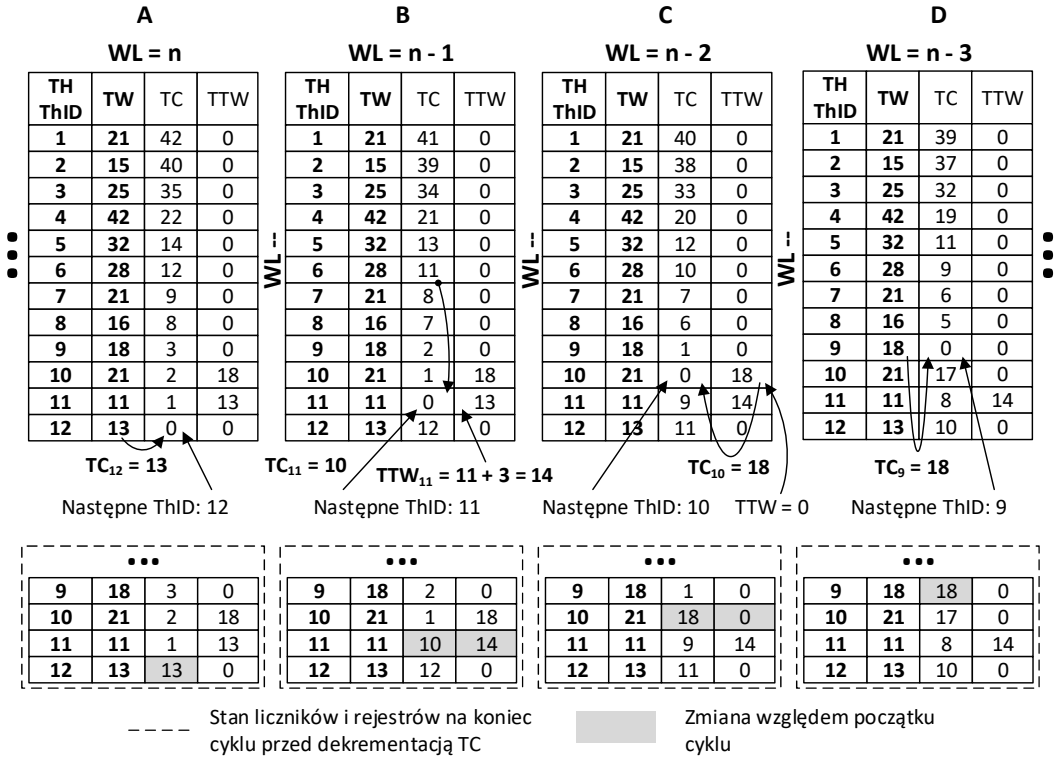
Szeregowanie zadań rozpoczyna się od wypełnienia liczników i rejestrów wartościami startowymi obliczonymi w poprzednim podrozdziale. Następnie

poszukiwany jest licznik TC, który zawiera wartość zero. Wątek, którego licznik TC, osiągnął wartość zero, będzie przetwarzany jako następny (zostanie dopisany do generowanego cyklu). Jednocześnie wyznaczane są nowe parametry rejestrów TC oraz TTW tego wątku. Jest to najbardziej złożona część szeregowania. W kolejnym kroku następuje dekrementacja wszystkich liczników TC oraz licznika WL i cała procedura jest powtarzana dla kolejnego zadania, którego licznik osiągnął wartość zero (bądź wstawiany jest cykl jałowy – wstawienie tzw. „bąbelka" z ang. bubble). Proces szeregowania kończy się w momencie, gdy cykl osiągnie długość równą parametrowi WL (wówczas stan licznika WL osiągnie wartość zero) (Rys. 60).

Podczas wykonywania operacji wyznaczania nowej wartości TC licznika wątku, podobnie jak podczas obliczania parametrów startowych, mogą nastąpić trzy sytuacje. Pierwszy przypadek (najprostszy) występuje wtedy, gdy szukany nowy stan licznika TC jest wolny, tzn. nowej wartości licznika danego wątku nie zawiera żaden z innych liczników TC pozostałych wątków. Druga sytuacja polega na tym, że co prawda nowy stan licznika jest już zajęty przez licznik TC innego wątku, ale dostępna jest niższa wartość TC. Do licznika TC tego wątku wpisujemy nową wartość niższą od wartości poszukiwanej. Najbardziej skomplikowany jest trzeci przypadek, kiedy nowy stan licznika TC danego wątku jest zajęty przez inny wątek i dodatkowo nie ma niższej wolnej wartości licznika TC. W takiej sytuacji szukana jest nowa, wyższa od proponowanej, wartość licznika TC (wyższa wartość zawsze może być znaleziona).

Pierwsza sytuacja została przedstawiona na Rys. 66A. Licznik TC_{12} wątku 12 (ThID 12) osiągnął wartość zero, a zatem wątek 12 jest następnym przetwarzanym wątkiem. Nowa wartość TC_{12} tego wątku obliczana jest na podstawie parametrów TW_{12} lub TTW_{12}. Jeżeli TTW_{12} jest różne od 0 do kolejnych obliczeń używana jest wartość TTW_{12}, w przeciwnym wypadku TW_{12}. W omawianej sytuacji wartość TTW_{12} jest równa zero, dlatego do dalszych obliczeń wykorzystywana jest wartość TW_{12}. W następnym kroku sprawdzane jest więc, czy któryś z liczników innych wątków (TC_1 do TC_{11}) zawiera stan 13 (TW_{12}). Stan ten jest wolny, więc do TC_{12} kopiowana jest wartość TW_{12} (13). Identyczny scenariusz przedstawiono na Rys. 66D. Na Rys. 66C również występuje podobna sytuacja, różnica polega na tym, że kolejny przeplatany wątek 10 posiada wartość TTW_{10} różną od zera. Sprawdzane jest więc, czy stan 18 (TTW_{10}) jest wolny (żaden licznik TC nie ma wartości 18), co jest prawdą. Następnie do IC_{10} jest przepisywana wartość TTW_{10} (18). Ostatnim krokiem algorytmu, jest usunięcie używanej

wartość TTW_{10}, czyli wpisanie do niej wartości 0. Przez to, że wartość TTW_{10} pasowała do TC bez zmian (nie zawierał jej żaden inny licznik), niepotrzebna jest już żadna wartość tymczasowa i można wrócić do szeregowania według TW_{10}.



Rys. 66. Szeregowanie właściwe - przykład 1

Druga, trudniejsza sytuacja została pokazana na Rys. 66B. Następnym przetwarzanym wątkiem jest wątek 11, ale stan 13 (TTW_{11} różne od zera) jest zajęty przez licznik TC_5. Stan 12 również jest zajęty przez TC_{12}, a 11 przez TC_6. Najbliższą niższą wartością jest więc 10(F_{TC}) i taką wartość przyjmie licznik TC_{11}. Aktualna wartość TTW, uwzględniająca różnice w czasie wystąpienia wątku (wątek wystąpi trzy cykle wcześniej niż powinien) obliczana jest według wzoru (15).

Ostatni (trzeci), najbardziej złożony przypadek został przedstawiony na Rys. 67. Jeżeli nie jest możliwe znalezienie niższego stanu licznika TC, znajdowana jest najbliższa wyższa wartość. Niestety powoduje to, że odstęp między przetwarzanymi instrukcjami takiego wątku będzie większy niż TW, co może naruszyć wymagania czasowe takiego zadania, mimo, że wartość średnia TW będzie poprawna. W specyficznych przypadkach (np. gdy będzie to ostatnia instrukcja zadania) błąd taki może wywołać naruszenie

ograniczeń czasowych. W takim przypadku, by nie naruszyć ograniczeń czasowych zadania, minimalna częstotliwość pracy zegara systemu będzie musiała być większa niż założona częstotliwość pracy rdzenia ($Fcore_j$). Dlatego zdarzenie takie jest zgłaszane poprzez inkrementacje błędów „Delay Error$_i$". Każde z zadań posiada swój indywidualny licznik błędów, aby umożliwić rozróżnienie zadań w których występują takie błędy.

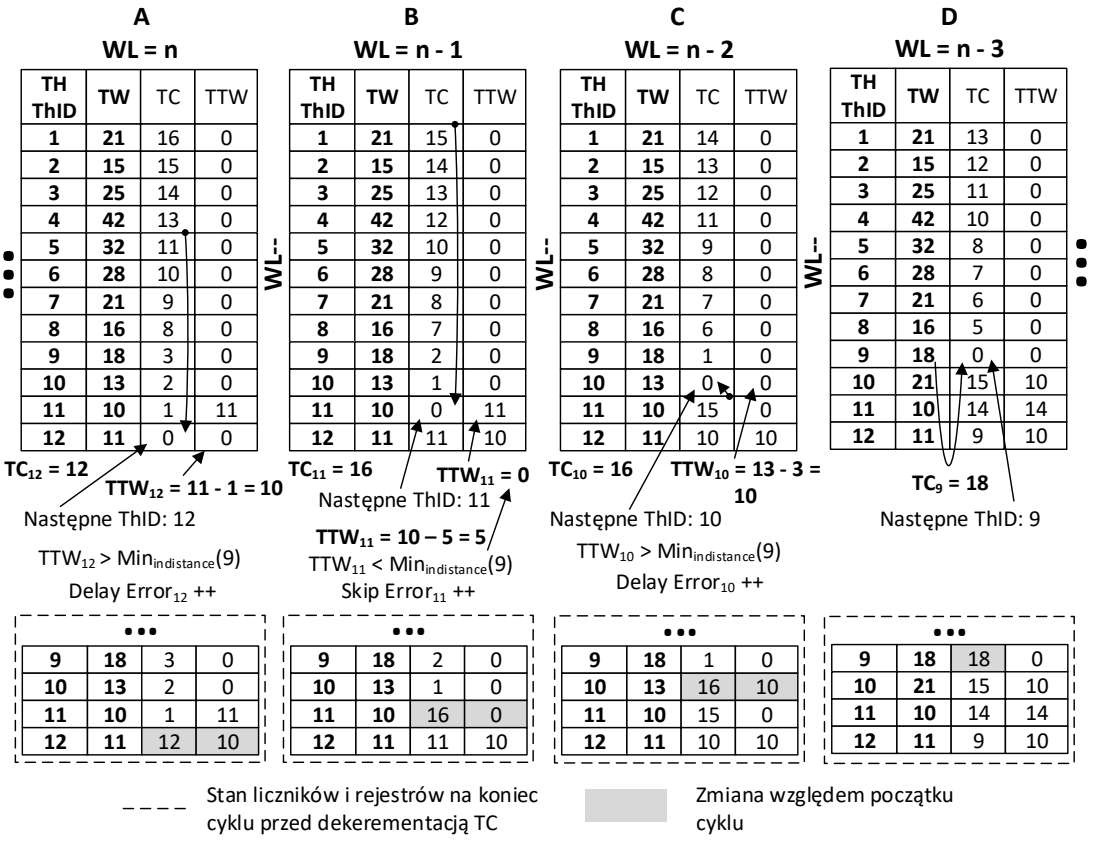
$$NEW_{TTW} = TW + \left((OLD_{TTW}\ lub\ TW) - F_{TC}\right) \tag{15}$$

gdzie:

NEW_{TTW} – Nowa wartość TTW

F_{TC} – Znaleziona nowa wolna wartość licznika TC

$OLD_{TTW}\ lub\ TW$ – Poprzednia wartość TTW gdy była różna od 0, w przeciwnym wypadku TW



Rys. 67. Szeregowanie właściwe - przykład 2

Omawiana sytuacja została przedstawiona na Rys. 67A. Stan licznika TC_{12} osiągnął stan 0, czyli będzie on przetwarzany w następnym takcie zegara (następuje dopisanie wątku o ThID12 do cyklu). Wartość TTW_{12} wynosi zero, więc szukany jest wolny stan licznika TC równy 11 (TW_{12}). Najbliższa niższa wartość licznika to 7. Niestety wartość ta jest niższa niż minimalny przelot ($Min_{indistance}$ (9)), wpisanie tej wartości do licznika TC_{12} spowodowałoby, że po upływie kolejnych 6 cykli zegara instrukcja 12 wątku zostałaby pobrana do wykonania w potoku. Taka sytuacja mogłaby doprowadzić do wystąpienie hazardu w procesorze potokowym (naruszenie minimalnego odstępu w przeplocie wątków [34]).

Wobec tego do licznika TC wpisana zostanie najbliższa wyższa wartość, czyli 12 (F_{TC}). Aktualna wartość $TTW_{12}(10)$ obliczana jest również według wzoru (15). W tym wypadku jednak wartość ta (10) będzie mniejsza niż TW_{12} (11).

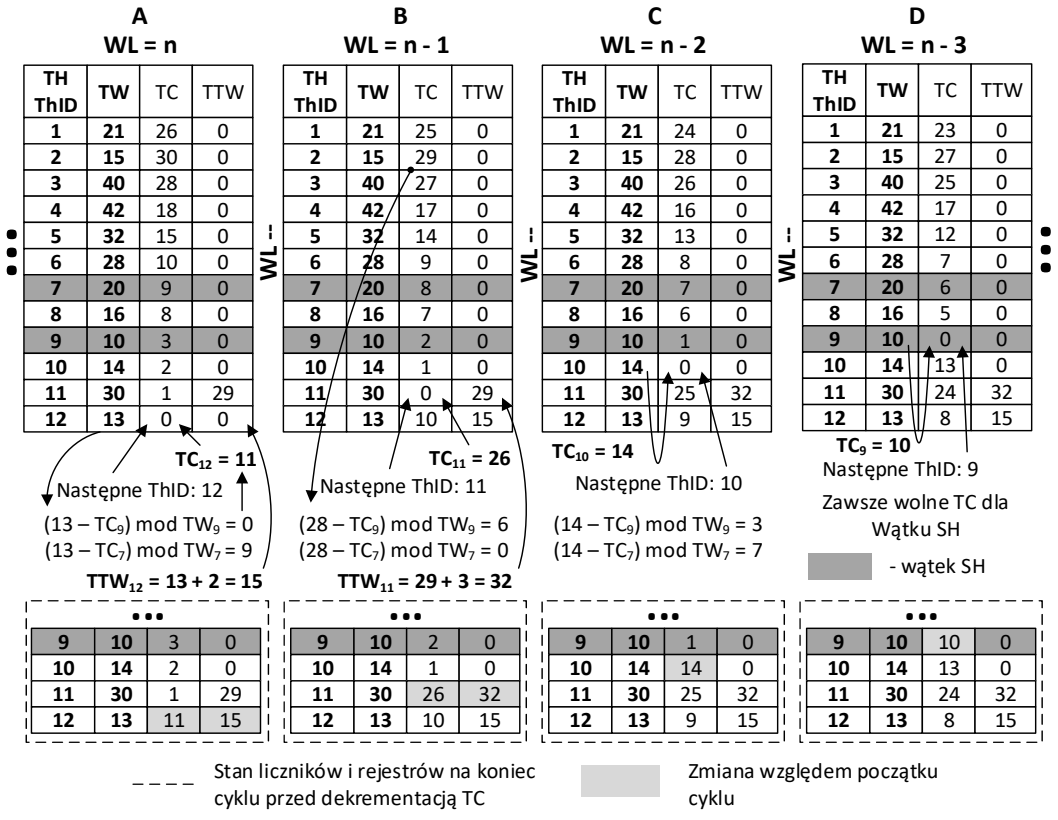
W kolejnym cyklu szeregowania (Rys. 67B) sytuacja jest jeszcze bardziej skomplikowana. W tym cyklu dopisany do CP będzie wątek o ThID11. Do licznika TC_{11} powinniśmy wpisać wartość 11 (TTW_{11} jest różne od zera i ma wartość 11). Niestety wartość ta jest zajęta, a najbliższa niższa wartość to 6, która jest mniejsza niż minimalny przelot ($Min_{indistance}$ (9)). Teraz mamy do czynienia z podobnym przypadkiem jak poprzednio. Najbliższa wyższa dostępna (wolna) wartość wynosi 16 (F_{TC}) i taka wartość zostanie zapisana do TC_{11}. Aktualna wartość TTW_{11} obliczona ze wzoru (15) wynosi 5. Wartość ta nie może być wpisana jako TTW_{11}, ponieważ jest mniejsza niż minimalny przelot ($Th_{min}(9)$). Wpisanie wartości 5 do TTW_{11} spowodowałoby szukanie następnej (za 16 cykli zegara, gdy wartość TC_{11} osiągnie 0) wartości TC_{11} wynoszącej 5. To mogłoby doprowadzić do dwukrotnego przetworzenie wątku o ThID11 w cyklu 5 cykli zegara. Dlatego informacja o stanie TTW_{11} jest tracona - w jej miejsce zostało wpisane 0. Następnie zostaje zgłoszony błąd „Skip $Error_i$" oznaczający wymuszenie usunięcia wartości TTW (odchylenia od TW) przez wątek 11. Błąd ten jest bardziej krytyczny niż „Delay Error", ponieważ nawet średnie TW nie będzie równe zakładanemu TW.

6.4.1.4 Szeregowanie wątków precyzyjnych - SHT

Wątki typu SHT według swoich założeń muszą być przetwarzane dokładnie według swojego parametru TW. Niedopuszczalne jest tu żadne przesunięcie w znalezieniu nowej wartości licznika TC, zarówno w dół jak i w górę. Należy pamiętać, że żeby to było

możliwe, TW wątków typu SHT pracujących w jednym rdzeniu musza spełniać określone ograniczenia (6.4.1.1 Wątki Precyzyjne - SHT strona 103).

Do zapewnienia stałego TW wątków SHT sprawdzany jest dodatkowy warunek. Warunek ten jest sprawdzany tylko dla wątków „zwykłych" gdy szukana jest nowa wartość licznika TC (po tym jak wartość TC osiągnęła 0). Wątki SHT posiadają priorytet przed wątkami HT (standardowymi) dlatego sprawdzane jest czy znaleziona wartość nie będzie w następnych iteracjach zajęta przez wątek typu SHT. Jeżeli jest to prawdą, szukana jest następna inna wartość (zgodnie z rozdziałem 6.4.1.3) tak, aby zostawić miejsce dla wątku SHT. Doprowadza to do sytuacji, w której podczas szukania nowej wartość licznika TC dla wątku SHT zawsze jest wolny stan licznika TC równy TW (dla wątku SHT). Wątek typu SHT będzie więc dopisywany do cyklu dokładnie co swoje TW. Powyższy typ szeregowania zostanie omówiony na Rys. 68.



Rys. 68. Szeregowanie wątków SHT – przykład

Na Rys. 68A następnym przetwarzanym wątkiem jest wątek 12 (ThID12). Następuje więc szukanie następnej wartości TC_{12}. Stan licznika 13 (TW_{12}) jest wolny, ale

sprawdzany jest również dodatkowy warunek dla każdego wątku SHT w rdzeniu. Wartość TC_{free}, ze wzoru (16), traktowana jest jak warunek logiczny. Jeżeli jest równa 0, oznacza to, że ta wartość TC w przyszłych iteracjach zostanie zajęta przez wątek typu SHT. Jeżeli jej wartość jest większa od zera, oznacza to, że wartość ta jest wolna.

$$TC_{free} = (F_{TC} - TC_h) \, mod \, TW_h \tag{16}$$

gdzie:

TC_{free} – Wartość logiczna wskazująca czy stan licznika TC jest wolny

F_{TC} – Znaleziona wartość licznika TC

TC_h – Aktualna wartość TC wątku hard

TW_h – Wartość TW wątku hard

Dla wątku 9 wartość TC_{free} jest równa zero (dla F_{TC} = 13), dlatego szukana jest następna dostępna wartość według algorytmów z poprzedniego podrozdziału. Nową wartością F_{TC} jest więc 11 i w tym wypadku dla obu wątków SHT wartość TC_{free} jest różna od 0. Stan TC_{11} wynosi więc 11, a TTW_{12} obliczone za pomocą wzoru (16) - 15.

Na Rys. 68B pokazano podobną sytuację, główną różnicą jest to, że następny przetwarzany wątek (ThID11) posiada wartość TTW różną od zera. Następuje szukanie nowej wartości TC_{11}, począwszy od wartości 29 (TTW_{11}). Niestety wartość ta jest zajęta przez TC_2. Szukana jest więc niższa wartość – 28(F_{TC}). Teraz sprawdzany jest dodatkowy warunek, TC_{free} dla wątku SHT7(ThID7) jest równe 0, więc wartość ta będzie zajęta. Szukany jest więc kolejny stan licznika (według algorytmów z poprzedniego podrozdziału) – 27(F_{TC}). Niestety jest on zajęty przez TC_3. Kolejna wartość F_{TC} to 26, wartość ta nie jest zajęta przez żaden z liczników. Sprawdzany jest więc dodatkowy warunek, TC_{free} dla wątku SHT7(ThID7) i SHT9(ThID9) jest różne od zera. Można więc już uaktualnić stan licznika TC_{11} wartością 26. Następnie wyliczana jest nowa wartość TTW_{11}, jest ona równa 32.

Na Rys. 68B pokazano najbardziej standardowy przypadek. Następnym przetwarzanym wątkiem jest wątek 10. Szukaną wartością licznika TC_{10} jest 14 (wartość TW, TTW jest równe 0). Wartość ta jest wolna, więc sprawdzany jest dodatkowy

warunek. Dla obu wątków SHT wartość TC_{free} jest różna od 0 (dla F_{TC} = 14). Do TC_{10} zostaje więc przypisana wartość 14. TTW nie przyjmuje żadnej wartości, ponieważ znaleziona wartość licznika TC nie wymagała zmiany.

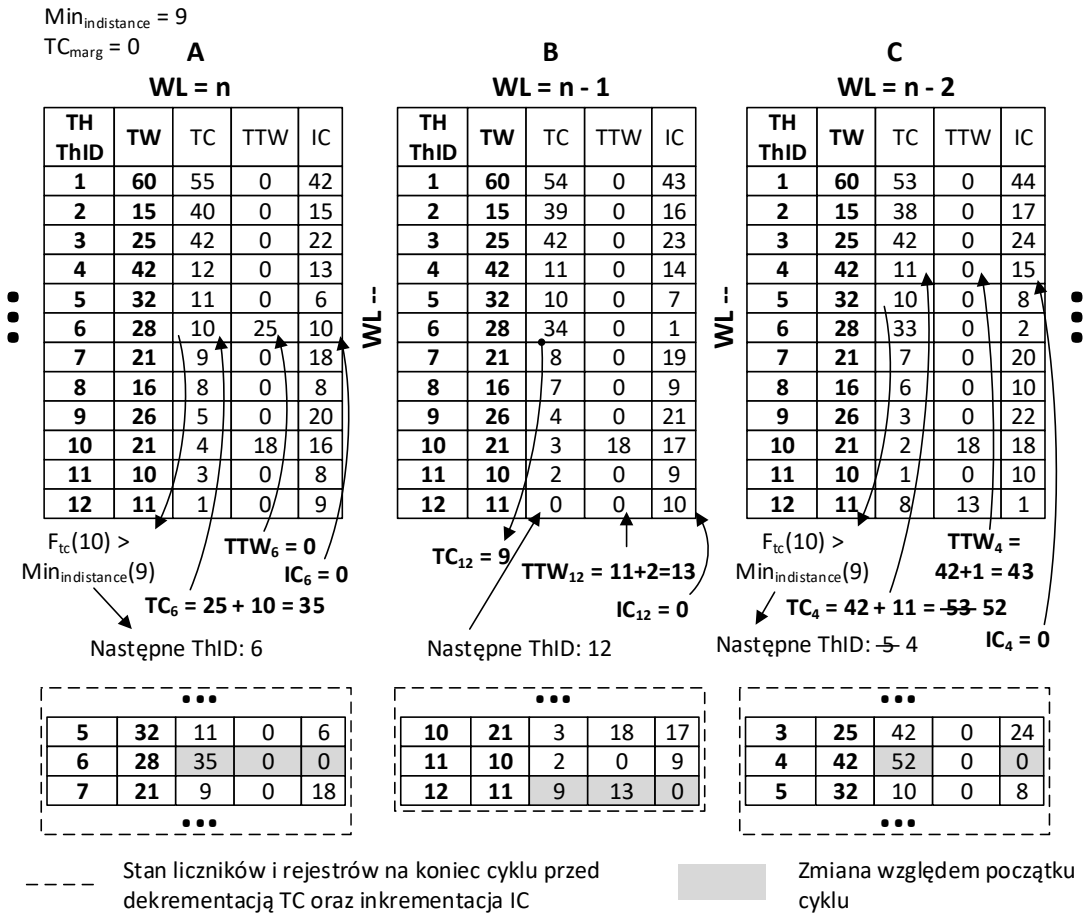
Ostatni rysunek (Rys. 68D), przedstawia sytuację, w której licznik TC_9 wątku SHT osiągnął wartość 0. Standardowo szukana jest nowa wartość TC_9 począwszy od wartości $TW_9(10)$ (TTW równe zero). Wartość ta jest wolna, więc do TC_9 zostaje wpisana wartość 10. Szukana wartość została przyjęta bez zmian, dlatego wartość TTW_9 również wynosi 0. Podczas omawiania tego rysunku, należy zwrócić uwagę na to, że jeżeli 3 cykle wcześnie przy WL = n (Rys. 68A), dodatkowy warunek TC_{free} nie byłby sprawdzony i wartość TC nie zostałaby skorygowana, w aktualnym cyklu stan licznika 10 byłby zajęty. Uniemożliwiłoby to zszeregowanie wątku SHT. Metoda ta również zapewnia, że TTW wątku SHT, zawsze będzie równe 0.

6.4.1.5 Eliminacja jałowych cykli potoku

W sytuacji, gdy w danym takcie zegara żaden z liczników nie osiągnął wartości zero, następnym przetwarzanym wątkiem jest wątek 0. Identyfikator ten (ThID0) oznacza, że system nie wykonuje żadnego zadania (nie wykonuje instrukcji żadnego wątku), czyli wchodzi w tzw. cykl jałowy. Algorytm przedstawiony w tym podrozdziale, pozwoli zminimalizować takie zdarzenia kosztem szybszego przetwarzania niektórych wątków. Oznacza to, że omawiany algorytm może spowodować większą różnicę między minimalnym, a maksymalnym czasem wykonaniem standardowych wątków. Dlatego ta część algorytmu jest opcjonalna. Jeżeli następnym dopisanym wątkiem do cyklu ma być wątek 0, algorytm ten sprawdza czy jest możliwość podmienienia tego adresu na inny. Jeżeli tak, dopisuje go aktualizując odpowiednio jego wartości TC oraz TTW.

Wątek sprzętowy nie może być przetwarzany częściej niż wartość $Min_{indistance}$, dlatego trzeba przechowywać dodatkową informację o tym, ile cykli wstecz dany wątek był przetwarzany. Do tego celu zaprojektowano dodatkowe liczniki „IC" („Interval Counter"). Liczniki te w przeciwieństwie do TC liczą w górę. Gdy wartość licznika TC spadnie do zera (odpowiadający mu wątek jest przetwarzany), wartość odpowiadającego mu licznika IC również jest zerowana. W ten sposób w każdym liczniku IC przechowywana jest informacja, ile cykli temu został przetwarzany odpowiadający mu wątek.

Gdy żaden z liczników TC nie zawiera wartości zero (sytuacja przedstawiona Rys. 69A), omawiany algorytm wyszukuje minimalny stan licznika TC, który jest większy niż $Min_{indistance}$ (9) (wzór (17)). Znaczenie parametru TC_{marg}, zostanie wyjaśnione później.



Rys. 69. Usuwanie cykli jałowych – przykład

$$TC_{min} = Th_{min} + TC_{marg} \quad (17)$$

gdzie:

TC_{min} – Minimalna wartość licznika TC

$Min_{indistance}$ – Minimalny przeplot systemu

TC_{marg} – Margines w szukaniu minimalnej wartości TC

Znaleziona wartość TC to 10(F_{TC}) dla wątku 6. Następnie sprawdzane jest czy IC_6(10) wątku 6 jest większe bądź równe $Min_{indistance}$ (9), co jest prawdą. Zagwarantuje to, że wątek 6 nie będzie przetwarzany szybciej niż jest w stanie robić to system. Ostatnią rzeczą, którą należy sprawdzić jest to, czy wątek 6 jest wątkiem typu SHT. Kolejność przetwarzania wątków SHT nie jest zmieniana przez ten algorytm. Wątek ten (ThID6) nie jest wątkiem SHT, więc następnym przetwarzanym wątkiem zostaje właśnie on (wątek 6), a jego wartość IC_6 zostaje wyzerowana. Kolejnym krokiem jest wyznaczenie nowych parametrów TC oraz TTW wątku 6. Nowa wartość TC musi uwzględniać ile cykli wcześniej dany wątek został dopisany do CP, bo tyle cykli później może zostać wykonana jego następna instrukcja. Wartość ta jest obliczana za pomocą wzoru (18) i wynosi 35. Wartość ta nie jest zajęta przez żaden inny licznik, dlatego nie jest szukana żadna mniejsza ani większa wartość licznika. TTW_6 może zostać wyzerowane, a wartość 35 zastaje zapisana w TC_6.

$$NEW_{TC} = (TTW\ lub\ TW) + OLD_{TC} \tag{18}$$

gdzie:

NEW_{TC} – Nowa wartość TC

OLD_{TC} – Poprzednia wartość TC

$TTW\ lub\ TW$ – Wartość *TTW* gdy była różna od 0, w przeciwnym wypadku *TW*

Na Rys. 69B pokazano kolejny standardowy cykl szeregowania. Stan licznika TC_{12} osiągnął wartość zero, więc następnym przetwarzanym wątkiem jest wątek o adresie ThID12. Wartość IC_{12} zostaje wyzerowana i szukany jest następny stan licznika TC_{12}. Stan 11 (TW_{12}) jest zajęty i najbliższą wolną, niższą wartością jest stan 9(F_{TC}). Jest to stan licznika, który został zwolniony w poprzednim cyklu (Rys. 69A). Wartość ta zostaje więc zapisana do TC_{12}(9) i zostaje obliczona aktualna wartość TTW_{12} (wzór(15)). Wartość ta wynosi 13 i jest różna od zera, ponieważ znaleziona wartość licznika TC_{12}(9) jest różna od szukanej (11). Należy tutaj zwrócić uwagę, że gdyby w poprzednim cyklu zegara nie była szukana wartość TC większa niż TC_{min}, nie byłoby teraz wolnego stanu licznika TC mniejszego niż szukana wartość (stan licznika 9). Spowodowane byłoby to

tym, że uzyskany w poprzednim cyklu pusty stan licznika miałby wartość mniejszą niż $Min_{indistance}$. Wątek, który osiągnął w aktualnym cyklu wartość 0 (ThID12), nie mógłby więc zająć tej wartości (Powodowało by to zbyt szybkie wykonanie instrukcji wątku 12, mogące powodować hazardy). Szukany byłby więc stan licznika większy niż 11(TW_{12}) co spowodowałoby inkrementację błędu „Delay" lub nawet „Skip Error".

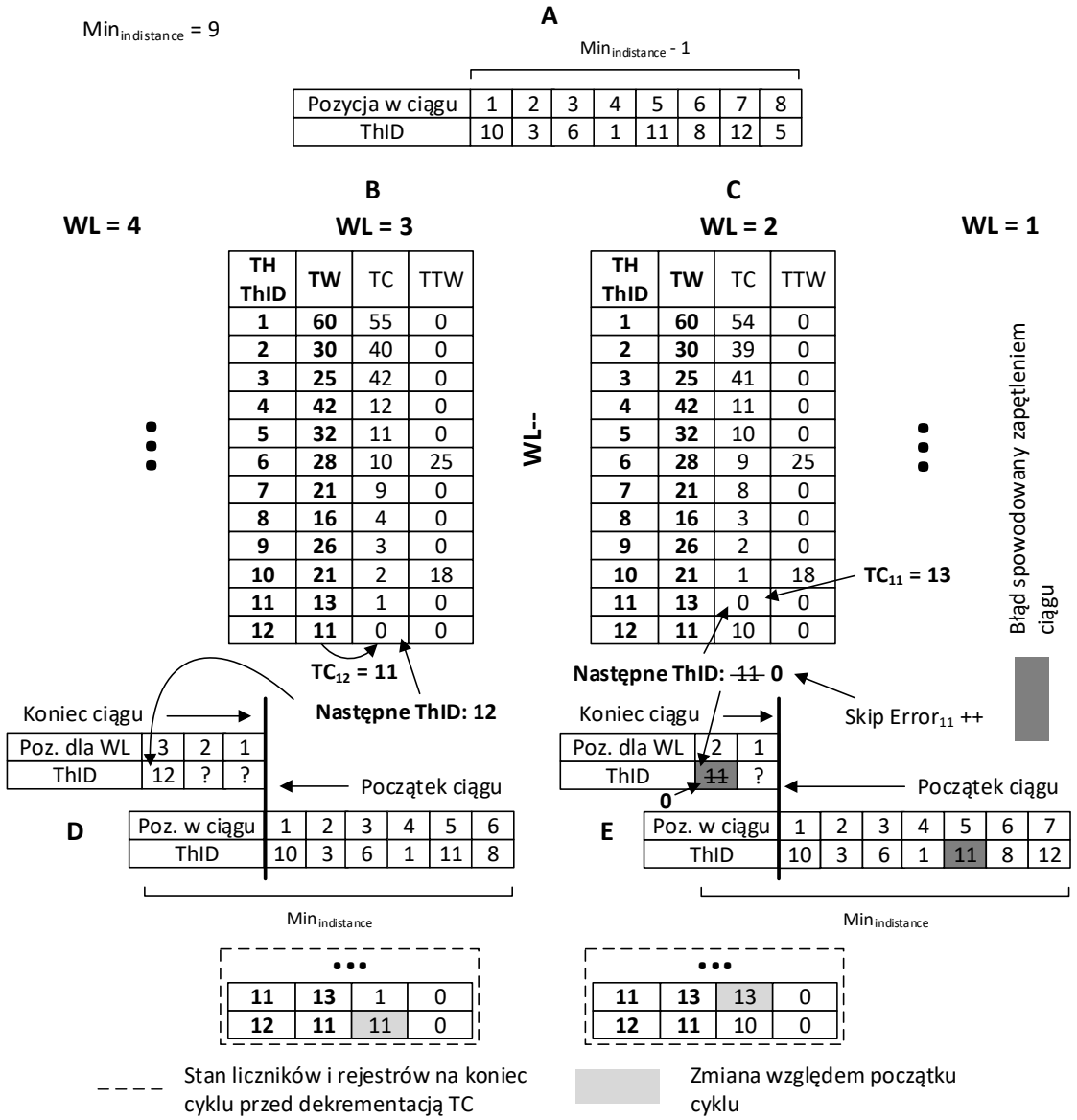
Parametr TC_{marg} ze wzoru (17) dobierany jest doświadczalnie i jego zwiększanie spowoduje zwiększenie prawdopodobieństwa, że w kolejnych taktach zegara zostanie wykorzystana, stworzona przez algorytm, pusta wartość TC. Niestety zwiększenie go powoduje również coraz większa wartość NEW_{TC}, co powoduje zwiększenie różnicy między maksymalnym, a minimalnym czasem wykonania zadania. Standardowo ten parametr posiada wartość 0.

Na Rys. 69C przedstawiono kolejny przypadek cyklu, w którym żaden z liczników nie osiągnął stanu 0. Szukana jest więc najniższa wartość licznika TC większa niż TC_{min} (9). Znaleziona wartość to 10 (TC_9, F_{TC}). Niestety wartość IC_5(8) jest niższa niż $Min_{indistance}$(9). Nie można więc użyć tego wątku i dopisać go do CP w tym momencie, ponieważ zostałby przetworzony dwukrotnie w ciągu 8(IC_5) cykli zegara. Szukana jest zatem następna wartość – 11 (TC_4, F_{TC}). Wartość IC_4(15) jest większa niż $Min_{indistance}$(9), więc spełnia warunek minimalnego przelotu. Następnym przetwarzanym wątkiem zostaje więc wątek 4 (ThID4). Następnie obliczana jest nowa wartość TC_4 za pomocą wzoru (18). Niestety wartość ta (53) jest już zajęta przez TC_1. Następuje więc standardowe szukanie nowej wartości licznika z poprzednich podrozdziałów. Najpierw szukana jest niższa wolna wartość niż 53. Znaleziona wolna wartość to 52, zostaje ona zapisana do TC_4. Następnie obliczana jest wartość TTW_4 (TTW jest niezbędne, ponieważ znaleziona wartość licznika TC_4(52) jest różna niż szukana (52)) za pomocą wzoru (15). Nowa wartość TTW_4 wynosi 43.

6.4.1.6 Usuwanie błędów powstałych po zapętleniu zszeregowanego cyklu

Ostatnim etapem szeregowania jest sprawdzenie czy w wygenerowanym cyklu CP identyfikatorów ThID, nie następuje częstsze przetwarzanie wątków niż $Min_{indistance}$, spowodowane zapętleniem tego cyklu. W docelowym zaimplementowanym sprzęcie, wygenerowany cykl będzie powtarzany (po ostatnim elemencie cyklu, będzie przetwarzany pierwszy element cyklu), więc może ten problem nastąpić. Aby zapobiec takiej sytuacji zapamiętywanych jest pierwszych ($Min_{indistance}$ – 1) identyfikatorów

z cyklu (Rys. 70A i Rys. 71A). Gdy podczas szeregowania wartość WL (liczba pozostałych przebiegów do końca szeregowania) spada poniżej $Min_{indistance}$, oznacza to, że są to ostatnie elementy cyklu, narażone na omawiany błąd. Przy wartości $Min_{indistance}$ równej 9 są to cykle dla wartości WL od 8 do 1 (cykl dla wartości WL = 0, nie występuje – Rys. 60 strona 102).



Rys. 70. Usuwanie błędów powstałych po zapętleniu cyklu – przykład 1

Możliwe są dwa warianty omawianego algorytmu:

- Wariant 1 – wersja podstawowa z wyłączoną korekcją błędów
- Wariant 2 – wersja rozszerzona z korekcją błędów w brzegowych segmentach sekwencji

W przypadku wariantu 1 wyłączona jest funkcja usuwania przebiegów nieprzetwarzających żadnych wątków (podrozdział 6.4.1.5). W takim przypadku błędy będą zastępowane po prostu adresami ThID0 (cyklami jałowymi potoku procesora). Algorytm ten przedstawiono na Rys. 70. Gdy wartość licznika WL spada poniżej $Min_{indistance}$ (np. WL jest równe 3, Rys. 70B) i wyznaczony jest następny przetwarzany wątek (ThID12 – TC_{12} jest równe 0), sprawdzany jest dodatkowy warunek. Kontrolowane jest czy po dodaniu do końca cyklu wyznaczonego wcześniej identyfikatora (ThID12) oraz identyfikatorów z jego początku (zapamiętanych na Rys. 71A), w oknie równym $Min_{indistance}$, identyfikator ThID któregoś z wątków się powtarza (Rys. 70D). W omawianym cyklu, żaden z identyfikatorów się nie powtarza, szeregowanie jest więc przeprowadzane według poprzednich podrozdziałów.

W następnym cyklu (Rys. 70C) wartość licznika TC_{11} osiągnęła wartość 0. Następnym przetwarzanym wątkiem jest więc wątek o identyfikatorze ThID11. Na Rys. 70E sprawdzany jest dodatkowy warunek. W tym wypadku, po zapętleniu cyklu, wątek o identyfikatorze ThID11 występuje częściej niż $Min_{indistance}$ (2 razy wciągu 7 cykli). Ponieważ taka sytuacja może spowodować wystąpienie hazardu w potoku, więc identyfikator ThID11 zamieniany jest na ThID0. Spowoduje to naruszenie wymagań czasowych wątku 11, więc jest inkrementowany błąd „Skip $Error_{11}$”. Dodatkowo w cyklu tym nie będzie przetwarzana przez system żadna instrukcja. Parametry TC_{11} oraz ewentualne TTW_{11} obliczane są według algorytmów z poprzednich podrozdziałów.

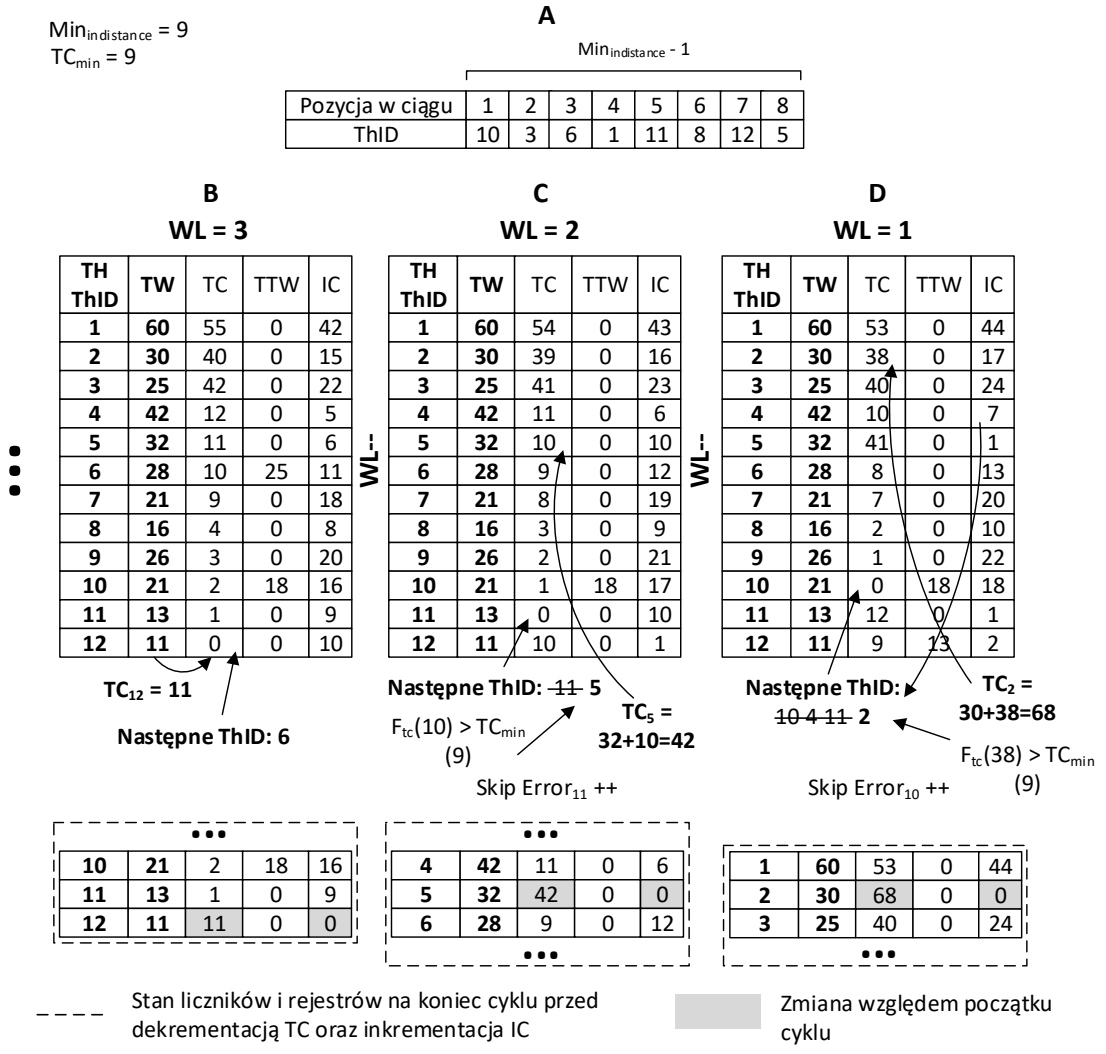
Drugi wariant algorytmu jest bardziej skomplikowany i uruchamiany jest, gdy włączona jest funkcja usuwania przebiegów nieprzetwarzających żadnych wątków (podrozdział 6.4.1.5). Do uruchamiania tego algorytmu będą potrzebne dodatkowe liczniki IC używane w poprzednim podrozdziale. Proces szeregowania zaczyna się standardowo. Dla wartości WL mniejszych od $Min_{indistance}$ szukany jest licznik TC z wartością 0. Na Rys. 71B licznik TC_{12} osiągnął wartość 0, więc wątek o identyfikatorze 12 powinien być następnym wątkiem dopisanym do cyklu. Teraz jest sprawdzany dodatkowy warunek, tj. czy dopisanie tego identyfikatora do cyklu, spowoduje błąd

powstały na skutek zapętlenia. Jest to sprawdzane dokładnie tak samo jak na Rys. 70D (przykład jest taki sam). W tym cyklu błąd ten nie występuje, więc cykl ten jest cyklem standardowym szeregowanym według algorytmów z poprzednich rozdziałów.

W kolejnym cyklu (Rys. 71C), następnym przetwarzanym wątkiem powinien być wątek o identyfikatorze ThID11 ($TC_{11} = 0$). Sprawdzany jest więc dodatkowy warunek błędu spowodowanego zapętleniem. Przykład jest taki sam jak na Rys. 70E. Dopisanie więc wątku 11 spowoduje błąd. Następuje inkrementacja błędu „Skip $Error_{11}$", ponieważ wątek ten nie może być wykonywany w tym cyklu. Zamiast dopisać do cyklu CP identyfikator ThID0, używana jest podobna metoda jak ta, którą zastosowano do usuwania przebiegów nieprzetwarzających żadnych wątków (podrozdział 6.4.1.5). Początkowo algorytm ten wygląda tak samo, szukana jest minimalna wartość TC, większa niż $TC_{min}(9)$ (obliczona na podstawie wzoru (17)). Wartość tą ($10(F_{TC})$) posiada licznik TC_5. Następnie sprawdzane jest czy wartość licznika $IC_5(10)$ jest większa niż $Min_{indistance}(9)$. Jest to prawdą, więc dopisanie wątku ThID5 do cyklu nie spowoduje błędu spowodowanego zbyt częstym jego przetwarzaniem. Kolejny krok jest krokiem dodatkowym względem poprzedniego algorytmu (z podrozdziału 6.4.1.5). Sprawdzane jest czy dopisanie wątku ThID5 do cyklu nie spowoduje błędu spowodowanego zapętleniem cyklu. W zapamiętanym wektorze (Rys. 71A), na pozycjach od 1 do 7 nie występuje wątek ThID5, więc do takiej sytuacji nie dochodzi. Wątek o identyfikatorze ThID5 może więc zostać dopisany do cyklu. Następnie obliczana jest nowa wartość $TC_5(42)$ według wzoru (18). Zgodnie z poprzednimi algorytmami wartość TTW nie jest obliczana, ponieważ nie jest potrzebna.

W ostatnim cyklu (Rys. 71D) następnym przetwarzanym wątkiem powinien być wątek ThID10 ($TC_{10} = 0$). Sprawdzany jest teraz dodatkowy warunek, sprawdzający czy nie dojdzie do błędu spowodowanego zapętleniem cyklu ($WL < Min_{indistance}$). Na pozycjach od 1 do 8 w zapamiętanym wektorze (Rys. 71A) znajduje się wątek ThID10 (pozycja 1), więc dochodzi do tego błędu. Następuje zinkrementowanie błędu „Skip $Error_{10}$". Teraz szukana jest najmniejsza wartość TC większa od $TC_{min}(9)$, jest to $10(F_{TC})$ z TC_4. Kolejno jest sprawdzane czy $IC_4(7)$ jest większe niż $Min_{indistance}(9)$. Nie jest to zgodne z prawdą, co oznacza, że wątku ThID10 nie można zastąpić wątkiem ThID4. Szukana jest więc najmniejsza zajęta wartość większa niż $10(TC_4, F_{TC})$. Wartością tą jest 12 w TC_{11}. Niestety w tym wypadku również $IC_{11}(1)$ jest mniejsze niż $Min_{indistance}(9)$. Szukana jest więc kolejna najmniejsza zajęta wartość większa niż $12(TC_5)$. W tym

scenariuszu jest to 38(F_{TC}) w liczniku TC_2. IC_2(17) jest większe niż $Min_{indistance}$(9), więc wybranie tego wątku nie spowoduje jego zbyt częstego przetwarzania. Następnie przechodzimy do kolejnego kroku algorytmu, który jest krokiem dodatkowym względem tego z poprzedniego rozdziału. Zostaje sprawdzone czy po dopisaniu ThID2 do cyklu następuje błąd spowodowany zapętleniem CP. Na pozycjach od 1 do 8 w zapamiętanym wektorze (Rys. 71A) nie występuje wątek 2, błąd ten więc nie następuje. Wątek ThID2 zostaje dopisany do cyklu. Szeregowanie kończy obliczenie nowej wartości TC_2(68) za pomocą wzoru (18). TTW w rozważanym przypadku również nie jest potrzebne.



Rys. 71. Usuwanie błędów powstałych po zapętleniu cyklu – przykład 2

6.4.2 Program szeregujący zadania

W celu szybkiego i łatwego szeregowania zadań zaprojektowano program komputerowy „Task Arrangment". Omawiany program został zaprogramowany w języku C# [71] (C Sharp) za pomocą programu Visual Studio 2017.

Program został zaprezentowany na Rys. 72. W sekcji „Input data" wprowadza się parametry niezbędne do procesu szeregowania. Parametry $Fcore_j$ oraz ThID i TF każdego wątku generowane są na etapie podziału zadań (6.3 Podział zadań strona 90). Adresy ThID wątków oraz ich parametry TF wprowadza się bezpośrednio do tabeli w sekcji „Input Data". W tabeli tej można również zaznaczyć, za pomocą odpowiedniego pola wyboru, które z wątków mają być watkami typu SHT. Parametr $Min_{indistance}$ zależy od struktury rdzeni systemu (5.1.2.1 Ograniczenia przeplotu wątków, strona43). Pozostałe parametry WL oraz TC_{marg}, są parametrami szeregowania i zostały one szerzej omówione w poprzednich podrozdziałach. Pole wyboru „Bubble Optimization" włącza lub wyłącza eliminację jałowych cykli potoku przedstawioną w podrozdziale 6.4.1.5.

Przyciski „Load Task TFs" oraz „Save Task TFs" pozwalają zapisać wszystkie dane wejściowe (dane z sekcji „Input Data") do pliku „ta.txt". Pozwala to uniknąć wielokrotnego wprowadzania tych samych danych. Przycisk „Calculate Initial Parameteres" pozwala obliczyć parametry inicjalizujące z podrozdziału 6.4.1.2. Wartości tych parametrów (TW, IV i TTW), zostaną policzone i wyświetlone w sekcji „Output Data". Dodatkowo pod etykietą „FSys min" wyświetlana jest minimalna częstotliwość, dla której można przeprowadzić proces szeregowania, aby warunek minimalnego przelotu ($Min_{indistance}$) był spełniony (wzór (14) na stronie 105).

Przycisk „Change Interleaving File Name and Path" pozwala nazwać plik zawierający przeplot („.il.txt") oraz wybrać lokalizacje jego zapisu. Pełna ścieżka zapisu wraz z nazwą pliku wyświetlana jest w lewym dolnym rogu aplikacji.

Szeregowanie rozpoczyna się po wciśnięciu przycisku „Arranging tasks and save the file". Uruchamiany jest wtedy algorytm omawiany w poprzednich podrozdziałach. Po tym procesie, w sekcji „Output Data", wyświetlana jest liczba i rodzaj błędów, które powstały na skutek szeregowania.

Rys. 72. Program ułatwiający szeregowanie zadań

6.5 Symulacja wykonywania zadań

Mikroarchitektura projektowanego systemu jest przewidywalna czasowo. Jedyne rozbieżności czasu wykonywania instrukcji mogą pochodzić z wykonywania instrukcji wymiany danych z innymi wątkami (w przypadku wątków typu HT (5.3.4.4 Przewidywalność czasowa wymiany danych, strona 81)) oraz z momentu uruchomienia zadania. Podczas wykonywania instrukcji wątków SHT (5.3.4.3 Typy obsługiwanych wątków, strona 78) różnica między maksymalnym, a minimalnym czasem wykonania zadania jest minimalna.

6.5.1 Opis Symulatora

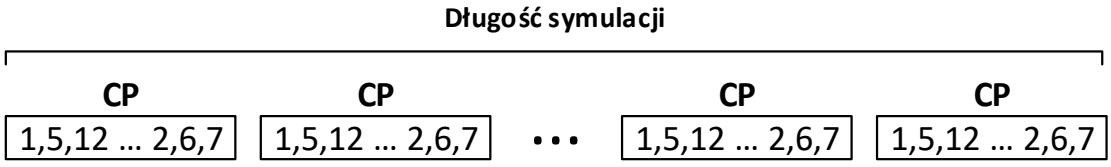
Bazując na powyższych założeniach zaprojektowany został symulator obliczający częstotliwość wykonanych instrukcji poszczególnych wątków. Symulacja jest wykonywana na podstawie wygenerowanego pliku zawierającego CP (cykl przeplotu) (plik „.il.txt"). Budowa symulatora zostanie przedstawiona w tym podrozdziale.

Symulator zlicza wszystkie wystąpienia zadań podczas przebiegu symulacji. W pliku zawierającym CP (plik „.il.txt") zapisana się kolejność przetwarzanych zadań, dlatego w prosty sposób można zebrać dane i na ich podstawie opracować odpowiednie statystki wykonywania poszczególnych zadań.

W poprzedniej wersji systemu [53], [75] stosowany był układ kontroli czasu wykonania zadania (Dodatek C - Mechanizm kontroli czasu wykonania zadania- strona 213). W aktualnej wersji nie jest on jednak potrzebny, ponieważ symulator zapewnia, że czas wykonania zadania nie zostanie przekroczony.

6.5.1.1 Czas trwania symulacji

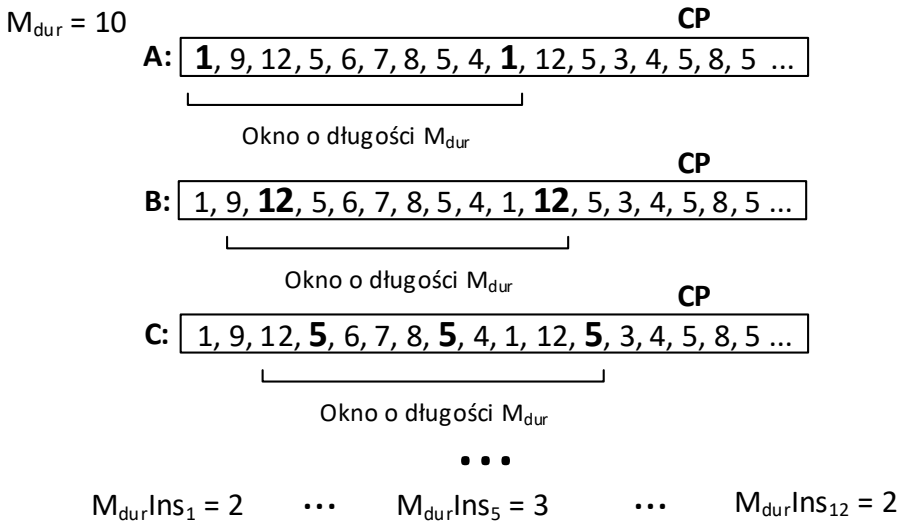
Pierwszym rozważanym parametrem symulacji będzie jej długość wyrażona liczbą cykli CP (Rys. 73). Aby pozyskać dane wszystkich harmonogramowanych zadań, jej minimalna długość powinna mieć wartość pozwalającą wykonać każde zadanie przynajmniej dwa razy – zostanie to wyjaśnione później.



Rys. 73. Długość symulacji

Każde z zadań posiada parametr C_i (4.2

Model zadania, strona 23). W przypadku zadań NCT (zadań niewspółpracujących), C_i jest długością tych zadań wyrażoną w instrukcjach, ponieważ po wykonaniu tylu instrukcji zadanie zostanie wykonane. W przypadku zadania CT (zadania współpracujące, o parametrze M_i różnym od zera) sytuacja jest bardziej skomplikowana. Długość wymiany danych z innym wątkiem wyrażana jest za pomocą parametru M_{dur} (5.3.4.4 Przewidywalność czasowa wymiany danych, strona 81). Jest on wyrażany w cyklach zegara systemowego. Aby obliczyć długość wykonywania zadania wyrażoną w instrukcjach, należy zamienić ten parametr na maksymalną liczbę instrukcji C_i jaki ma dany wątek wykonać, żeby instrukcja wymiany danych została wykonana. Wartość ta zależy od zawartość cyklu CP, dlatego należy sprawdzić rozkład uszeregowanych zadań. Proces ten przedstawiono na Rys. 74.



Rys. 74. Liczba instrukcji Ci potrzebnych do wykonania instrukcji wymiany danych z innym wątkiem

W procesie tym tworzone jest okno o długości M_{dur}, w rozważanym przykładzie równe 10. Jest ono cykliczne przesuwane w prawo (o jeden identyfikator), aż do końca CP. Na rysunku przedstawiono 3 pierwsze pozycje okna (Rys. 74 A, B i C). W każdym z takich okien obliczana jest liczba wystąpień każdego z identyfikatorów. Następnie zapamiętywana jest maksymalna liczba wystąpień w parametrach $M_{dur}Ins_i$. Jest to właśnie maksymalna liczba instrukcji Ci potrzebnych do wykonania instrukcji wymiany danych z innym wątkiem.

W przypadku wątków SHT, wartość $M_{dur}Ins_i$ będzie stała w każdym sprawdzanym oknie i można ją obliczyć za pomocą wzoru (19).

$$M_{dur}Ins_i = \left\lceil \frac{M_{dur}}{TW_{SHT_i}} \right\rceil \tag{19}$$

gdzie:

$M_{dur}Ins_i$ – Maksymalna liczba instrukcji standardowych, potrzebnych do wykonania instrukcji wymiany danych z innym wątkiem

M_{dur} – Liczba standardowych instrukcji, potrzebnych do wykonania instrukcji wymiany danych

TW_{SHT_i} – Parametr TW_i zadania typu SHT dla którego przeprowadzane są obliczenia

Znając wartość $M_{dur}Ins_i$ każdego z wątków w prosty sposób, za pomocą wzoru (20), można obliczyć maksymalną długość każdego wątku wyrażoną w standardowych instrukcjach. Parametr ten będzie opisany symbolem $MaxI_i$ („Max instruction”).

W przypadku zadań typu NCT oraz CTHM liczba instrukcji potrzebnych do wykonania będzie zawsze stała. Parametr $MaxI_i$ będzie więc równy $MinI_i$. W przypadku zadań CT zadania mogą wykonać się wcześniej. Parametr $MinI_i$ dla tych zadań można obliczyć ze wzoru (20), ponieważ w najlepszym wypadku instrukcja wymiany danych wykona się w tym samym czasie co standardowa instrukcja.

$$MaxI_i = M_{dur}Ins_i \cdot M_i + C_i \quad (20)$$

gdzie:

$MaxI_i$ – Odpowiednik maksymalnej liczby instrukcji standardowych, potrzebnych do wykonania zadania *i*

$M_{dur}Ins_i$ – Maksymalna liczba instrukcji standardowych, potrzebnych do wykonania instrukcji wymiany danych z innym wątkiem

M_i – Liczba standardowych instrukcji, potrzebnych do wykonania zadania *i* (bez M_i)

C_i – Liczba instrukcji wymiany danych z innym wątkiem, potrzebnych do wykonania zadania *i*

$$MinI_i = M_i + C_i \quad (21)$$

gdzie:

$MinI_i$ – Odpowiednik maksymalnej liczby instrukcji standardowych, potrzebnych do wykonania zadania *i*

M_i – Liczba standardowych instrukcji, potrzebnych do wykonania zadania *i* (bez M_i)

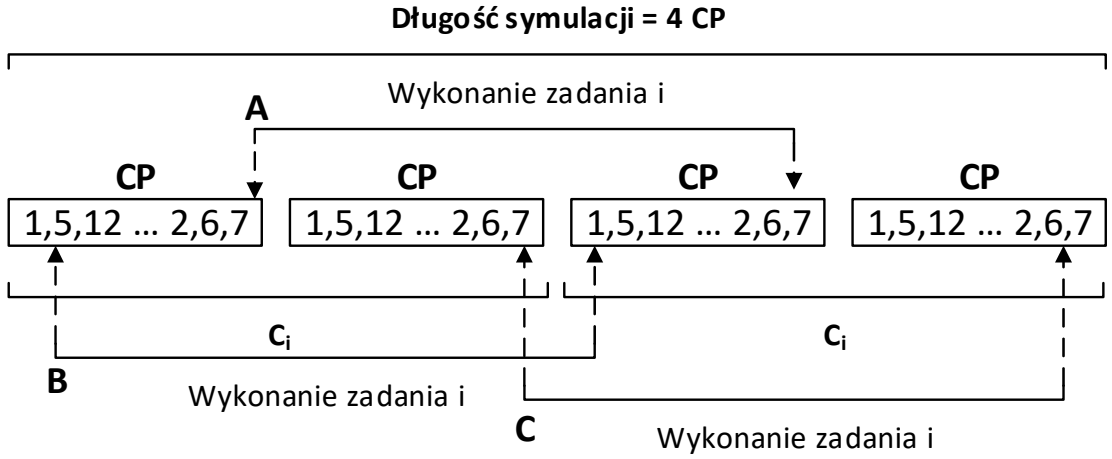
C_i – Liczba instrukcji wymiany danych z innym wątkiem, potrzebnych do wykonania zadania *i*

Parametr $MaxI_i$ wątków typu NCT jest zawsze równy C_i tych wątków – M_i tych zadań jest równe 0. Jeżeli podczas symulacji zostanie przetworzonych tyle identyfikatorów danego zadania ile wynosi jego $MaxI_i$, oznacza to, że podczas procesu symulacji wykonano dane zadanie co najmniej jednokrotnie.

Niemożliwe do przewidzenia są jednak momenty uruchomienia poszczególnych zadań, wynika to z założeń zastosowania systemu (4.1

Charakterystyka systemu, strona 22). Długość symulacji pozwalająca na co najmniej dwukrotne wykonanie najdłuższego zadania pozwala na sprawdzenie wszystkich takich

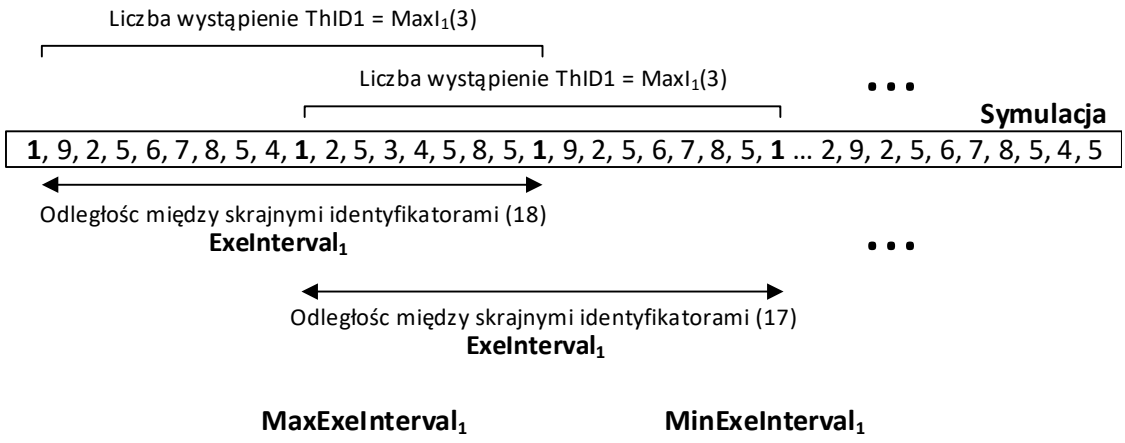
możliwości. Załóżmy, że najdłuższe zadanie (największe $MaxI_i$) potrzebuje dokładnie 2 cykli CP, żeby wykonać się jednokrotnie (Rys. 75). Minimalna długość symulacji musi wynosić więc 4 CP. Zauważmy, że jeżeli zadania zostaną uruchomione w momentach A, B, C przedstawionych na Rys. 75 lub w dowolnym inny momencie podczas dwóch pierwszych cykli CP, zadania te wykonają się w całości. Podczas pracy systemu cykle te będą przetwarzane w nieskończoność, ale wystarczy przeanalizować ten fragment, aby sprawdzić przewidywalność czasową systemu. Dzieje się tak, ponieważ analizowana kombinacja identyfikatorów ThID będzie nieustannie powtarzana.



Rys. 75. Minimalna długość symulacji

6.5.1.2 Minimalny i maksymalny czas wykonania zadań

Obliczanie maksymalnego i minimalnego czasu wykonywania zadań zrealizowane jest etapami. W pierwszym kroku obliczana jest maksymalna i minimalna liczba taktów zegara potrzebna na wykonanie danego wątku. W sprawdzanym w symulacji ciągu identyfikatorów ThID szukane są wszystkie możliwe przedziały, w których może się rozpocząć i zakończyć zadanie. Znając parametry $MaxI_i$ oraz $MinI_i$ zadań należy znaleźć wszystkie przedziały, w których na pierwszym i ostatnim miejscu występuje szukany identyfikator oraz w przedziale występuje liczba identyfikatorów równa $MaxI_i$ oraz $MinI_i$ szukanego zadania. Przykład dla zadania o identyfikatorze ThID1, $MaxI_1 = 3$ oraz $MinI_1 = 3$ przedstawiono na Rys. 76.



Rys. 76. Maksymalna oraz minimalna liczba cykli zegara, potrzebna do wykonania zadania

Następnie wyszukiwana jest maksymalna (*MaxExeIntervalᵢ* dla *MaxIᵢ*) oraz minimalna (*MinExeIntervalᵢ* dla *MinIᵢ*) długość znalezionych powyżej przedziałów. Jest to maksymalna oraz minimalna liczba cykli zegara dla powstałego na etapie szeregowania zadań CP, które należy wykonać, aby wykonać instrukcje zadania. W przypadku zadań typu SHT, wartości *MaxExeIntervalᵢ* oraz *MinExeIntervalᵢ* wątku SHT będą sobie równe i mogą zostać obliczone również ze wzoru (22). Dokładny opis realizacji pozyskiwania wartości tych parametrów przez symulator zostanie opisany w następnym podrozdziale.

$$MaxExeInterval_i = MaxI_i \cdot TW_{SHT_i} \tag{22}$$

gdzie:

$MaxExeInterval_i$ – Maksymalna liczba cykli zadania potrzebna do wykonania instrukcji zegara

$MaxI_i$ – Odpowiednik maksymalnej liczba instrukcji standardowych, potrzebnych do wykonania zadania *i*

TW_{SHT_i} – Parametr TW_i zadania typu SHT dla którego przeprowadzane są obliczenia

Do obliczenia maksymalnej oraz minimalnej liczby taktów zegara potrzebnych do wykonania zadania należy również rozpatrzeć proces uruchamiania potoku przetwarzania

oraz oczekiwanie na przetworzenie pierwszej instrukcji zadania. Zadania w systemie przetwarzane są cyklicznie i nigdy nie ma pewności, że zadanie zostanie uruchomione akurat w momencie, kiedy będzie jego kolej w CP. W najgorszym możliwym przypadku zadanie miało być przetwarzane cykl wcześniej niż zostało uruchomione przez przerwanie. Maksymalną liczbę cykli zegara potrzebną do wykonania zadania można więc obliczyć ze wzoru (23).

$$MaxExeTicks_i = MaxExeInterval_i + 4 + MaxTW_i - 1 \qquad (23)$$

gdzie:

$MaxExeTicks_i$ – Maksymalna liczba cykli zadania potrzebna do zakończenia wykonywania zadania

$MaxExeInterval_i$ – Maksymalna liczba cykli zadania potrzebna do wykonania wszystkich instrukcji zadania

$MaxTW_i$ – Maksymalna odległość pomiędzy dwoma instrukcjami tego samego zadania w CP

Parametr $MaxTW_i$ może być sprawdzony przez symulator poprzez analizę pliku CP. W przypadku zadań typu SH, jest on równy parametrowi TW_{SHTi} zadania. Parametr $MaxTW_i$ musi być obniżony o jeden takt, ponieważ wtedy następuje najgorszy przypadek (przetwarzanie instrukcji tego zadania w poprzednim takcie). Składowa czterech taktów, we wzorze (23), pochodzi z opóźnienia potoku przetwarzania systemu.

Minimalna liczba taktów zegara potrzebnych do wykonania zadania powinna uwzględniać najlepszy możliwy przypadek. W takiej sytuacji zadanie zostanie uruchomione w najlepszym dla niego momencie. Liczba tych taktów może być obliczona ze wzoru (24).

$$MinExeTicks_i = MinExeInterval_i + 4 \quad (24)$$

gdzie:

$MinExeTicks_i$ – Minimalna liczba cykli zegara potrzebnych do zakończenia pracy zadania

$MinExeInterval_i$ – Minimalna liczba cykli zadania potrzebna do wykonania wszystkich instrukcji zadania

Znając wartości powyższych parametrów (*MaxExeTicks*$_i$ oraz *MinExeTicks*$_i$) w prosty sposób można policzyć maksymalny oraz minimalny czas wykonania zadania przy danej częstotliwości symulacji (wzory (25) oraz(26)).

$$MaxExeTime_i = MaxExeTicks_i \cdot \frac{1}{Sim_{Freq}} \quad (25)$$

gdzie:

$MaxExeTime_i$ – Maksymalny czas wykonania zadania *i*

$MaxExeTicks_i$ – Maksymalna liczba cykli zegara systemu, potrzebnych do wykonania zadania *i*

Sim_{Freq} – Częstotliwość pracy zegara systemowego, używana podczas symulacji

$$MinExeTime_i = MinExeTicks_i \cdot \frac{1}{Sim_{Freq}} \quad (26)$$

gdzie*:*

$MinExeTime_i$ – Minimalny czas wykonania zadania *i*

$MinExeTicks_i$ – Minimalna liczba cykli zegara systemu, potrzebnych do wykonania zadania *i*

Sim_{Freq} – Częstotliwość pracy zegara systemowego, używana podczas symulacji

Znając parametr D_i (deadline) każdego z zadań możliwe jest również obliczenie minimalnej częstotliwości pracy zegara systemowego, przy której wszystkie zadania zostaną wykonane na czas.

$$Min_{Freq} = Max\left(\frac{MaxExeTicks_i}{D_i}, \frac{MaxExeTicks_{i+1}}{D_{i+1}}, \dots\right) \qquad (27)$$

gdzie:

Min_{Freq} – Minimalna częstotliwość pracy zegara systemowego, przy której wszystkie zadanie zastaną wykonane przed czasem D_i

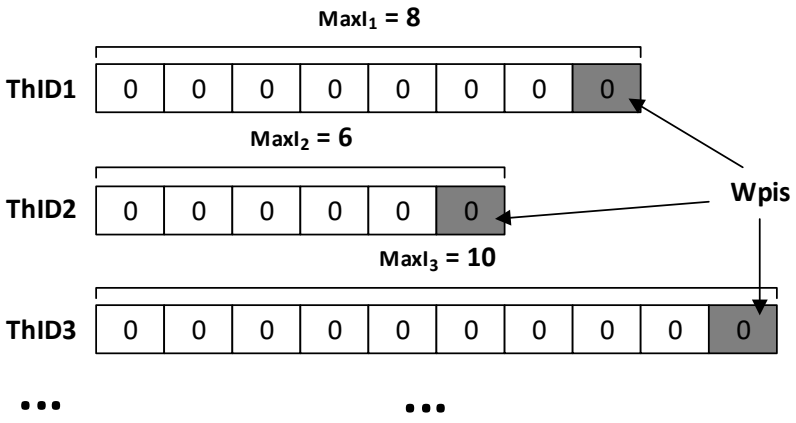
$MaxExeTicks_i$ – Maksymalna liczba cykli zegara systemu, potrzebnych do wykonania zadania *i*

D_i – Maksymalny dopuszczalny czas wykonania zadania *i* w (deadline)

6.5.1.3 Opis implementacji metody wyszukiwania maksymalnej liczby cykli zegara, potrzebnej do wykonania zadania.

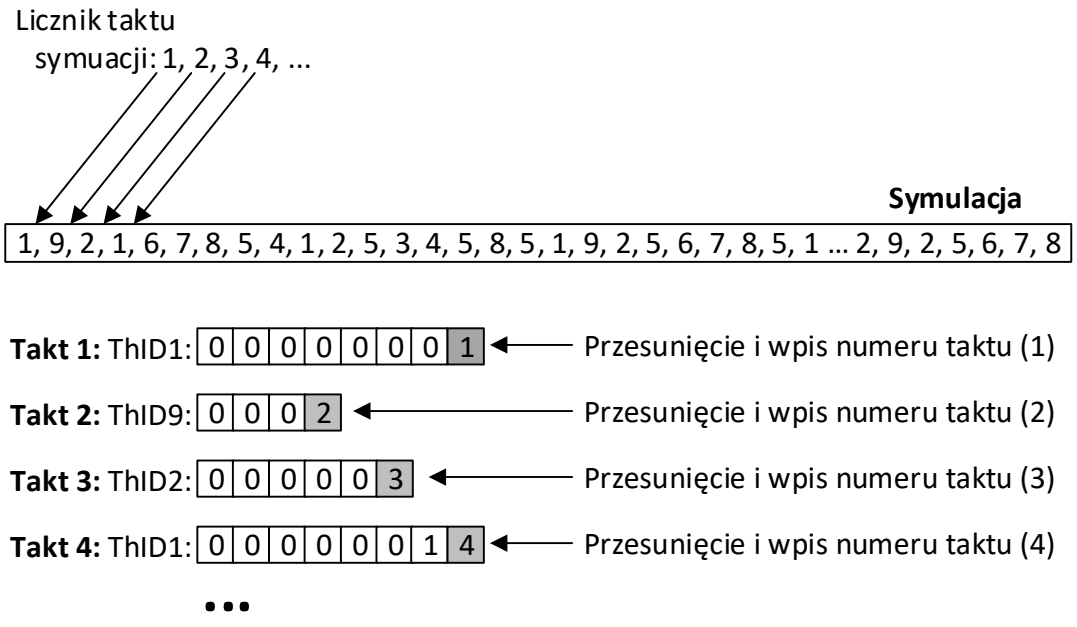
Metoda wyszukiwania maksymalnej oraz minimalnej liczby cykli zegara, opisana w poprzednim podrozdziale, może sprawiać wrażenie bardzo złożonej obliczeniowo. Aby możliwa był jej realizacja w jak najszybszym czasie, opracowano następującą metodę implementacji tego algorytmu.

Każdemu zadaniu pracującemu w systemie przydzielany jest rejestr przesuwny o długości równej wartości $MaxI_i$ oraz $MinI_i$ danego zadania (Rys. 77). Na rysunkach zostanie przedstawiony proces obliczania maksymalnej liczby cykli zegara. Podczas wpisu do takiego rejestru zawartość całego rejestru jest przesuwana w lewo (skrajny lewy element jest tracony). Nowa wartość zostaje wpisana do skrajnego prawego pola (na Rys. 77, komórkę zaznaczono kolorem szarym).



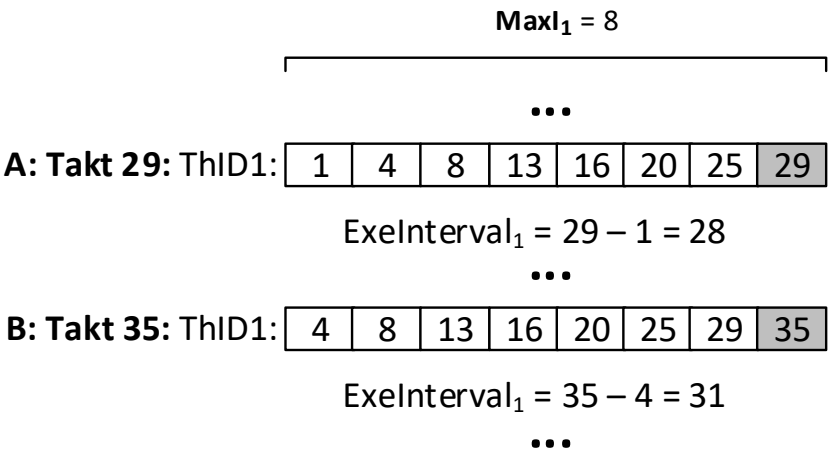
Rys. 77. Rejestry przesuwne przechowywujące takty przetwarzania zadań

Następnie uruchamiany jest proces symulacji polegający na cyklicznym pobieraniu wartości klejonego identyfikatora z ciągu symulacji. Uruchamiany jest również licznik taktu symulacji (Rys. 78) zliczający liczbę identyfikatorów pobranych od początku symulacji. Jeżeli nie jest to cykl jałowy (pobrany identyfikator jest różny od 0) do rejestru przesuwnego, odpowiedniego dla pobranego identyfikatora, wpisywany jest stan licznika symulacji, czyli symulowanego taktu zegara. Początek symulacji (dla pierwszych czterech taktów symulacji) przedstawiono na Rys. 78.



Rys. 78. Proces wypełnienia rejestrów przechowywujących takty przetwarzania zadań

Po takim procesie rejestry przesuwne zadań zawierają numery taktów, w których były wykonywane. Jeżeli któryś z tych rejestrów się wypełni (z pierwszej pozycji rejestru zniknie 0), możliwe jest szybkie i stosunkowo nieskomplikowane wyznaczenie długości wykonywania zadania wyrażone liczbą taktów zegarowych. Wystarczy odjąć skrajnie prawą wartość rejestru od skrajnie lewej (Rys. 79).



Rys. 79. Przykład obliczania długości wykonywania zadania

Jeżeli zadanie zostałoby uruchomione w pierwszym takcie symulacji (Rys. 79A), jego wykonanie wynosiło by 28 taktów zegara. Natomiast, gdy start zadania wystąpiłby w takcie 4 (Rys. 79B), zadanie zostało by wykonane w ciągu 31 cykli zegara. Proces taki jest przeprowadzany, aż do końca symulacji. Maksymalna wartość $ExeInterval_i$ jest zapamiętywana. W przypadku $MinInterval_i$ proces jest identyczny, różnica polega na rozmiarze rejestru ($MinI_i$ zamiast $MaxI_i$) oraz na zapamiętaniu najmniejszej wartości.

6.5.2 Program umożliwiający symulacje

Symulator został zaimplementowany w języku C# [71] (C Sharp) za pomocą programu Visual Studio 2017. Program „Task Symulator" (Rys. 80) umożliwia obliczenie maksymalnego i minimalnego czasu wykonania zadania dla zadanej częstotliwości symulacji („Simulate Freq [MHz]"). Obliczana jest również minimalna częstotliwość pracy systemu, dla której wszystkie zadania zakończą swoją pracę przed swoim czasem D_i (deadline).

Pierwszym krokiem po uruchomieniu programu powinno być załadowanie podstawowych informacji o zadaniach (4.2

Model zadania, strona 23): ThID, Ci, Mi oraz D_i. Jest to realizowane za pomocą przycisku „Load Task Data", który otwiera pliki „.td.txt" zawierające te informacje. Następnie powinien być wczytany plik o rozszerzeniu „.il.txt" zawierający cykl CP, poprzez użycie przycisku „Load Interleaving file". Ścieżka do załadowanego pliku „.il.txt" wyświetlana jest w lewym dolnym rogu okna programu.

W pierwszym polu tekstowym „Simulation Windows" wprowadzana jest długość symulacji wyrażona liczbą okien CP (6.5.1.1 Czas trwania symulacji, strona 126). W polu poniżej, o nazwie „Core Label", można wprowadzić etykietę dla aktualnie symulowanego CP (każde CP to jeden rdzeń). Zostanie ona wyświetlona w tabeli obok w kolumnie „Core". Pozwoli ona rozróżnić rdzenie w tabeli, ponieważ możliwe jest wczytanie jednego pliku „.td.txt" z danymi dla wszystkich zadań pracującymi w systemie. Następnie można kolejno symulować pliki CP dla wszystkich rdzeni, aż do wypełnienia całej tabeli (przeprowadzenia symulacji dla wszystkich zadań pracujących w systemie).

Przycisk „Calculate Exe Ticks" uruchamia pierwszą fazę symulacji obliczając liczbę taktów systemu potrzebnych do wykonania zadań. Policzone dane wyświetlane są w tabeli. Dodatkowo obliczane i wyświetlane są takie parametry jak:

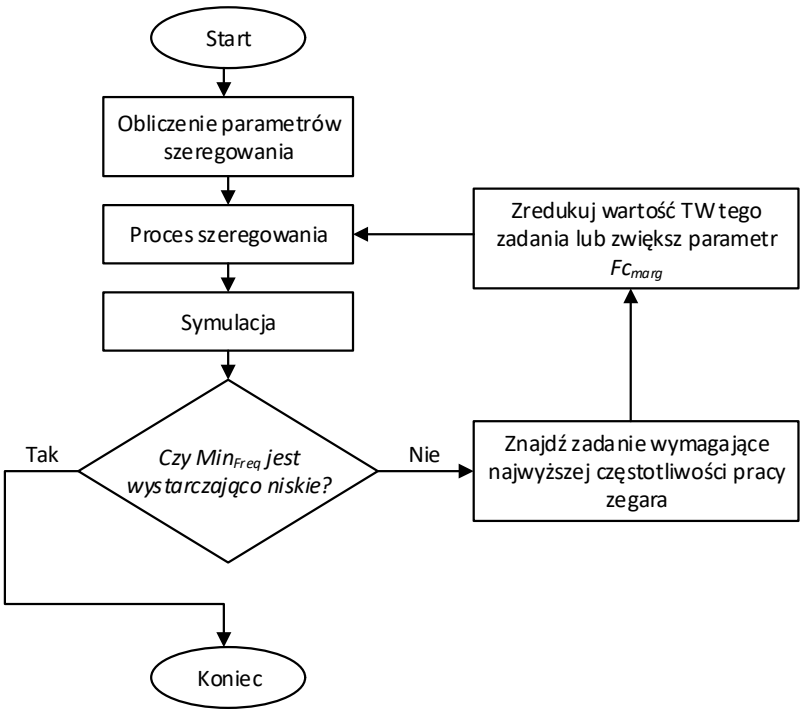
- „Min Inter" – minimalna odległość między takimi samymi identyfikatorami ThID w symulowanym ciągu
- „Number of bubbles" – liczba jałowych cykli w symulowanym ciągu
- „Simulation Lenght (Ticks)" – długość symulacji wyrażona w cyklach zegara
- „MinFreq [MHz]" – minimalna częstotliwość pracy systemu, dla której wszystkie zadania zostaną wykonana przed czasem D_i

Drugi przycisk, „Calculate Exe Time", uruchamia obliczenia związane z czasem wykonania zadań. Na podstawie wybranej częstotliwości symulacji (pole „Simulation Freq [MHz]") obliczany jest między innymi maksymalny i minimalny czas wykonania zadania (parametry „MinExeTime[ns]" oraz „MaxExeTime[ns]"). Wszystkie wyniki umieszczane są w tabeli.

Rys. 80. Program umożliwiający symulacje

6.5.3 Optymalizacja procesu szeregowania

Symulator pozwala na zweryfikowanie i optymalizację procesu szeregowania. W niektórych przypadkach parametry TW zadań są tak dobrane, że po procesie szeregowania niektóre z zadań mają znacznie więcej błędów „Delay Error" oraz „Skip error" niż pozostałe. Powoduje to, że te pojedyncze zadania ograniczają minimalną częstotliwość pracy systemu. Symulator na podstawie pliku CP może wyliczyć minimalną częstotliwość pracy każdego z zadań, a więc i wyznaczyć, które zadanie wymusza zwiększenie częstotliwości pracy systemu. Proces szeregowania można zoptymalizować przez zmniejszenie parametru TW takiego zadania lub zwiększając parametr FC_{marg}, który globalnie zmniejszy wartości TW wszystkich zadań. Ideę procesu optymalizacji pokazano na Rys. 81.



Rys. 81. Schemat ideowy optymalizacji procesu szeregowania

7 Weryfikacja działania systemu

W rozdziale tym zostanie przeprowadzona weryfikacja metodologii harmonogramowania zadań oraz koncepcji i implementacji systemu. Pierwsze dwa podrozdziały będą dotyczyć weryfikacji modelu systemu. W trzecim rozdziale zostanie zweryfikowana metodologia harmonogramowania.

7.1 Weryfikacja konfigurowalności systemu

W podrozdziale tym przedstawione zostaną wyniki eksperymentów przeprowadzonych z wykorzystaniem zaprojektowanego systemu. Dotyczyć będą one zapotrzebowania systemu na zasoby oraz energię w różnych jego konfiguracjach. Następnie system zostanie porównany również do jednozadaniowego systemu niewykorzystującego przeplotu wątków.

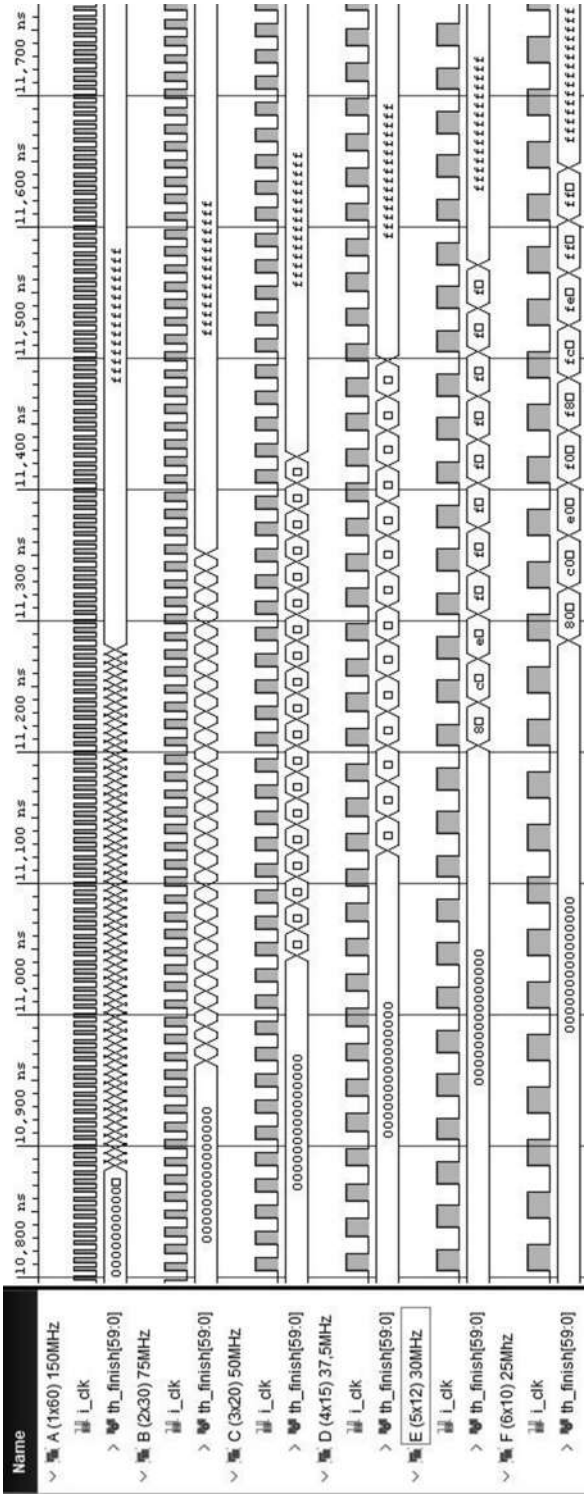
7.1.1 Scenariusz testowy

Głównym scenariuszem testowym są różne konfiguracje podziału sześćdziesięciu takich samych zadań. Zadania te są wykonywane w systemach o różnej liczbie rdzeni. Parametry pracy tych systemów są tak dobrane, że w każdym ze scenariuszy system wykonuje tę samą pracę w tym samym czasie. Taka metodologia testowania pozwoli osiągnąć wiarygodne porównanie. Scenariusze utworzono według następującego klucza:

A. System jednordzeniowy (1x60 zadań) pracujący z zegarem 150[MHz]
B. System dwurdzeniowy (2x30 zadań) pracujący z zegarem 75[MHz]
C. System trzyrdzeniowy (3x20 zadań) pracujący z zegarem 50[MHz]
D. System czterordzeniowy (4x15 zadań) pracujący z zegarem 37,5[MHz]
E. System pięciordzeniowy (5x12 zadań) pracujących z zegarem 30[MHz]
F. System sześciordzeniowy (6x10 zadań) pracujących z zegarem 25[MHz]

Zadania, które przetwarzane są równolegle (w systemach wielordzeniowych), wykonują się szybciej, dlatego zrekompensowano to zmniejszając częstotliwość zegara takich systemów. Na Rys. 82 przedstawiono symulację pokazującą czas wykonania powyższych scenariuszy. Sygnał „i_clk” to zegar każdego ze scenariuszy, natomiast wektor sygnałów „th_finish” to sygnały oznaczające zakończenie pracy każdego z 60-ciu

zadań (każdy sygnał to jedno zadanie, zmiana stanu z niskiego na wysoki oznacza wykonanie zadania).



Rys. 82. Czas wykonania scenariuszy, symulacja

Jak można zauważyć, wykonanie zadań nie następuje równocześnie w każdym scenariuszu. Różnica między najszybszym wykonaniem scenariusza A (11,278 [ns]), a najwolniejszym F (11,645 [ns]) wynosi ok. 367 [ps]. Wynika ona z przetwarzania potokowego procesora. Aby procesor potokowy wykonał pierwszy rozkaz musi uruchomić potok. W omawianej konfiguracji potrzeba 11 taktów zegara, aby procesor rozpoczął swoją pracę. 11 taktów zegara, trwa inny czas dla zegara 150 [MHz] (73.33 [ps]), a inny dla 25 [MHz] (440 [ps]) (przesunięcie między scenariuszem A i F to 440 [ps] – 73 [ps] = 367 [ps]). Przesunięcie to jest więc stałe i niezależne od długości pracy systemu. Po uwzględnieniu czasu „startu" scenariusza, każdy z nich wykonuje tę samą pracę w tym samym czasie (60 zadań w czasie 11,2 [ns]).

7.1.2 Badanie zapotrzebowania systemu na zasoby

Podczas wykonywania eksperymentów wszystkie konfiguracje systemu były syntezowane oraz implementowane do układu FPGA za pomocą oprogramowania Vivado 2018.3. Po każdej syntezie program ten generuje raport zasobów wymaganych do implementacji systemu. Przykładowy raport umieszczono na Rys. 83. Najbardziej podstawowe zasoby, które będą rozważane to „Slice LUTs", „Slice Registers", „F7 Muxes" oraz „F8 Muxes". „Slice LUTs" oraz „Slice Registers" to podstawowe zasoby występujące w układach FPGA, natomiast „F7 Muxes" oraz „F8 Muxes" to dodatkowe zasoby przełączające (multipleksery). Pozostałe zasoby przedstawione na rysunku nie dotyczą samej implementacji systemu. „Block RAM Tile" to zasoby, z których tworzona jest pamięć wątków (5.1.1 Pamięć oraz rejestry systemu, strona 38). Pamięć ta została utworzona za pomocą dedykowanego makra, dlatego każde zadanie pracujące w systemie wymaga zawsze jednego takiego zasobu. Na przedstawionym przykładzie (Rys. 83) przedstawiono system składający się z 1 rdzenia, w którym pracuje 30 zadań. Z powyższego powodu, zdecydowano się w dalszych badaniach tego zasobu nie rozważać, ponieważ jego liczba jest uzależniona tylko i wyłącznie od liczby zadań pracujących w systemie. Kolejny pokazany zasób to „Bonded IOB". Jest on wykorzystywany do tworzenia portów wejścia/wyjścia z układu FPGA. Jego wielkość jest więc uzależniona od liczby sygnałów jakimi system będzie się komunikował ze środowiskiem pracy. Ostatnim omawianym zasobem jest „MMCM E2_AD". Zasób ten jest potrzebny do utworzenia układu podziału lub mnożnika częstotliwości (ang. „Mixed-Mode Clock Manager"), odpowiednika układu pętli PLL. Układ ten pozwala wytworzyć

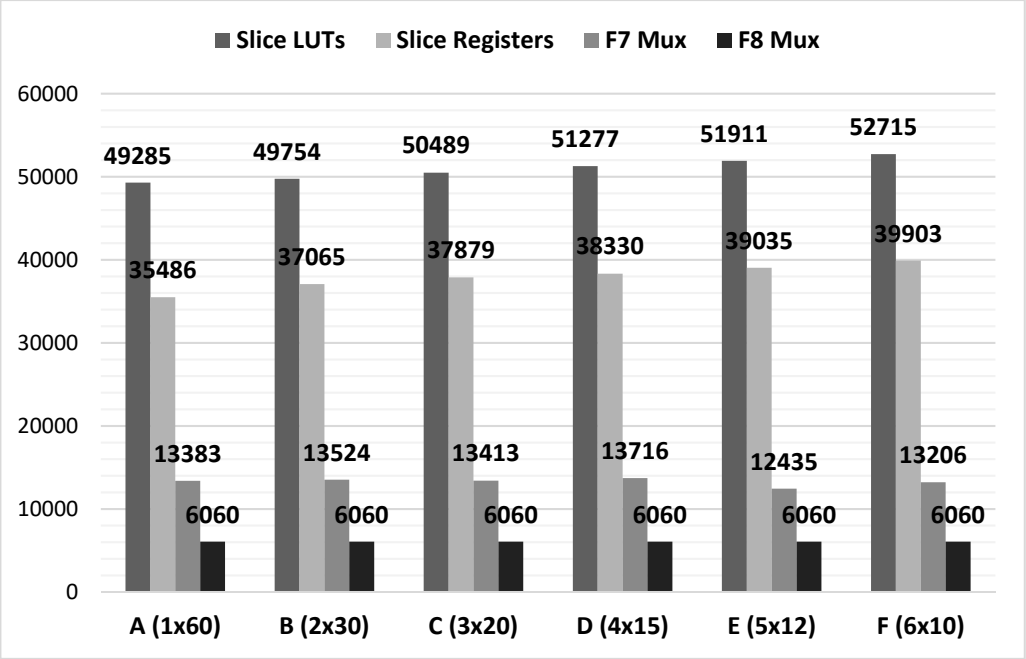
dowolną (z pewnego przedziału) częstotliwość zegara taktującego system. W przypadku wytworzenia jednej częstotliwości wyjściowej potrzebny będzie zawsze tylko jeden taki zasób.

Name	Slice LUTs (303600)	Slice Registers (607200)	F7 Muxes (151800)	F8 Muxes (75900)	Block RAM Tile (1030)	Bonded IOB (700)	MMCM E2_ADV (14)	LUT as Logic (303600)	LUT as Memory (130800)
top_design_wrapper	25000	18521	6763	3030	30	11	1	24925	75
top_design_i (top_design)	25000	18521	6763	3030	30	0	1	24925	75
clk_wiz (top_design_clk_...	0	0	0	0	0	0	1	0	0
rst_clk_wiz_100M (top_de...	15	108	0	0	0	0	0	14	1
top_system_0 (top_desig...	24985	18413	6763	3030	30	0	0	24911	74
inst (top_design_top_s...	24985	18413	6763	3030	30	0	0	24911	74
core_with_mem_rg...	24985	18413	6763	3030	30	0	0	24911	74

Rys. 83. Przykładowy raport zużycia zasobów sprzętowych

Podsumowując, w eksperymentach zapotrzebowania systemu na zasoby zdecydowano się badać zużycie czterech zasobów: „Slice LUTs”, „Slice Registers”, „F7 Muxes” oraz „F8 Muxes”.

W pierwszym przeprowadzonym eksperymencie zbadano zużycie zasobów dla scenariuszy wykonujących te same zadania w tym samym czasie (7.1.1 Scenariusz testowy, strona 141). Wyniki badań umieszczono na Rys. 84. Można na nim zauważyć, że zgodnie z przewidywaniami, liczba podstawowych zasobów „Slice LUTs” oraz „Slice Registers” rośnie wraz z liczbą rdzeni systemu. Jest to spowodowane tym, że w konfiguracjach wielordzeniowych duplikowana jest logika odpowiedzialna za potok przetwarzania systemu. Zasoby „F7 Mux” są porównywalne w każdym scenariuszu. Wyjątkiem jest tutaj konfiguracja E i F, w której można zaobserwować spadek tych zasobów. Odstępstwo to, najprawdopodobniej jest spowodowane znalezioną możliwością optymalizacji dla tej konfiguracji przez narzędzie syntezy. Liczba zasobów „F8 Mux” jest taka sama dla wszystkich scenariuszy. Podczas procesu implementacji wykorzystywane są one do utworzenia banków rejestrów ogólnego przeznaczenia, których w każdym scenariuszy jest tyle samo (16).

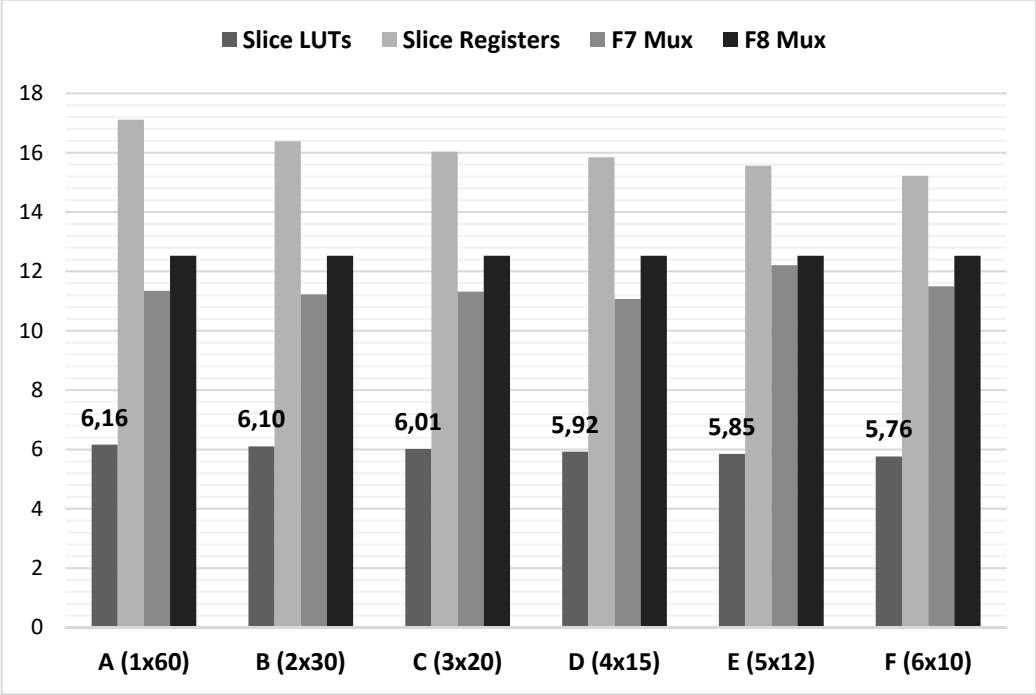


Rys. 84. Zużyte zasoby w różnych scenariuszach

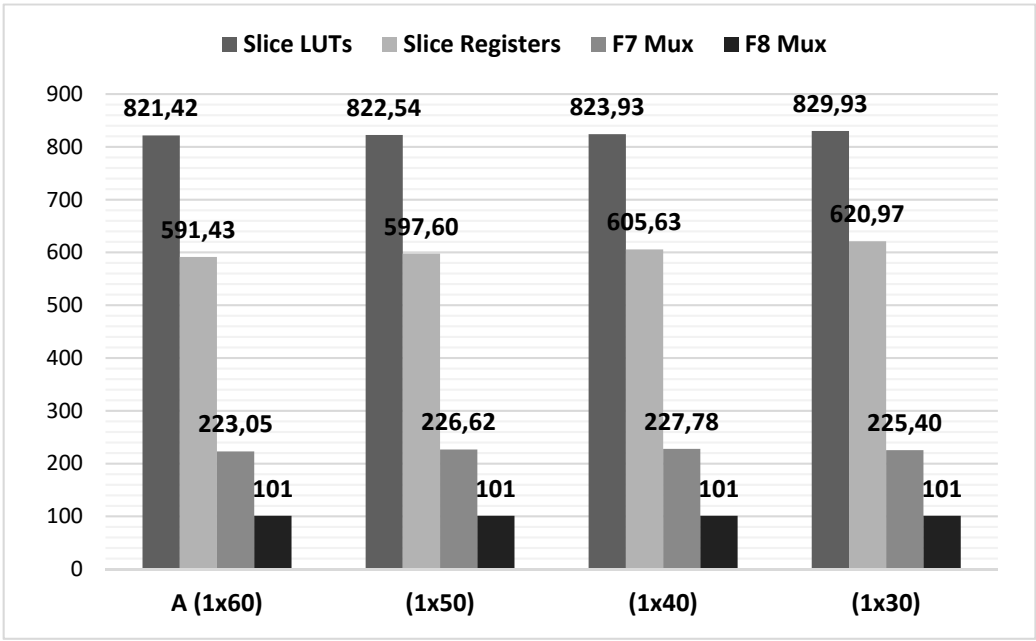
Następnie, na podstawie liczby dostępnych zasobów w wykorzystywanym układzie Virtex-7, obliczono teoretyczną liczbę systemów w różnych wariantach (scenariusze od A do F), które można uruchomić w jednym układzie FPGA. Na Rys. 85 umieszczono uzyskane wyniki. Dokładne liczbowe wyniki na wykresie umieszczono tylko dla zasobów „Slice LUTs", ponieważ te zasoby są decydujące o liczbie systemów w układzie. Z pozyskanych danych można odczytać, że dla scenariuszy A, B oraz C maksymalna liczba systemów to 6. Oznacza to, że w tych konfiguracjach teoretyczna maksymalna liczba obsługiwanych zadań to 360 (sześć systemów 60 zadaniowych). W przypadku scenariuszy D, E i F, liczba ta spada do 5, czyli 300 obsługiwanych zadań (pięć systemów 60 zadaniowych).

W kolejnej części eksperymentów przeprowadzono implementację innych konfiguracji, w których scenariusze mają różną wydajność. Scenariusze nazywano według następującego klucza: *X*x*Y*, gdzie *X* to liczba rdzeni systemu, natomiast *Y* to liczba zadań w każdym z rdzeni. W pierwszym badaniu porównano konfiguracje jednordzeniowe z różną liczbą zadań. Każdy scenariusz implementowano z tą samą częstotliwością zegara systemowego, wynoszącą 100 [MHz]. Następnie w każdym scenariuszu obliczono ilość zasobów jaka przypada na jedno zadanie. Wyniki przedstawiono na Rys. 86. Można z niego wyciągnąć jednoznaczny wniosek, że

najbardziej oszczędne pod względem zasobów, są konfiguracje z jak największą liczbą zadań w jednym rdzeniu.

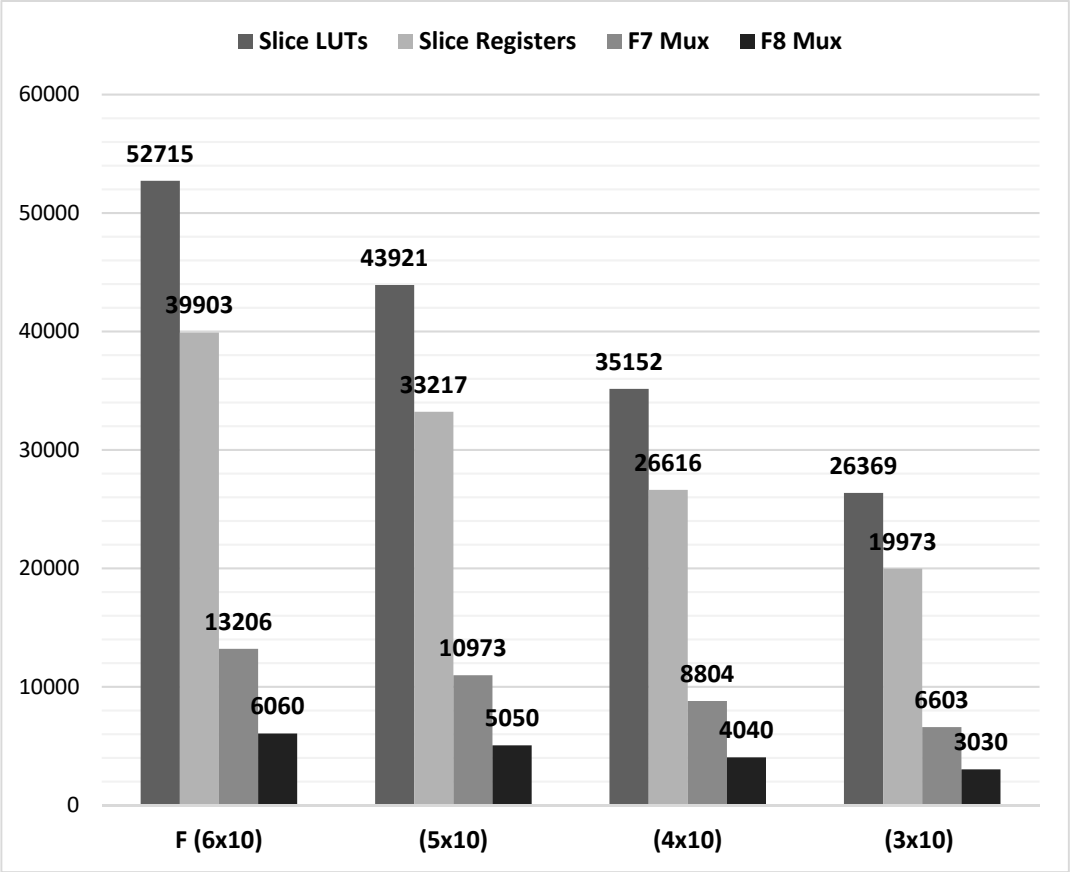


Rys. 85. Teoretyczna liczba systemów mieszczących się w wykorzystywanym układzie Virtex-7



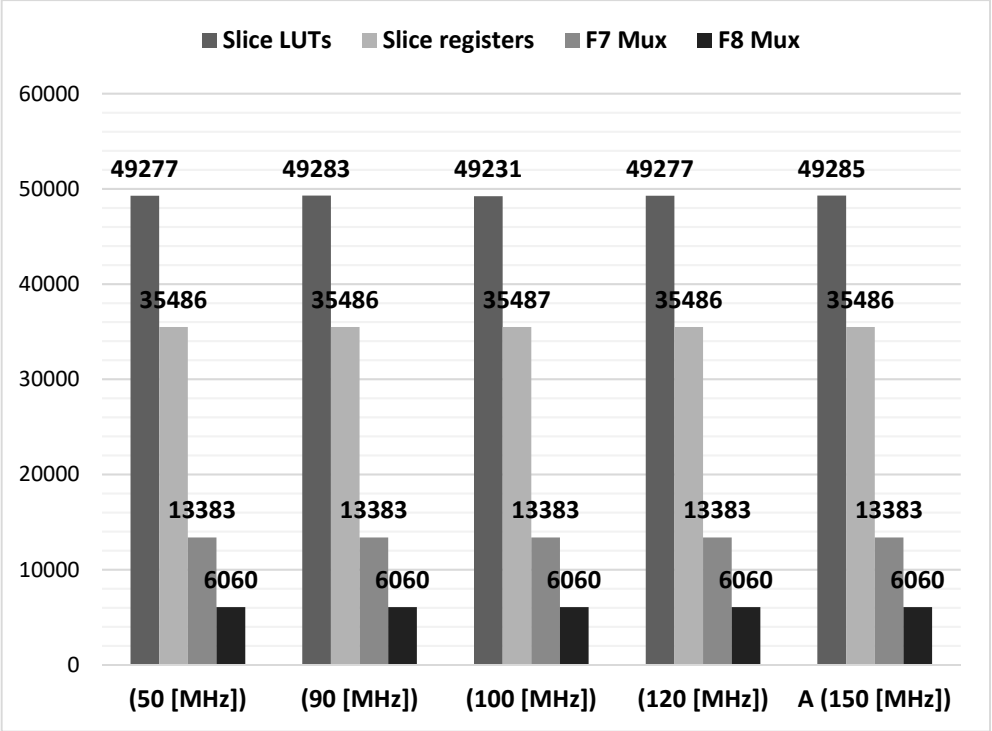
Rys. 86. Liczba zasobów na zadanie, w jednordzeniowych konfiguracjach

Następnie sprawdzono, w jaki sposób przyrastają zasoby zwielokrotniając liczbę takich samych dziesięciowątkowych rdzeni. Zależność tą pokazano na Rys. 87. Wszystkie zasoby przyrastają liniowo wraz z liczbą rdzeni.



Rys. 87. Zużycie zasobów, przy różnych liczbach dziesięciowątkowych rdzeni

Kolejnym badaniem było sprawdzenie jaki wpływ na wymagane zasoby ma zmiana częstotliwości pracy zegara systemowego. Wszystkie pomiary przeprowadzono dla takiej samej konfiguracji systemu: jednordzeniowego systemu przetwarzającego 60 wątków. Każdy pomiar przeprowadzono dla innej częstotliwości zegara. Wyniki przedstawiono na Rys. 88. Wpływ częstotliwości na zasoby jest bardzo niewielki i spowodowany jest inną optymalizacją syntezy układu, jeżeli czas propagacji sygnałów musi być krótszy (częstotliwość pracy zegara jest wyższa). Dlatego w dalszych rozważaniach przyjęto, że zużycie zasobów jest niezależne od częstotliwości pracy zegara systemu.



Rys. 88. Wpływ częstotliwości pracy zegara, na zapotrzebowanie na zasoby systemu. Jednordzeniowa konfiguracja przetwarzająca 60 zadań.

7.1.3 Badania zapotrzebowania systemu na energie

W kolejnym etapie badań sprawdzano jak w różnych konfiguracjach systemu zmienia się jego zapotrzebowanie na energię. Używane narzędzie, Vivado 2018.3, pozwala po syntezie opisu sprzętu oszacować zużywaną przez system moc [76]. W realizowanych badaniach moc szacowana będzie w sposób probabilistyczny, oznacza to, że zużywana moc nie jest obliczana na podstawie przełączeń sygnałów pochodzących z symulacji. Symulacja na tak niskim poziomie trwała by zbyt długo, aby można ją było wykonać na standardowym komputerze PC. Probabilistyczne szacowanie mocy odbywa się za pomocą statystyki i prawdopodobieństwa przełączeń sygnałów. Proces taki trwa stosunkowo szybko, ale niestety wyniki bezwzględne pomiarów nie są dokładne. Natomiast stosunki pomiarów uzyskanych tą metodą są bardzo dokładne. W rozdziale tym, będą porównywane różne konfiguracje systemów, aby sprawdzić, kiedy zapotrzebowanie na moc będzie najmniejsze. Dlatego metoda ta jest wystarczająca.

Następną informacją wartą rozważenia, jest różnica między mocą a energią w systemach czasu rzeczywistego. Energia to moc w jednostce czasu, wyrażana jest za pomocą jednostki [J]. Energię można obliczyć za pomocą wzoru (28).

$$E = P \cdot t \tag{28}$$

gdzie:

E – Energia, wyrażana w [J]

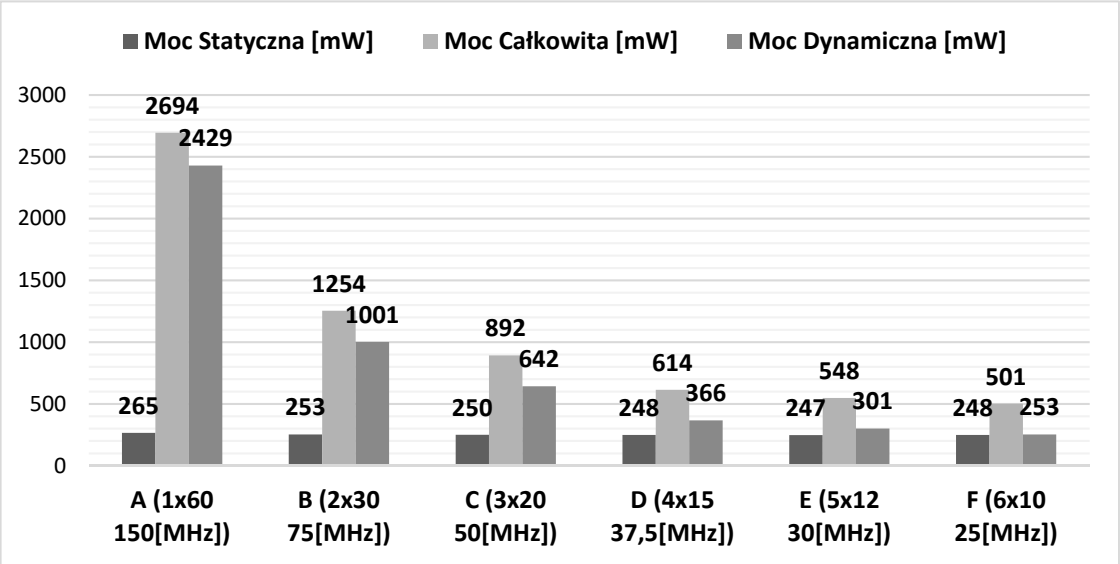
P – Moc, wyrażana w [W]

t – Czas, wyrażany w [s]

W przypadku omawianego systemu czasu rzeczywistego, czas wykonania tego samego zadania w różnych konfiguracjach powinien trwać tą samą ilość czasu (zadanie zawsze ma swój maksymalny czas wykonania – czas D_i). Oznacza to, że wystarczy porównywać tylko wyniki zużycia mocy. Stosunki mocy oraz energii w badanych scenariuszach będą takie same, ponieważ czas wykonywania zadań jest taki sam. Natomiast wartości bezwzględne, zarówno mocy jak i energii, nie są pozyskiwane w dokładny sposób – są szacowane metoda probabilistyczną.

Pierwszym eksperymentem było sprawdzenie zużycia mocy przez konfiguracje zaprezentowane w rozdziale 7.1.1 Scenariusz testowy (strona 141). W pierwszym badaniu zostanie oszacowana moc dla całego układu. Zarówno dla zaprojektowanego systemu jak i układu dzielnika/zwielokrotnienia częstotliwości (MMCM). Używane oprogramowanie może oszacować moc całkowitą i statyczną tylko dla całego układu FPGA. Wyniki przedstawiono na Rys. 89. Jak można zaobserwować, moc statyczna nieznacznie maleje wraz z częstotliwością pracy układu. Przy sześciokrotnym zmniejszeniu częstotliwości moc statyczna spadła o ok. 6,5 [%]. Przez tak małą zmianę tej mocy oraz trudności w jej uzyskiwaniu (moc statyczna odnosi się zawsze tylko do całego układu FPGA) zdecydowano się w dalszych badaniach brać pod uwagę tylko moc dynamiczną. Projektant wykorzystujący układy FPGA ma bardzo znikomy wpływ na moc statyczną tych układów. Zależy ona głownie od technologii wytworzenia układu.

Porównując moc całkowitą traconą w układzie FPGA można zauważyć dużą redukcję zapotrzebowania na moc, wraz z redukcją częstotliwości zegara. Przy sześciokrotnej redukcji częstotliwości uzyskano ponad pięciokrotną redukcję zapotrzebowania na moc. Wynika to z zależności przedstawionej w rozdziale 4.7 Energooszczędność systemu, na stronie 34.

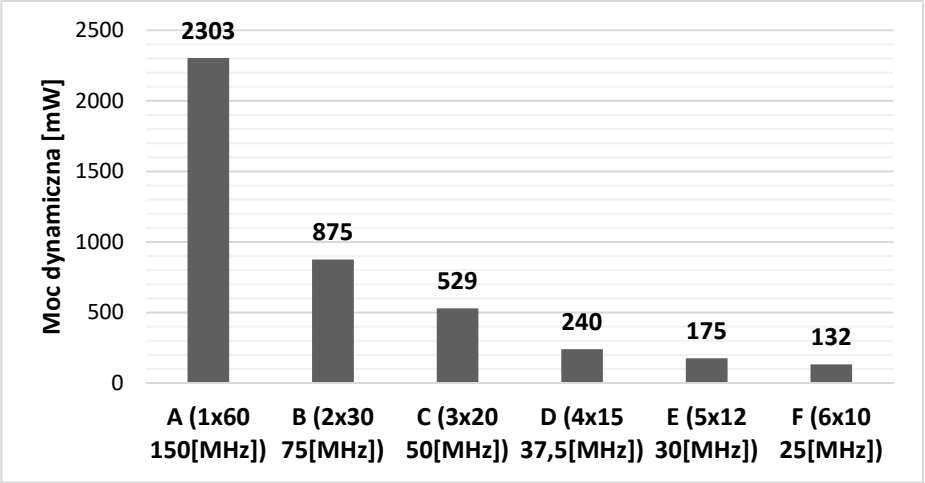


Rys. 89. Zużycie mocy przez układ FPGA, w konfiguracjach wykonujących tą samą prace w tym samym czasie

Kolejnym eksperymentem było porównanie mocy dynamicznej, zużywanej tylko przez zaprojektowany system. W poprzednich pomiarach moc zużywał również układ MMCM. Moc ta była na poziomie od 112 [mW] do 126 [mW]. Pomiary przedstawiono na Rys. 90. W tym eksperymencie uzyskano ponad 17-krotną redukcję mocy dynamicznej przy sześciokrotnej redukcji częstotliwości.

Następnie sprawdzono zależność między zużywaną mocą a częstotliwością przy różnej liczbie zadań. Na pierwszym wykresie (Rys. 91), oznaczonym od A do F, zaprezentowano dane z poprzedniego 60-cio zadaniowego scenariusza. Do wykresu dodano również pomiary dla 50-ciu, 40-tu oraz 30-tu zadań. Częstotliwość w dodatkowych scenariuszach została tak dobrana, aby przy każdej serii system wykonywał tę samą pracę w tym samym czasie. Jak można zaobserwować, zależność mocy od częstotliwości dąży do zależności liniowej. W scenariuszach o większej liczbie

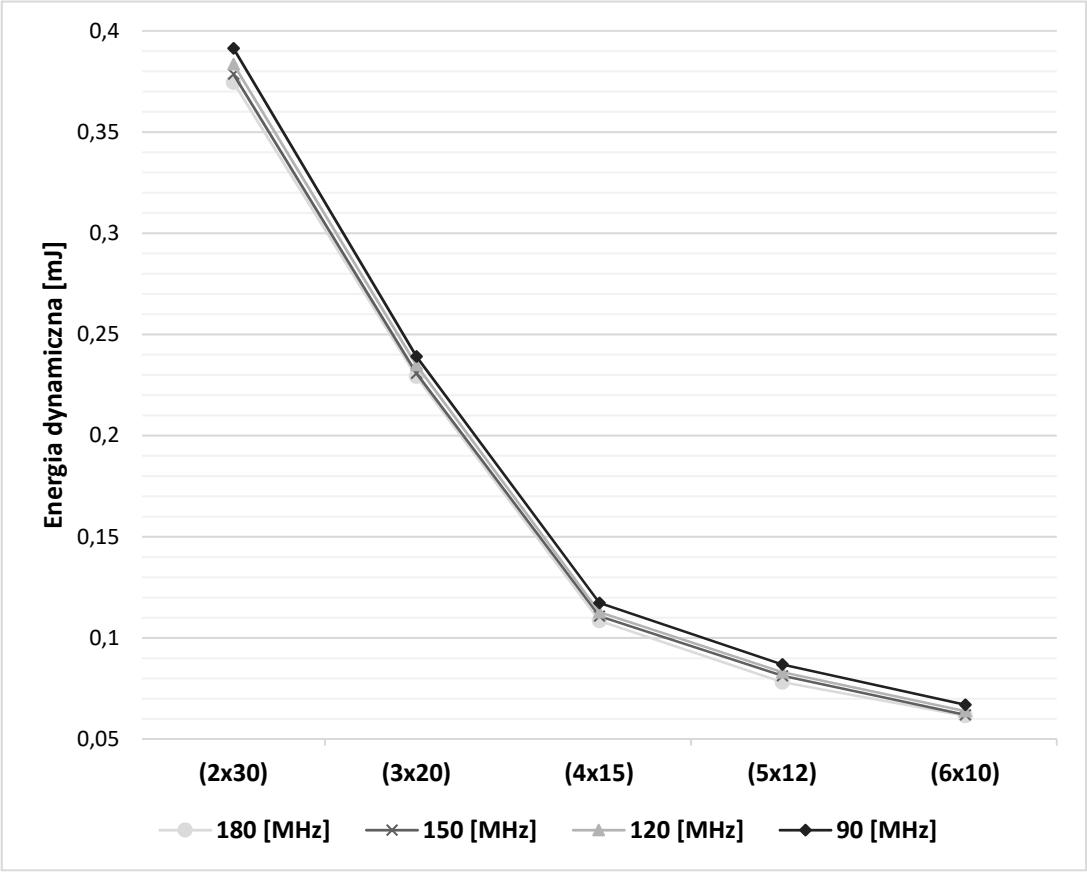
wątków, zależność ta jest bardziej stroma, ponieważ zwiększa się ilość zasobów potrzebnych do wygenerowania systemu oraz zwiększa się aktywność przełączeniowa układu (parametr α ze wzoru (4), strona 34).



Rys. 90. Zużycie mocy przez system, w konfiguracjach wykonujących tą samą prace w tym samym czasie

Rys. 91. Porównanie zużycia mocy, dla systemów przetwarzających różną liczbę zadań

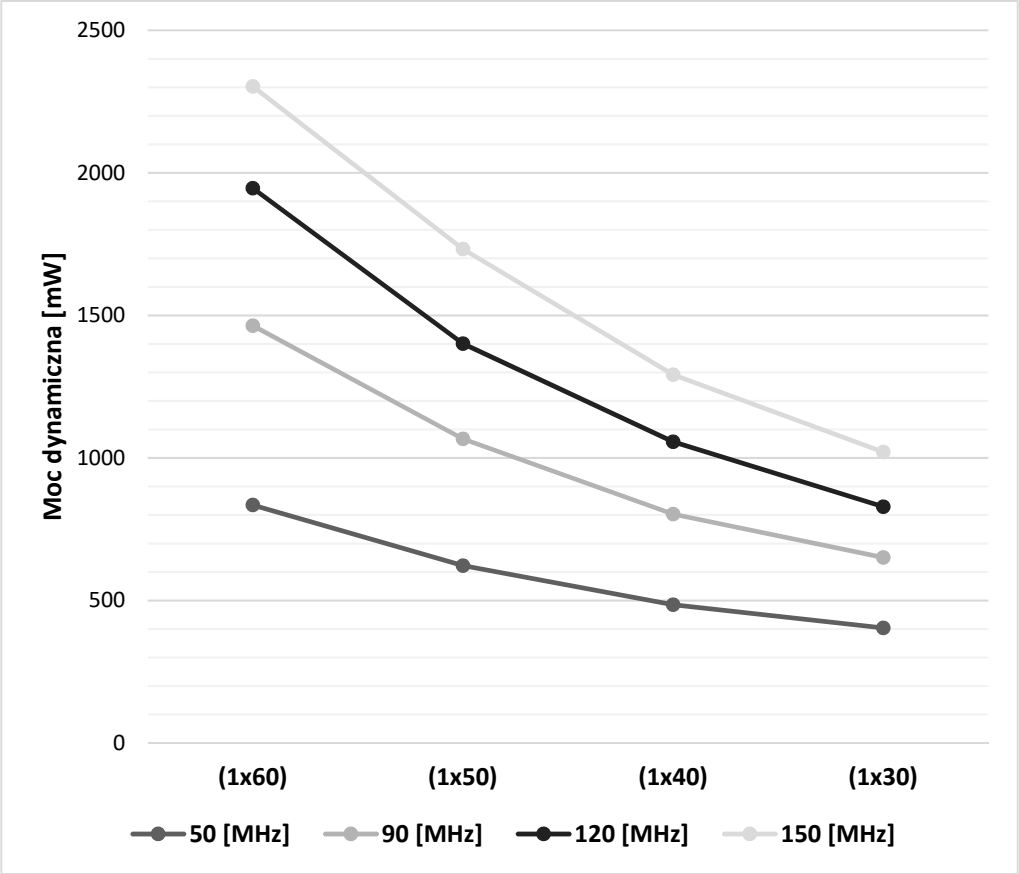
W kolejnym eksperymencie, porównano energię potrzebną do wykonania 60-ciu zadań o parametrze $MaxI_i$ (wzór (20), strona 129) równym 1000. Porównanie to nie uwzględnia wymagań czasowych zadań. Zadania w scenariuszu 2x30 będą wykonywały się dwa razy szybciej niż w scenariuszu 4x15 przy tej samej częstotliwości. Przy obliczaniu energii brany jest pod uwagę różny czas wykonywania zadań. Badanie to ma na celu sprawdzić, czy bardziej opłacalne pod względem energetycznym jest szybsze wykonywanie zadań, zużywając chwilo większą moc, czy dłuższe zużywające jej chwilowo mniej. Wyniki przedstawiono na Rys. 92. Można na nim zaobserwować, że zużycie energii jest bardzo zbliżone. Jednak najbardziej energooszczędną metodą jest szybsze wykonywanie zadań (przy wyższej częstotliwości) przy wyższym zużyciu mocy.



Rys. 92. Energia potrzebna na wykonanie 60 zadań o T_i = 1000, bez uwzględnienia wymagań czasowych tych zadań

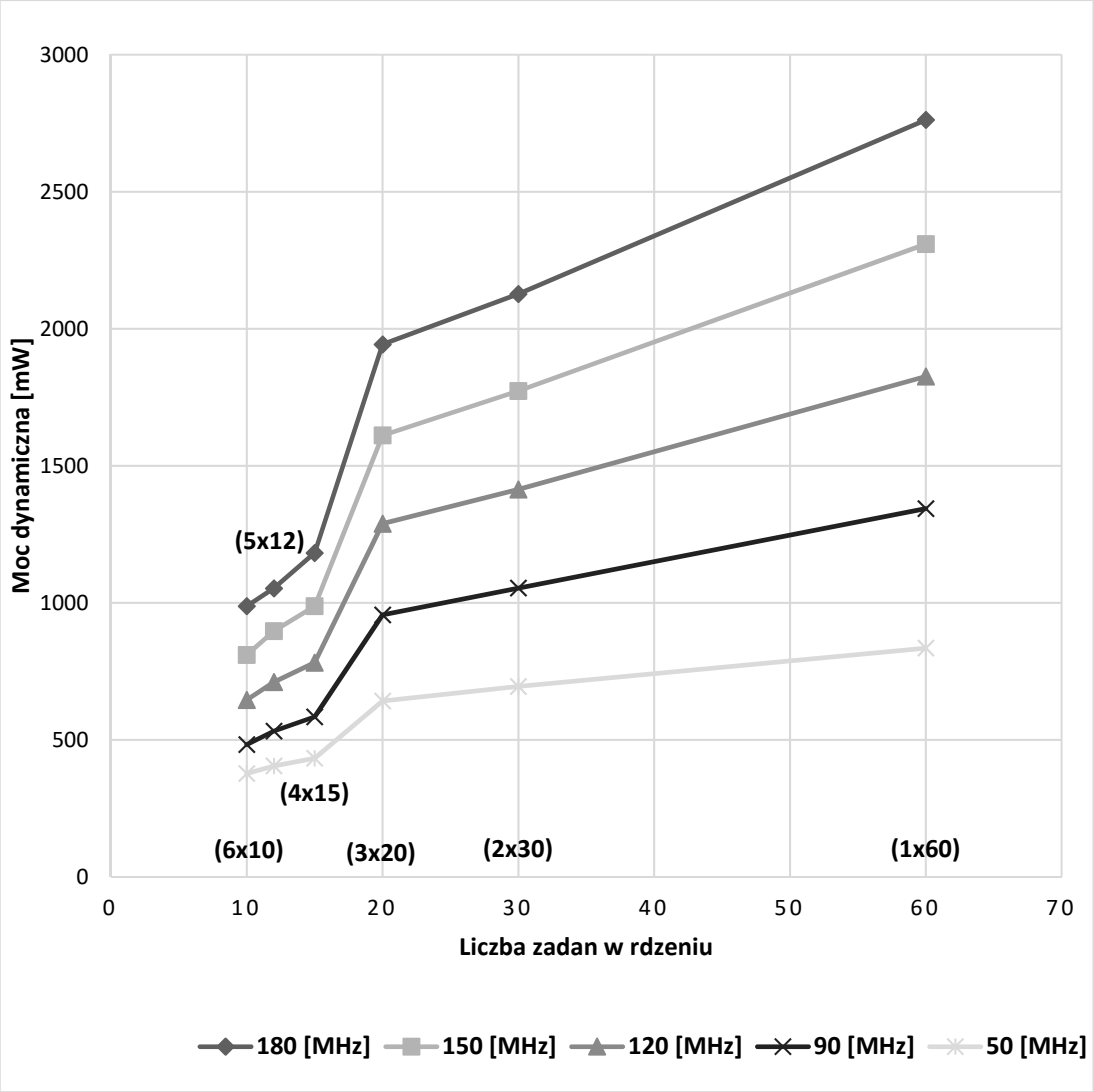
Następnie porównano między sobą zużycie mocy dynamicznej w konfiguracjach jednordzeniowych przy różnych częstotliwościach pracy zegara. Wyniki pomiarów

zaprezentowano na Rys. 93. Zużycie mocy rośnie wraz z częstotliwością zegara oraz rozmiarem rdzenia. Przy większych rdzeniach charakterystyka ta jest coraz bardziej stroma, wynika to ze zwiększonej liczby przełączeń (parametr α).



Rys. 93. Zużycie mocy dynamicznej w konfiguracjach jednordzeniowych

Ostatnim badaniem było porównanie zużycia mocy dynamicznej w różnych konfiguracjach oraz różnych częstotliwościach pracy 60-cio zadaniowego systemu. Pomiary przedstawiono na Rys. 94. Po przekroczeniu 15 zadań w jednym rdzeniu można zaobserwować zwiększone zapotrzebowanie na moc dynamiczną. Porównując wykres ten do wykresu zużycia zasobów (Rys. 84, strona 145) stwierdzono, że nie ma na to wpływu zwiększona liczba zasobów, ponieważ te zmieniają się w sposób zbliżony do liniowego. Podwyższone zużycie mocy powyżej jakiegoś progu zasobów, musi więc mieć swoje źródło w strukturze układu FPGA.



Rys. 94. Porównanie zużycia mocy dynamicznej, w 60-cio zadaniowym systemie

Powyższe badania posłużyły również do opracowana metody szacowania zapotrzebowania na moc systemu, na podstawie konfiguracji oraz częstotliwości pracy systemu [75].

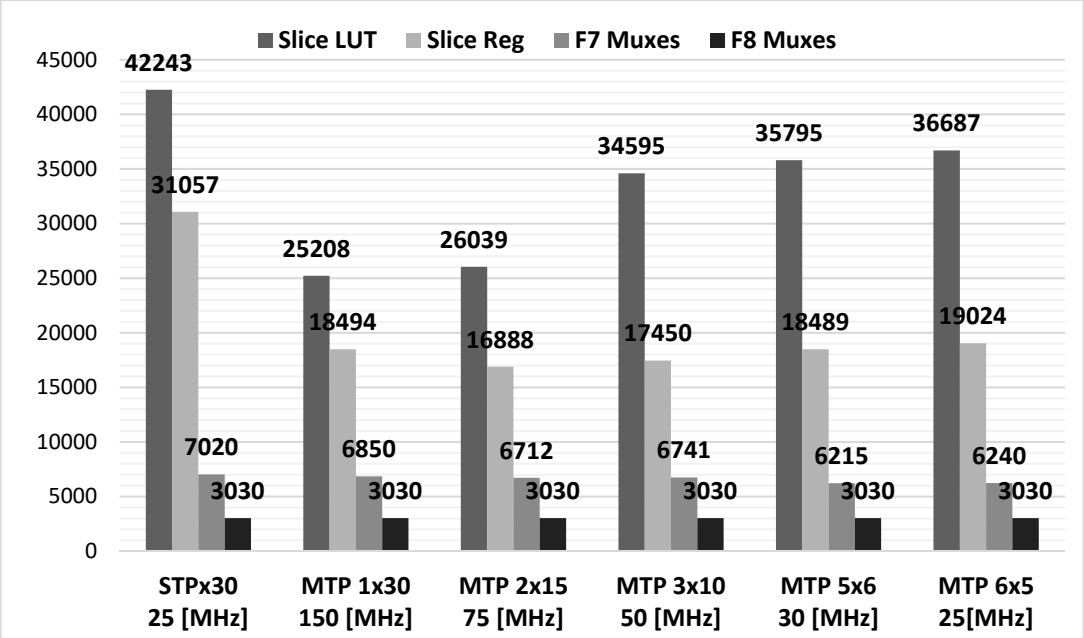
7.1.4 Porównanie systemu wielozadaniowego z klasycznym systemem jednozadaniowym

Po wykluczeniu komunikacji między zadaniami, omawiany pojedynczy system można w prosty sposób porównać do większej liczby jednozadaniowych układów. Przykładowo, prace 20 zadaniowego systemu omawianego w tej pracy (niewykorzystującego wymiany danych między wątkami) można porównać do 20

układów jednozadaniowych. Porównywany system składający się z jednozadaniowych układów również musi być przewidywalny czasowo, tzn. wykonywać instrukcje ze stałym czasem wykonania. W systemie jednozadaniowym nie można tego problemu rozwiązać za pomocą metody przeplotu wątków. Aby rozwiązać ten problem zdecydowano, że każda instrukcja będzie wykonywana kilka taktów zegara. Mikroarchitektura ta, przypomina więc klasyczny procesor niekorzystający z przetwarzania potokowego[65].

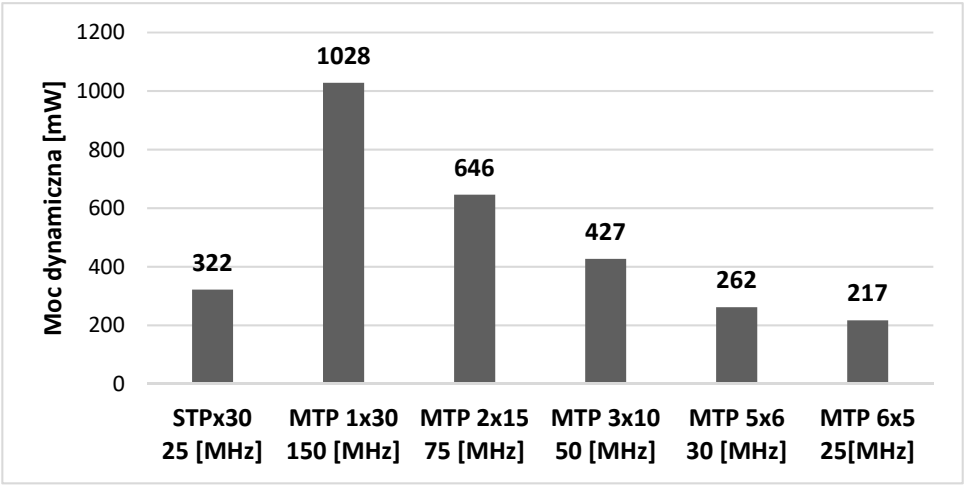
Wykorzystując mikroarchitekturę opracowanego systemu zaprojektowano układ wykonujący jedno zadanie i przetwarzający każdą instrukcję w ciągu 5 cykli zegara. Wersja jednozadaniowa (STP), względem wielozadaniowej (MTP) (Rys. 19, strona 43), pozbawiona jest etapów potoku przełączających wspólne dla kilku wątków zasoby. Posiada wiec on 7 etapów wykonywania instrukcji: IF, ID, SHIFT, ALU, EXE, MEM oraz WB. Etapy SHIFT oraz ALU zostały, aby zapewnić podobną maksymalną częstotliwość taktowania oraz kompatybilność z wersją MTP. Aby uniknąć hazardów oraz zapewnić przewidywalność czasową układu STP, zadanie w takim układzie nie może być przetwarzane wcześniej niż raz na 5 cykli zegara.

W pierwszym badaniu porównano zasoby obydwu systemów przetwarzających 30 zadań. Przygotowane scenariusze testów przypominały scenariusz z rozdziału 7.1.1 Scenariusz testowy (strona 141). Każdy z nich wykonywał tę samą pracę w tym samym czasie. Równoległość pracy scenariuszy była rekompensowana przez redukcję częstotliwości pracy systemu. Wyniki pomiarów zaprezentowano na Rys. 95. Można na nim zaobserwować, że zasoby potrzebne do implementacji układu STP są większe niż MTP w każdej testowanej konfiguracji, mimo dodatkowych modułów w MTP pozwalających na komunikację między zadaniami. Wyniki te spowodowane są tym, że układy STP nie wykorzystują żadnych wspólnych zasobów pomiędzy sobą, każdy z nich jest izolowany od innych. W wykonanych badaniach można też zaobserwować zmniejszone zapotrzebowanie na zasoby w scenariuszu MTP 2x15, względem charakterystyk pozostałych MTP. Wynika ona z optymalizacji zastosowanej przez narzędzie syntezy, która był możliwa dla tej konfiguracji.



Rys. 95. Porównanie zużycia zasobów między 30-sto zadaniowym systemem STP, a MTP

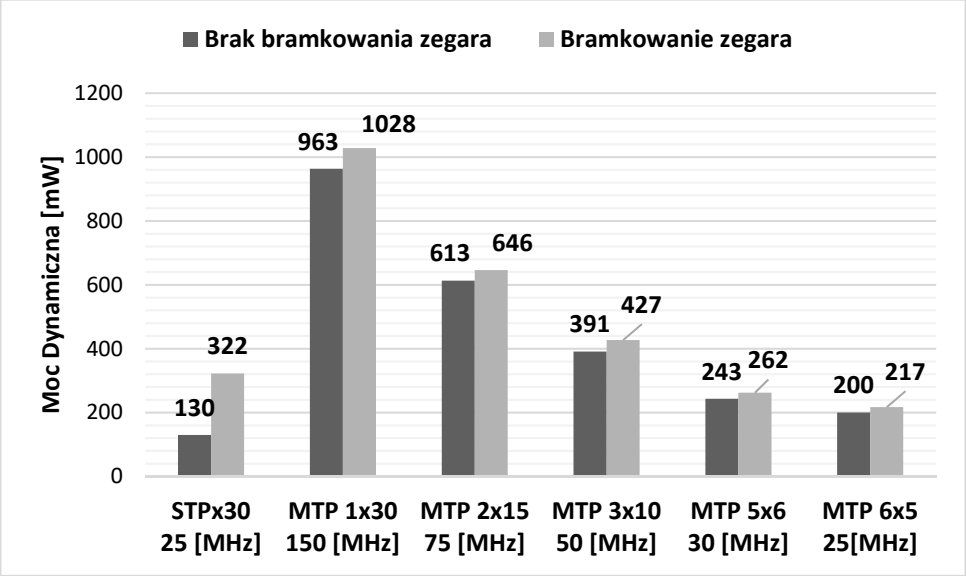
W kolejnym badaniu sprawdzono zapotrzebowanie na moc dynamiczną testowanych scenariuszy. Wyniki eksperymentów zaprezentowano na Rys. 96. Pod względem zużycia mocy, STP jest bardziej oszczędne od trzech konfiguracji MTP pracujących z najwyższą częstotliwością pracy zegara. Konfiguracje MTP pracujące z częstotliwością 30 [MHz] oraz 25 [MHz] zużywają mniej mocy dynamicznej niż STP.



Rys. 96. Porównanie zużycia mocy dynamicznej między 30-sto zadaniowym systemem STP, a MTP

Procesor STP zużywa stosunkowo dużo mocy, ponieważ nie wszystkie jego etapy przetwarzania instrukcji są wykorzystywane w każdym cyklu zegara, a wszystkie z nich generują przełączenia sygnałów, które prowadzą do pochłaniania mocy. Moc tą można zredukować stosując mechanizmy bramkowania zegara w niewykorzystywanych modułach procesora [77]. Niestety w układach MTP takie mechanizmy nie przyniosą dużych oszczędności, ponieważ w ich mikroarchitekturze każdy etap przetwarzania jest wykorzystywany w każdym takcie zegara [78]. Bramkowane mogłyby być tylko niezależne zasoby każdego wątku, takie jak pamięć i rejestry ogólnego przeznaczenia.

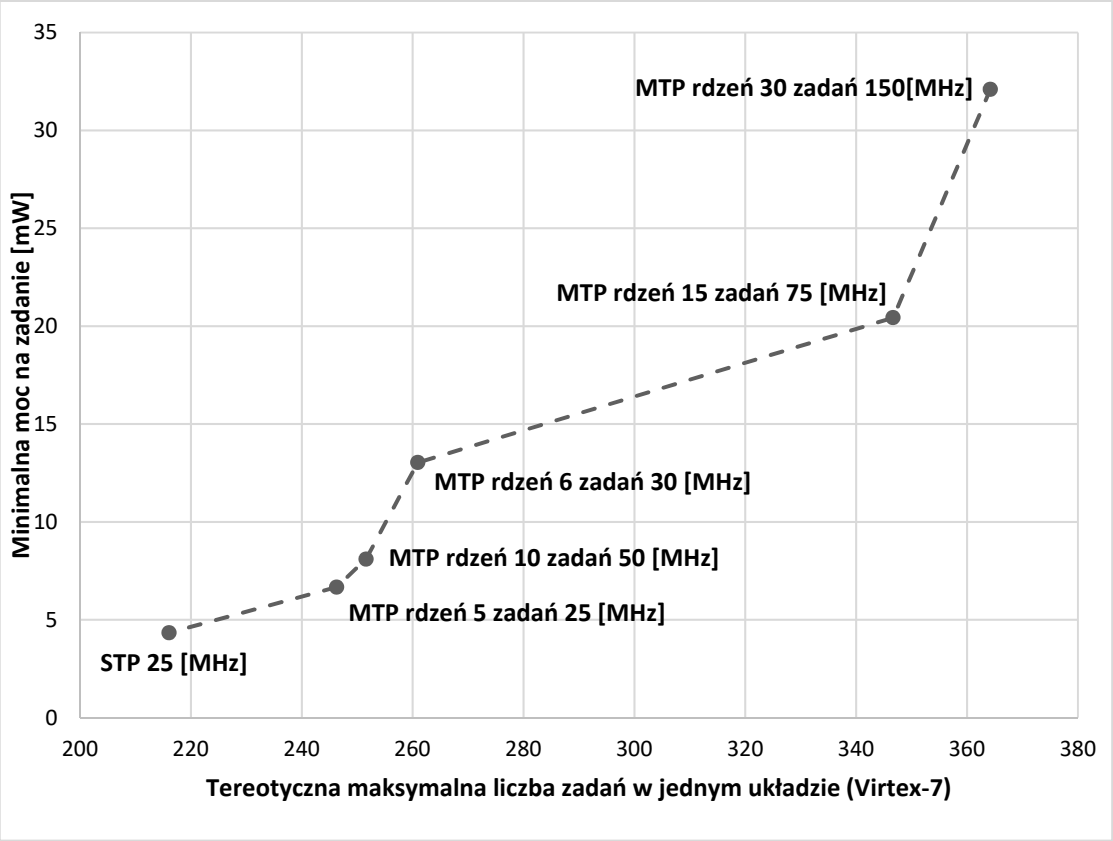
W literaturze [78] przeprowadzono również testy wpływu technik bramkowania zegara [79], [80] na systemy STP oraz MTP. Przyrost zasobów związanych z tą metodą jest niewielki i nie przekracza 1 [%]. Oszczędności związane z użyciem metod bramkowania zegara przedstawiono na Rys. 97. Największe oszczędności (wynoszące prawie 60 [%]) uzyskano w systemie STP. Porównując wyniki z systemem MTP, system STP wykorzystujący bramkowanie zegara ma najmniejsze zużycie mocy i energii.



Rys. 97. Porównanie oszczędności uzyskanych za pomocą metody bramkowania zegara

Na podstawie powyższych badań stwierdzono, że systemy posiadające większe zapotrzebowanie na zasoby są bardziej energooszczędne. Bazując na liczbie zasobów, układu FPGA Virtex-7, wyrysowano zależność maksymalnej liczby zadań jaką może

pomieścić ten układ, od mocy jaka jest pochłaniana przez jedno zadanie. Zależność tą przedstawiono na Rys. 98. Na rysunku tym przedstawione są konfiguracje o najmniejszym zapotrzebowaniu na moc, wykorzystujące mechanizm bramkowania zegara.



Rys. 98. Zależność między minimalną wymaganą mocą na zadanie, a maksymalną liczbą zadań w układzie

7.2 Weryfikacja przewidywalności czasowej systemu

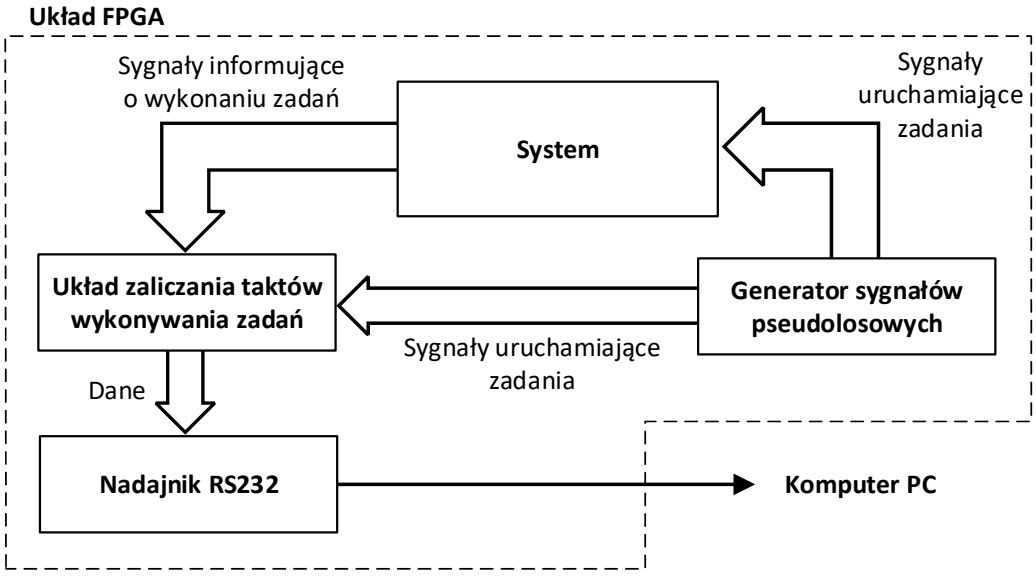
Przewidywalność czasową zaproponowanej mikroarchitektury sprawdzono przy użyciu autorskiego środowiska pomiarowego. Ze względu na dużą liczbę połączeń (każde z testowanych zadań wymaga co najmniej dwóch sygnałów) zdecydowano się całość środowiska zaimplementować w pojedynczym układzie FPGA.

7.2.1 Opis środowiska pomiarowego

Schemat środowiska przedstawiono na Rys. 99. Składa się ono z 4 modułów. Pierwszym z nich jest zaprojektowany w tej pracy wielozadaniowy system. Zgodnie z koncepcją systemu (4.1

Charakterystyka systemu - strona 22) zadania uruchamiane są przez sygnały zewnętrzne. Sygnały te generowane są przez generator sygnałów pseudolosowych zaimplementowanym przy użyciu rejestru LFSR.

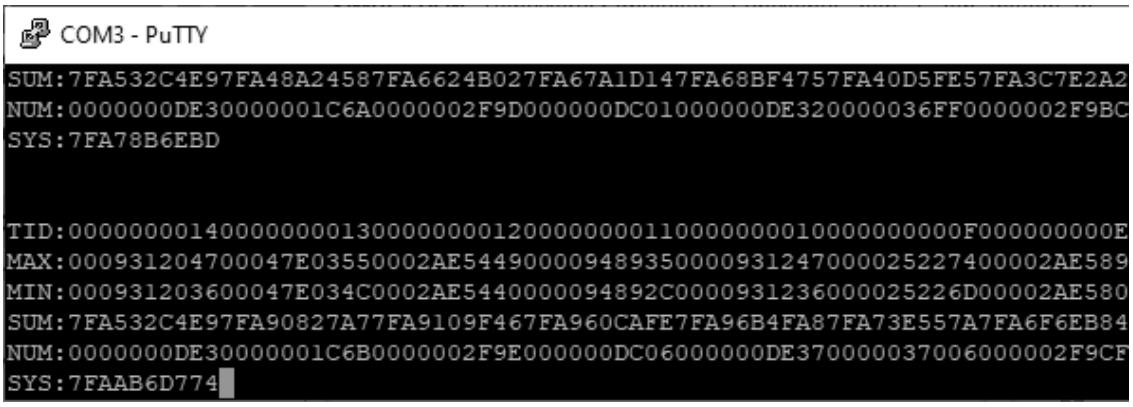
Następny moduł (Układ zliczania taktów wykonywania zadań) zlicza liczbę taktów zegara pomiędzy sygnałem uruchamiającym zadanie, a sygnałem zwrotnym pochodzącym z systemu sygnalizującym wykonanie zadania. Moduł ten przechowuje maksymalną i minimalną liczbę zliczonych taktów zegara od momentu uruchomienia systemu. Ze względu na problem synchronizacji domen zegarowych zdecydowano się nie wprowadzać kolejnego zegara taktującego moduły pomiarowe.



Rys. 99. Schemat ideowy środowiska pomiarowego

Ostatni moduł nadajnika RS232 zamienia transmisję równoległą na szeregową. Transmisja jest inicjalizowana po wciśnięciu odpowiedniego przycisku na płycie prototypowej z układem FPGA. Oprócz minimalnej (MIN) i maksymalnej (MAX) liczby taktów zegara potrzebnych do wykonania każdego zadania, wysyłane są również takie

informacje jak liczba wykonań każdego zadania (NUM), całkowita liczba taktów potrzebnych na wielokrotne wykonanie zadań (SUM) oraz całkowita liczba taktów trwania eksperymentu (SYS). Takie dane pozwolą obliczyć średni czas wykonania zadania.



Rys. 100. Fragment danych pozyskiwanych przy użyciu stanowiska pomiarowego

Na Rys. 100 przedstawiono fragment odebranych danych. Dane odczytywane są w 10 znakowych paczkach, zapisywanych w systemie heksadecymalnym. Informacje odczytywane są 10 znakowymi kolumnami. Wiersz pierwszy (TID) to identyfikator zadania ThID. Wszystkie dane w kolumnie poniżej ThID dotyczą tego zadania.

7.2.2 Przewidywalność czasowa mikroarchitektury systemu

Pierwszym etapem weryfikacji przewidywalności czasowej było porównanie teoretycznego i praktycznego czasu wykonania zadań w systemie. W etapie tym sprawdzono czy zaproponowany model obliczenia czasu wykonania zadań jest prawidłowy.

Proces weryfikacji przewidywalności czasowej mikroarchitektury polegał na uruchomieni 20 zadań o znacznie różniących się parametrach C_i i M_i w dwóch konfiguracjach systemu: jednordzeniowej (1x20) oraz dwurdzeniowej (2x10). W etapie tym zweryfikowany ma być tylko model obliczeń, dlatego zdecydowano się nie harmonogramować zadań, tylko użyć prostego i krótkiego cyklu przeplotu. W konfiguracjach jednordzeniowych CP ma długość 20, a każde z zadań parametr TW równy 20. Natomiast w konfiguracji jednordzeniowej CP ma długość 10, a każdy parametr TW zadania jest równy 10. Taki cykl przeplotu powoduje, że każde z 20 zadań

spełnia założenia zadań typu SHT. Spowoduje to, że możliwe będzie obliczenie czasów wykonywania zadań bez użycia symulatora, dzięki czemu możliwe będzie zweryfikowanie działania systemu, a nie symulatora. Zaproponowano 4 scenariusze w zależności od typu przetwarzanych zadań:

- 20 niewspółpracujących (NCT)
- 20 zadań współpracujących o stałym czasie wykonania instrukcji wymiany danych (CTHM)
- 20 zadań współpracujących (CT)
- Zadania mieszane:
    - 10 zadań NCT
    - 5 zadań (CTHM)
    - 5 zadań CT

Wszystkie scenariusze zaimplementowano oraz uruchomiono w środowisku pomiarowym na czas około jednej godziny z częstotliwością pracy zegara wynoszącym 110 [MHz]. Następnie zebrano dane i porównano je z teoretycznymi obliczeniami.

7.2.2.1 Zadania niewspółpracujące

W pierwszym eksperymencie zaproponowano scenariusz, w którym zadania nie wymieniają się ze sobą żadnymi danymi, ich parametry M_i będą więc równe 0. Parametry tych zadań umieszczono w Tab. 11.

Tab. 11. Parametry zadań - Zadania niewspółpracujące

Thid	C_i	M_i	HM	Konfiguracja 1	Konfiguracja 2
1	7	0	Nie	Rdzeń 0	Rdzeń 0
2	34	0	Nie	Rdzeń 0	Rdzeń 0
3	64	0	Nie	Rdzeń 0	Rdzeń 0
4	124	0	Nie	Rdzeń 0	Rdzeń 0
5	244	0	Nie	Rdzeń 0	Rdzeń 0
6	484	0	Nie	Rdzeń 0	Rdzeń 0
7	964	0	Nie	Rdzeń 0	Rdzeń 0
8	1924	0	Nie	Rdzeń 0	Rdzeń 0
9	3844	0	Nie	Rdzeń 0	Rdzeń 0

Thid	C_i	M_i	HM	Konfiguracja 1	Konfiguracja 2
10	7684	0	Nie	Rdzeń 0	Rdzeń 0
11	15364	0	Nie	Rdzeń 0	Rdzeń 1
12	30724	0	Nie	Rdzeń 0	Rdzeń 1
13	61444	0	Nie	Rdzeń 0	Rdzeń 1
14	121684	0	Nie	Rdzeń 0	Rdzeń 1
15	243364	0	Nie	Rdzeń 0	Rdzeń 1
16	486724	0	Nie	Rdzeń 0	Rdzeń 1
17	973444	0	Nie	Rdzeń 0	Rdzeń 1
18	1945684	0	Nie	Rdzeń 0	Rdzeń 1
19	7536724	0	Nie	Rdzeń 0	Rdzeń 1
20	15073444	0	Nie	Rdzeń 0	Rdzeń 1

Maksymalną i minimalną liczbę cykli zegara potrzebnych do zakończenia pracy zegara obliczono za pomocą wzorów (29) oraz (30). Wzory te zostały otrzymane po połączeniu wzorów (19), (21), (22), (23) oraz (24) (Symulacja wykonywania zadań – strona 126) przy zachowaniu warunku, że wszystkie zadania są zadaniami SHT.

$$MaxExeTicks_i = \left[C_i + M_i \left(\left\lceil \frac{M_{dur}}{TW_{SHT_i}} \right\rceil\right)\right] \cdot TW_{SHT_i} + TW_{SHT_i} + 3 \qquad (29)$$

gdzie:

$MaxExeTicks_i$ – Maksymalna liczba cykli zegara potrzebnych do wykonania zadania

M_{dur} – Liczba standardowych instrukcji, potrzebnych do wykonania instrukcji wymiany danych

TW_{SHT_i} – Parametr TW_i zadania typu SHT dla którego przeprowadzane są obliczenia

Następnie obliczono maksymalną i minimalną liczbę taktów zegara potrzebnych do wykonania każdego zadania w konfiguracji 1 (1x20) (M_{dur} = 11, TW zadań równe 20).

Wyniki umieszczono w Tab. 12. Do tabeli dopisano również dane uzyskane z pomiarów za pomocą środowiska pomiarowego. W ten sam sposób uzyskano dane dla konfiguracji 2 (2x10) (M_{dur} = 11, TW zadań równe 10) w Tab. 13. W ostatnich kolumnach tabel pokazano różnice między obliczeniami, a maksymalnym i minimalnym pomiarem.

$$MinExeTicks_i = \left[C_i + M_i \left(\left\lceil \frac{M_{dur}}{TW_{SHT_i}} \right\rceil\right)\right] \cdot TW_{SHT_i} + 4 \qquad (30)$$

gdzie:

$MinExeTicks_i$ – Maksymalna liczba cykli zegara potrzebnych do wykonania zadania

M_{dur} – Liczba standardowych instrukcji, potrzebnych do wykonania instrukcji wymiany danych

TW_{SHT_i} – Parametr TW_i zadania typu SHT dla którego przeprowadzane są obliczenia

Tab. 12. Porównanie obliczeń i pomiarów - Zadania niewspółpracujące, konfiguracja 1 (1x20)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
1	144	163	144	163	153,3049301	800458026	0	0
2	684	703	684	702	693,5166668	350778925	0	1
3	1284	1303	1285	1303	1294,47541	208031391	1	0
4	2484	2503	2484	2503	2493,179381	118144291	0	0
5	4884	4903	4884	4903	4892,695313	62360697	0	0
6	9684	9703	9684	9702	9693,136364	32154735	0	1
7	19284	19303	19285	19303	19294,21702	16355763	1	0
8	38484	38503	38484	38503	38493,27541	8255213	0	0
9	76884	76903	76884	76901	76893,23529	4141140	0	2
10	153684	153703	153684	153702	153693,4087	2075081	0	1
11	307284	307303	307285	307303	307294,5924	1038280	1	0
12	614484	614503	614484	614503	614493,6524	519342	0	0

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
13	1228884	1228903	1228884	1228903	1228893,273	259731	0	0
14	2433684	2433703	2433685	2433702	2433690,429	131167	1	1
15	4867284	4867303	4867285	4867303	4867294,553	65587	1	0
16	9734484	9734503	9734484	9734503	9734493,154	32794	0	0
17	19468884	19468903	19468884	19468903	19468894,09	16397	0	0
18	38913684	38913703	38913684	38913702	38913694,05	8203	0	1
19	150734484	150734503	150734485	150734503	150734493,7	2117	1	0
20	301468884	301468903	301468884	301468903	301468894,2	1058	0	0

Tab. 13. Porównanie obliczeń i pomiarów - Zadania niewspółpracujące, konfiguracja 2 (2x10)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
1	74	83	74	83	78,29451218	799828927	0	0
2	344	353	344	353	348,6666667	494528371	0	0
3	644	653	644	653	648,3237409	338951888	0	0
4	1244	1253	1244	1253	1248,768182	214588245	0	0
5	2444	2453	2444	2453	2448,383333	117048135	0	0
6	4844	4853	4844	4853	4848,627451	62181821	0	0
7	9644	9653	9644	9653	9648,384615	31700536	0	0
8	19244	19253	19244	19253	19248,71915	16372804	0	0
9	38444	38453	38444	38453	38448,3037	8229946	0	0
10	76844	76853	76844	76853	76848,52941	4145454	0	0
11	153644	153653	153644	153653	153648,5478	2077243	0	0
12	307244	307253	307244	307253	307248,7	1039361	0	0
13	614444	614453	614444	614453	614448,165	519856	0	0
14	1216844	1216853	1216844	1216853	1216848,635	262557	0	0
15	2433644	2433653	2433645	2433652	2433647,571	131303	1	1
16	4867244	4867253	4867244	4867253	4867248,701	65655	0	0
17	9734444	9734453	9734444	9734453	9734448,381	32828	0	0

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
18	19456844	19456853	19456844	19456853	19456848,82	16424	0	0
19	75367244	75367253	75367244	75367253	75367248,18	4240	0	0
20	150734444	150734453	150734444	150734453	150734448,8	2120	0	0

Wszystkie wartości w kolumnach „Porównanie" Tab. 12 oraz Tab. 13 są dodatnie. Oznacza to, że zadania wykonały się szybciej niż obliczony maksymalny czas wykonania oraz wolniej niż obliczony minimalny czas wykonania. Różnice te jednak nie są duże (maksymalna różnica to 2 takty zegara) i spowodowane są niewylosowaniem najbardziej skrajnych warunków startu pracy zadań przez generator sygnałów pseudolosowych.

7.2.2.2 Zadania współpracujące CTHM

W kolejnym eksperymencie porównano teoretyczne i praktyczne wykonania zadań współpracujących o stałym czasie wymiany danych między zadaniami (CTHM) (Typy obsługiwanych wątków – strona 78). Dane zadań umieszczono w Tab. 14.

Tab. 14. Parametry zadań - Zadania współpracujące CTHM

Thid	C_i	M_i	HM	Konfiguracja 1	Konfiguracja 2
1	9	1	Tak	Rdzeń 0	Rdzeń 0
2	36	10	Tak	Rdzeń 0	Rdzeń 0
3	66	20	Tak	Rdzeń 0	Rdzeń 0
4	126	40	Tak	Rdzeń 0	Rdzeń 0
5	246	80	Tak	Rdzeń 0	Rdzeń 0
6	486	160	Tak	Rdzeń 0	Rdzeń 0
7	966	320	Tak	Rdzeń 0	Rdzeń 0
8	1926	640	Tak	Rdzeń 0	Rdzeń 0
9	3846	1280	Tak	Rdzeń 0	Rdzeń 0
10	7686	2560	Tak	Rdzeń 0	Rdzeń 0
11	15066	5020	Tak	Rdzeń 0	Rdzeń 1
12	30126	10040	Tak	Rdzeń 0	Rdzeń 1
13	60246	20080	Tak	Rdzeń 0	Rdzeń 1

Thid	C_i	M_i	HM	Konfiguracja 1	Konfiguracja 2
14	120486	40160	Tak	Rdzeń 0	Rdzeń 1
15	240966	80320	Tak	Rdzeń 0	Rdzeń 1
16	481926	160640	Tak	Rdzeń 0	Rdzeń 1
17	963846	321280	Tak	Rdzeń 0	Rdzeń 1
18	1927686	642560	Tak	Rdzeń 0	Rdzeń 1
19	3855366	1285120	Tak	Rdzeń 0	Rdzeń 1
20	7710726	2570240	Tak	Rdzeń 0	Rdzeń 1

Następnie obliczono maksymalną i minimalną liczbę taktów potrzebną do wykonania każdego z zadań (wzór (29) i (30)). Wyniki obliczeń scenariusza 1 (1x20) (M_{dur} = 53, TW zadań równe 20) oraz scenariusza 2 (2x10) (M_{dur} = 53, TW zadań równe 10) umieszczono w Tab. 15 oraz Tab. 16.

Tab. 15. Porównanie obliczeń i pomiarów - Zadania współpracujące CTHM, konfiguracja 1 (1x20)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	**Max**	**Min**	**Max**	**Średnie wykonanie**	**Liczba wykonań**	**Różnica min**	**Różnica max**
1	244	263	244	263	253,158494	627934786	0	0
2	1324	1343	1324	1341	1333,032828	217742710	0	2
3	2524	2543	2525	2542	2534,076596	129216002	1	1
4	4924	4943	4926	4943	4934,744001	68731916	2	0
5	9724	9743	9724	9743	9732,907693	35740596	0	0
6	19324	19343	19324	19341	19333,25532	18459428	0	2
7	38524	38543	38525	38542	38533,91111	9278808	1	1
8	76924	76943	76926	76943	76934,53974	4667376	2	0
9	153724	153743	153724	153743	153732,98	2339809	0	0
10	307324	307343	307324	307341	307333,551	1171430	0	2
11	602524	602543	602525	602542	602533,8	597668	1	1
12	1204924	1204943	1204926	1204943	1204934,69	298961	2	0
13	2409724	2409743	2409724	2409743	2409733,344	149507	0	0
14	4819324	4819343	4819324	4819341	4819332,683	74757	0	2

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
15	9638524	9638543	9638525	9638542	9638533,732	37381	1	1
16	19276924	19276943	19276926	19276943	19276934,97	18691	2	0
17	38553724	38553743	38553724	38553743	38553733,74	9345	0	0
18	77107324	77107343	77107324	77107341	77107333,49	4672	0	2
19	154214524	154214543	154214525	154214542	154214534,2	2336	1	1
20	308428924	308428943	308428926	308428943	308428934,9	1168	2	0

Tab. 16. Porównanie obliczeń i pomiarów - Zadania współpracujące CTHM, konfiguracja 2 (2x10)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
1	154	163	154	163	158,2803532	1205668630	0	0
2	964	973	964	973	968,6534653	448022556	0	0
3	1864	1873	1864	1873	1868,567308	276798093	0	0
4	3664	3673	3664	3673	3668,636364	146383606	0	0
5	7264	7273	7264	7273	7268,470588	75409736	0	0
6	14464	14473	14464	14473	14468,88889	39922801	0	0
7	28864	28873	28864	28873	28868,44444	19961401	0	0
8	57664	57673	57664	57673	57668,63934	10021773	0	0
9	115264	115273	115264	115273	115267,5882	5027315	0	0
10	230464	230473	230464	230473	230468,6957	2519134	0	0
11	451864	451873	451864	451873	451868,3	1285758	0	0
12	903664	903673	903664	903673	903668,653	643069	0	0
13	1807264	1807273	1807264	1807273	1807268,417	321633	0	0
14	3614464	3614473	3614464	3614473	3614468,721	160828	0	0
15	7228864	7228873	7228864	7228873	7228868,345	80417	0	0
16	14457664	14457673	14457664	14457673	14457668,48	40209	0	0
17	28915264	28915273	28915264	28915273	28915268	20104	0	0
18	57830464	57830473	57830464	57830473	57830468,73	10052	0	0
19	115660864	115660873	115660864	115660873	115660868,3	5026	0	0

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
20	231321664	231321673	231321664	231321673	231321668,6	2513	0	0

W przypadku scenariusza jednordzeniowego wszystkie różnice porównania obliczeń i wartości praktycznej są dodatnie. Różnice na poziomie maksymalnie 2 taktów zegara pochodzą z niewylosowanych przez środowisko testowe najbardziej skrajnych przypadków. W scenariuszu dwurdzeniowym wszystkie różnice są równe 0. Oznacza to, że wszystkie obliczone wartości są równe wartością praktycznym i wszystkie skrajne przepadki zostały wylosowane.

7.2.2.3 Zadania współpracujące CT

Ostatnią grupą zadań, są zadania współpracujące o zmiennym czasie wymiany danych. Czas wymiany danych w takich zadaniach może być znacznie mniejszy niż maksymalny, co będzie powodowało dużą różnicę między maksymalnym a minimalnym czasem wykonania zadania. Parametry zadań oraz konfiguracje systemu przedstawiono w Tab. 17.

Tab. 17. Parametry zadań - Zadania współpracujące CT

Thid	C_i	M_i	HM	Konfiguracja 1	Konfiguracja 2
1	9	1	Nie	Rdzeń 0	Rdzeń 0
2	36	10	Nie	Rdzeń 0	Rdzeń 0
3	66	20	Nie	Rdzeń 0	Rdzeń 0
4	126	40	Nie	Rdzeń 0	Rdzeń 0
5	246	80	Nie	Rdzeń 0	Rdzeń 0
6	486	160	Nie	Rdzeń 0	Rdzeń 0
7	966	320	Nie	Rdzeń 0	Rdzeń 0
8	1926	640	Nie	Rdzeń 0	Rdzeń 0
9	3846	1280	Nie	Rdzeń 0	Rdzeń 0
10	7686	2560	Nie	Rdzeń 0	Rdzeń 0

Thid	C_i	M_i	HM	Konfiguracja 1	Konfiguracja 2
11	15066	5020	Nie	Rdzeń 0	Rdzeń 1
12	30126	10040	Nie	Rdzeń 0	Rdzeń 1
13	60246	20080	Nie	Rdzeń 0	Rdzeń 1
14	120486	40160	Nie	Rdzeń 0	Rdzeń 1
15	240966	80320	Nie	Rdzeń 0	Rdzeń 1
16	481926	160640	Nie	Rdzeń 0	Rdzeń 1
17	963846	321280	Nie	Rdzeń 0	Rdzeń 1
18	1927686	642560	Nie	Rdzeń 0	Rdzeń 1
19	3855366	1285120	Nie	Rdzeń 0	Rdzeń 1
20	7710726	2570240	Nie	Rdzeń 0	Rdzeń 1

W Tab. 18 umieszczono porównanie liczby taktów potrzebnych do wykonania zadań w konfiguracji 1 (1x20) (M_{dur} = 53, TW zadań równe 20). Maksymalną liczbę taktów potrzebną do wykonania zadania obliczono za pomocą wzoru (29). Zadania w tym scenariuszu nie są typu CTHM, co powoduje, że wymiana danych pomiędzy zadaniami może się wykonać tak samo szybko jak standardowa instrukcja. Dlatego minimalną liczbę taktów potrzebną do wykonania zadania należy obliczyć ze wzoru (31).

$$MinExeTicksCT_i = (C_i + M_i) \cdot TW_{SHT_i} + 4 \tag{31}$$

gdzie:

$MinExeTicksCT_i$ – Minimalna liczba cykli zegara potrzebnych do wykonania zadania typu CT

TW_{SHT_i} – Parametr TW_i zadania typu SHT dla którego przeprowadzane są obliczenia

W ostatnich kolumnach Tab. 18 (Porównanie) można zaobserwować ciekawą zależność, która pokazuje, że zadania wykonują się albo blisko swojego teoretycznego maksymalnego czasu wykonania lub blisko teoretycznego minimalnego czasu wykonania. Omawiana konfiguracja jest konfiguracją szczególną tj. w rdzeniu pracuje tyle samo zadań, ile w całym systemie i dodatkowo wszystkie zadania są zadaniami

współpracującymi. Prawo do wysłania zapytania na magistrale przez zadanie jest przyznawane cyklicznie, co 40 taktów (dwukrotność liczby zadań). Zapytanie jest więc synchronizowane z przetwarzanym zadaniem (40 jest wielokrotnością 20 przetwarzanych zadań). W takiej szczególnej konfiguracji zadania będą zawsze wysyłały informacje po takim samym czasie od momentu zapytania. Wszystkie pomiary mieszczą się w zakresie obliczonym na podstawie parametrów systemu i parametrów zadań.

Tab. 18. Porównanie obliczeń i pomiarów - Zadania współpracujące CT, konfiguracja 1 (1x20)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
1	204	263	224	263	242,092882	553034982	20	0
2	924	1343	1304	1337	1320,761307	191065904	380	6
3	1724	2543	2505	2538	2521,872341	112815296	781	5
4	3324	4943	4906	4939	4922,271999	60008136	1582	4
5	6524	9743	9707	9740	9723,523079	31204231	3183	3
6	12924	19343	19308	19341	19325,03405	16116470	6384	2
7	25724	38543	38509	38542	38524,93333	8101098	12785	1
8	51324	76943	76910	76943	76926,21646	4074971	25586	0
9	102524	153743	102527	102560	102545,6863	3060414	3	51183
10	204924	307343	204928	204961	204945,0067	1532122	4	102382
11	401724	602543	401729	401762	401746,2439	782426	5	200781
12	803324	1204943	803330	803363	803345,2076	391437	6	401580
13	1606524	2409743	1606524	1606563	1606542,643	195771	0	803180
14	3212924	4819343	3212924	3212959	3212940,978	97901	0	1606384
15	6425724	9638543	9638505	9638538	9638521,426	32636	3212781	5
16	12851324	19276943	19276906	19276939	19276922,26	16318	6425582	4
17	25702524	38553743	38553707	38553740	38553722,78	8159	12851183	3
18	51404924	77107343	77107308	77107341	77107325,06	4079	25702384	2
19	102809724	154214543	154214509	154214542	154214525,8	2039	51404785	1
20	205619324	308428943	308428910	308428943	308428926,4	1019	102809586	0

Obliczenia konfiguracji 2 (2x10) (M_{dur} = 53, TW zadań równe 10) umieszczono w Tab. 19. Konfiguracja ta jest bliższa standardowym przypadkom. Różnice w wartościach obliczonych i pochodzących z pomiarów rosną wraz z długością (parametrem C_i oraz M_i) zadania. Powodem tego jest to, że wraz z długością zadania spada liczba jego wykonań, czyli liczba losowań momentu startu zadania. Drugim powodem jest zwieszona liczba instrukcji wymiany danych, które w obliczeniach uwzględniają zawsze najgorszy, czyli najdłuższy przypadek wymiany danych. Wszystkie wartości praktyczne mieszczą się w zakresie obliczeń teoretycznych.

Warto również podkreślić tutaj różnice między zadaniami CT a CTHM. Porównując Tab. 15 i Tab. 16 z Tab. 18 oraz Tab. 19 można zauważyć, że maksymalne teoretyczne czasy wykonania zadań są takie same. Różnica między tymi zadaniami powoduje różnicę w minimalnym czasie wykonania zadania. Zadania CT mogą wykonać się szybciej niż CTHM, lecz różnica między maksymalnym a minimalnym czasem wykonania zadań CT jest większa.

Tab. 19. Porównanie obliczeń i pomiarów - Zadania współpracujące CT, konfiguracja 2 (2x10)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
1	104	163	119	152	135,517338	966348911	15	11
2	464	973	839	873	856,536672	382299393	375	100
3	864	1873	1634	1673	1651,97910	229106076	770	200
4	1664	3673	3234	3267	3250,10270	126521266	1570	406
5	3264	7273	6445	6479	6462,73845	66680126	3181	794
6	6464	14473	12846	12879	12865,66	34194936	6382	1594
7	12864	28873	25647	25680	25663,6	17097469	12783	3193
8	25664	57673	51248	51281	51264,8060	8680253	25584	6392
9	51264	115273	102434	102473	102452,232	4362802	51170	12800
10	102464	230473	204834	204867	204850,078	2184676	102370	25606
11	200864	451873	401635	401668	401651,679	1115052	200771	50205
12	401664	903673	803236	803269	803252,342	557685	401572	100404
13	803264	1807273	1606437	1606470	1606452,66	278917	803173	200803
14	1606464	3614473	3212838	3212871	3212854,44	139473	1606374	401602

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
15	3212864	7228873	6425649	6425682	6425663,41	69739	3212785	803191
16	6425664	14457673	12851250	12851283	12851266,6	34870	6425586	1606390
17	12851264	28915273	25702444	25702483	25702463,3	17435	12851180	3212790
18	25702464	57830473	51404844	51404879	51404861,7	8717	25702380	6425594
19	51404864	115660873	102809637	102809670	102809651	4358	51404773	12851203
20	102809664	231321673	205619238	205619271	205619254	2179	102809574	25702402

7.2.2.4 Zadania mieszane

W ostatnim eksperymencie postanowiono sprawdzić scenariusz zawierający różne typy zadań. Dane zadań przedstawiono w Tab. 20.

Tab. 20. Parametry zadań - Zadania mieszane

Thid	C_i	M_i	HM	Konfiguracja 1	Konfiguracja 2
1	7	0	Nie	Rdzeń 0	Rdzeń 0
2	36	10	Tak	Rdzeń 0	Rdzeń 0
3	64	0	Nie	Rdzeń 0	Rdzeń 0
4	126	40	Nie	Rdzeń 0	Rdzeń 0
5	244	0	Nie	Rdzeń 0	Rdzeń 0
6	486	160	Tak	Rdzeń 0	Rdzeń 0
7	964	0	Nie	Rdzeń 0	Rdzeń 0
8	1926	640	Nie	Rdzeń 0	Rdzeń 0
9	3844	0	Nie	Rdzeń 0	Rdzeń 0
10	7686	2560	Tak	Rdzeń 0	Rdzeń 0
11	15364	0	Nie	Rdzeń 0	Rdzeń 1
12	30126	10040	Nie	Rdzeń 0	Rdzeń 1
13	61444	0	Nie	Rdzeń 0	Rdzeń 1
14	120486	40160	Tak	Rdzeń 0	Rdzeń 1
15	243364	0	Nie	Rdzeń 0	Rdzeń 1
16	481926	160640	Nie	Rdzeń 0	Rdzeń 1
17	973444	0	Nie	Rdzeń 0	Rdzeń 1

18	1927686	642560	Tak	Rdzeń 0	Rdzeń 1
19	7536724	0	Nie	Rdzeń 0	Rdzeń 1
20	7710726	2570240	Nie	Rdzeń 0	Rdzeń 1

W konfiguracji jednordzeniowej (1x20) (M_{dur} = 33, TW zadań równe 20) (Tab. 21) różnice między teoretyczną liczbą taktów potrzebną do wykonania zadania a praktyczną dla zadań NCT oraz CTHM jest niewielka i wynosi do 2 cykli zegara. W przypadku zadań CT, znowu jest to konfiguracja szczególna i zawsze oczekiwanie na dostęp do magistrali dla zadań będzie trwało tyle samo. W tym przypadku wszystkie czasy wykonań są równe czasom maksymalnym.

Tab. 21. Porównanie obliczeń i pomiarów - Zadania mieszane, konfiguracja 1 (1x20)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
1	144	163	144	163	153,30493	873794796	0	0
2	1124	1143	1124	1142	1133,358763	257936994	0	1
3	1284	1303	1285	1303	1294,47541	227090920	1	0
4	3324	4143	4124	4143	4133,033334	79774328	800	0
5	4884	4903	4884	4903	4892,695313	68074093	0	0
6	16124	16143	16124	16142	16134,275	21273154	0	1
7	19284	19303	19285	19303	19294,21702	17854254	1	0
8	51324	64143	64124	64143	64132,98949	5413257	12800	0
9	76884	76903	76884	76901	76893,23529	4520545	0	2
10	256124	256143	256124	256142	256132,7392	1359118	0	1
11	307284	307303	307285	307303	307294,5923	1133405	1	0
12	803324	1004143	1004124	1004143	1004133,667	346844	200800	0
13	1228884	1228903	1228884	1228903	1228893,273	283527	0	0
14	4016124	4016143	4016124	4016142	4016133,693	86769	0	1
15	4867284	4867303	4867285	4867303	4867294,553	71596	1	0
16	12851324	16064143	16064124	16064143	16064133,2	21693	3212800	0
17	19468884	19468903	19468884	19468903	19468894,09	17899	0	0
18	64256124	64256143	64256124	64256142	64256133,46	5423	0	1
19	150734484	150734503	150734485	150734503	150734493,7	2311	1	0

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
20	205619324	257024143	257024124	257024143	257024133,6	1355	51404800	0

W Tab. 22 przedstawiono dane dla konfiguracji dwurdzeniowej (2x10) (M_{dur} = 33, TW zadań równe 10). Wszystkie różnice między teoretyczną liczbą taktów potrzebną do wykonania zadania a wartością praktyczną dla zadań NCT oraz CTHM są równe zero. Wyjątkiem jest tu zadanie 15, dla którego różnica ta wynosi 1 takt zegara. W przypadku zadań CT różnice te są większe, lecz dalej mieszczą się w granicach wyznaczonych teoretycznie.

Tab. 22. Porównanie obliczeń i pomiarów - Zadania mieszane, konfiguracja 2 (2x10)

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
1	74	83	74	83	78,29451218	1372318592	0	0
2	764	773	764	773	768,6288492	516293032	0	0
3	644	653	644	653	648,3237409	581561841	0	0
4	1664	2873	2454	2471	2462,824268	199990330	790	402
5	2444	2453	2444	2453	2448,383333	200827111	0	0
6	11264	11273	11264	11273	11268,70588	47417512	0	0
7	9644	9653	9644	9653	9648,384615	54390676	0	0
8	25664	44873	38454	38471	38462,47407	14120656	12790	6402
9	38444	38453	38444	38453	38448,3037	14120656	0	0
10	179264	179273	179264	179273	179268,7037	3053114	0	0
11	153644	153653	153644	153653	153648,5478	3564061	0	0
12	401664	702873	602454	602471	602462,6	909543	200790	100402
13	614444	614453	614444	614453	614448,165	891951	0	0
14	2811264	2811273	2811264	2811273	2811268,653	195023	0	0
15	2433644	2433653	2433645	2433652	2433647,571	225286	1	1
16	6425664	11244873	9638454	9638471	9638462,722	56887	3212790	1606402
17	9734444	9734453	9734444	9734453	9734448,381	56326	0	0

ThID	Obliczenia		Pomiar				Porównanie	
	Min	Max	Min	Max	Średnie wykonanie	Liczba wykonań	Różnica min	Różnica max
18	44979264	44979273	44979264	44979273	44979268,7	12190	0	0
19	75367244	75367253	75367244	75367253	75367248,18	7275	0	0
20	102809664	179916873	154214454	154214471	154214462,6	3555	51404790	25702402

7.3 Przykładowa realizacja systemu

W podrozdziale tym zostanie przeprowadzony proces harmonogramowania wraz z implementacją, syntezą oraz uruchomieniem 60-cio zadaniowego systemu. Następnie zostanie zaprezentowana i zweryfikowana poprawność całego procesu realizacji systemu. Badania zaprezentowane w tym rozdziale dotyczą najnowszej wersji systemu.

7.3.1 Proces harmonogramowania zadań

Parametry harmonogramowanych zadań umieszczono w Tab. 4 na stronie 89. Pierwszy etap harmonogramowania (Podział zadań) został omówiony na przykładzie powyżej wymienionego zestawu zadań w podrozdziale 6.3 na stronie 90. Wynikiem tego etapu są trzy konfiguracje zależne od wybranej metodologii, które przedstawiono w Tab. 23.

Tab. 23. Realizacja systemu - Konfiguracje systemu

Scenariusz	Metodologia	Liczba rdzeni	Rdzenie	Zadania	$Fcore_j$ [MHz]
A	MINRES	1	0	60	90
B	MAXPRO	3	0	22	30
			1	18	29
			2	22	30
C	SFERA	2	0	29	45
			1	31	45

Następnie dla każdej z tych konfiguracji zostanie wykonany i omówiony proces szeregowania zadań oraz symulacji działania systemu.

7.3.1.1 Szeregowanie i symulacja - scenariusz MINRES

Pierwszym krokiem harmonogramowania było uruchomienie procesu szeregowania zadań dla jedynego rdzenia 0, scenariusza A. Etap ten uruchomiono dla standardowych wartości parametrów szeregowania ($Fcore_j$ = 90[MHz] oraz FC_{marg} = 0[MHz]). Jako długość cyklu CP (WL) wybrano wartość 2880, co jest wielokrotnością największego parametru TW wątku SHT (TW_5 = 72).

Po wygenerowaniu CP uruchomiono na jego podstawie symulator wykonywania zadań, który wyznaczył minimalną częstotliwość systemu przy której wszystkie zadania wykonają się na czas, na 108 [MHz]. Wartość ta nie była zadowalająca, więc za pomocą symulatora znaleziono zadanie, które ogranicza tą częstotliwość. Było to zadanie o ThID 46, które w procesie szeregowania miało wartość TW_{46} równą 23. Wartość tą zmniejszono do 20 i ponownie uruchomiono proces szeregowania i symulacji. Po tej zmianie parametrów, minimalna częstotliwość pracy systemu spadła do 100 [MHz] i ograniczało ją klika zadań, dlatego uznano tą wartość wystarczającą.

7.3.1.2 Szeregowanie i symulacja - scenariusz MAXPRO

Scenariusz ten jest trzy rdzeniowy, więc operacje szeregowania i symulacji trzeba przeprowadzić trzykrotnie. Dane uzyskane po tym procesie oraz parametry wejściowe szeregowania umieszczono w Tab. 24.

Tab. 24. Parametry szeregowania oraz min częstotliwość pracy systemu - scenariusz MAXPRO

Rdzeń	**$Fcore_j$ [MHz]**	**Fc_{marg} [MHz]**	**WL**	**Minimalna częstotliwość pracy rdzenia [MHz]**
0	30	0	2700	37
1	29	0 (5)	2700	40 (38)
2	30	0	3200	36

W rdzeniu 0 i 2, minimalną częstotliwość pracy systemu ograniczała znaczna liczba zadań, dlatego szeregowanie tych zadań uznano za wystarczające. W przypadku rdzenia 1, zmiana parametru FC_{margin} na wartość 5 pozwoliła zmniejszyć minimalną częstotliwość pracy rdzenia oraz systemu do 38 [MHz]. W Tab. 24 wartości w nawiasach oznaczają wartości uzyskane po optymalizacji systemu. Minimalna częstotliwość pracy systemu wynosi 38 [MHz].

7.3.1.3 Szeregowanie i symulacja – scenariusz SFERA

W scenariuszu dwurdzeniowym, proces szeregowania uruchomiono z parametrami wejściowymi przedstawionymi w Tab. 25. W rdzeniu 0 zmniejszono minimalną częstotliwość pracy systemu do 51 [MHz] poprzez redukcję parametrów zadań: TW_{46} z 12 na 11 oraz TW_{51} z 51 na 10. W przypadku rdzenia 1 zmniejszono parametr TW_{49} z 16 na 12. Po procesie szeregowania i symulacji, minimalna częstotliwość pracy systemu wynosi 51 [MHz].

Tab. 25. Parametry szeregowania oraz min częstotliwość pracy systemu - scenariusz SFERA

Rdzeń	$Fcore_j$ [MHz]	Fc_{marg} [MHz]	WL	Minimalna częstotliwość pracy rdzenia [MHz]
0	45	0	2400	65 (51)
1	45	0	3456	55 (51)

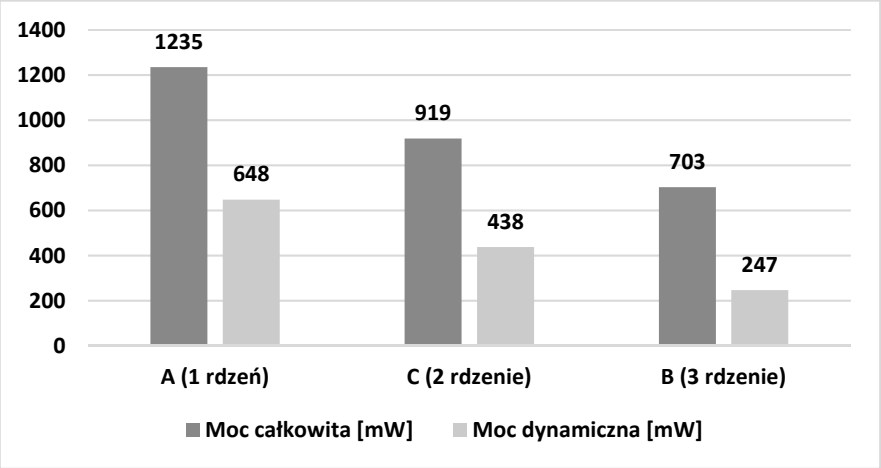
7.3.2 Badania zapotrzebowania na moc i zasoby systemu

Po procesie harmonogramowania zaimplementowano wygenerowane w tym procesie konfiguracje do układu FPGA. Metoda uzyskania prezentowanych niżej danych jest taka sama jak w podrozdziałach 7.1.2 (strona 143) oraz 7.1.3 (strona 148).

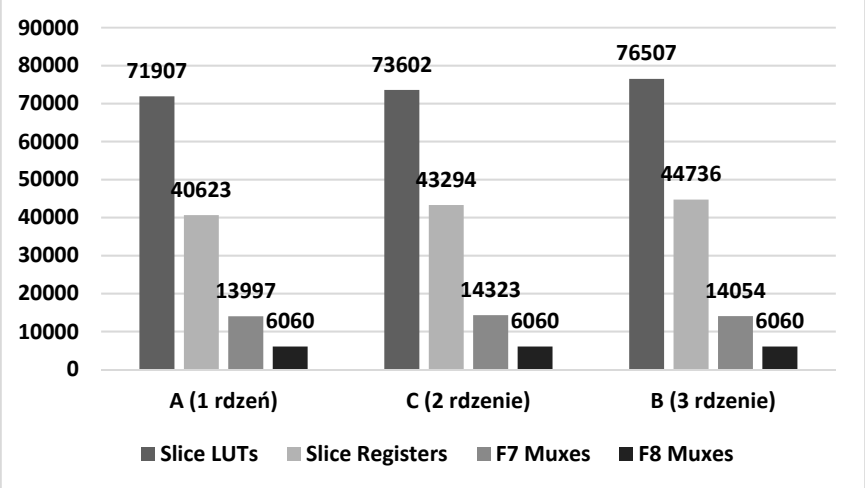
Porównanie zapotrzebowania na moc w zaproponowanych konfiguracjach systemów przedstawiono na Rys. 101. W przypadku mocy całkowitej różnica między skrajnymi konfiguracjami (A i B) wynosi ok. 532 [mW], co stanowi ok. 42 [%] zapotrzebowania na moc konfiguracji jednordzeniowej. W przypadku mocy dynamicznej

wartość ta wynosi ok. 401 [mW], co po przeliczeniu stanowi ok. 61 [%] zapotrzebowania konfiguracji A.

Na Rys. 102 przedstawiono porównanie zasobów potrzebnych do implementacji każdego ze scenariuszy. W każdej z konfiguracji liczba zasobów rośnie wraz z liczbą rdzeni. Wyjątkiem są tu zasoby F7MUX w przypadku scenariusza B. Narzędzie syntezy znalazło i wykorzystało możliwości optymalizacji, wartość ta jest więc mniejsza niż w scenariuszu C.



Rys. 101. Porównanie zapotrzebowania na moc - Realizacja systemu

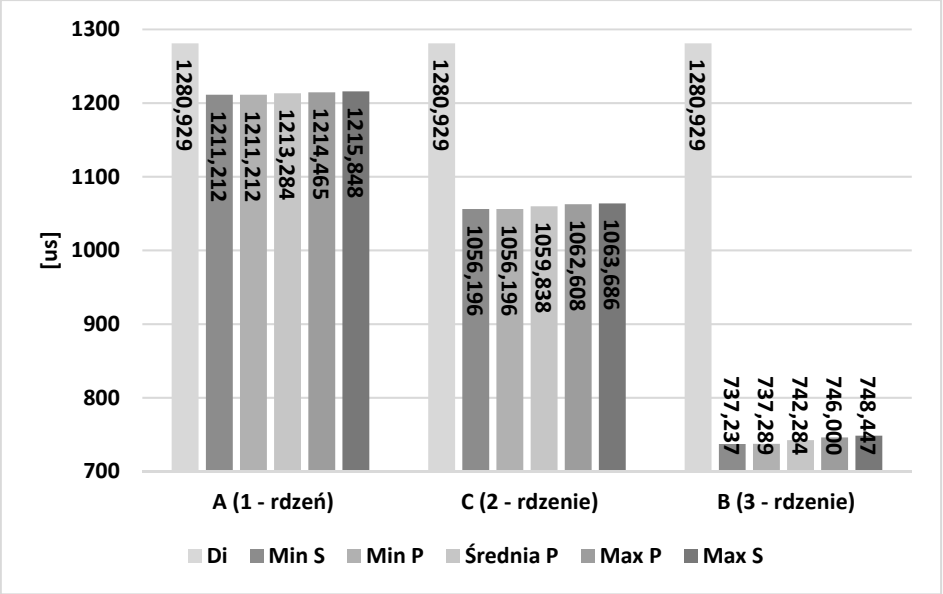


Rys. 102. Porównanie zapotrzebowania na zasoby - Realizacja systemu

7.3.3 Weryfikacja przewidywalności czasowej metodologii harmonogramowania

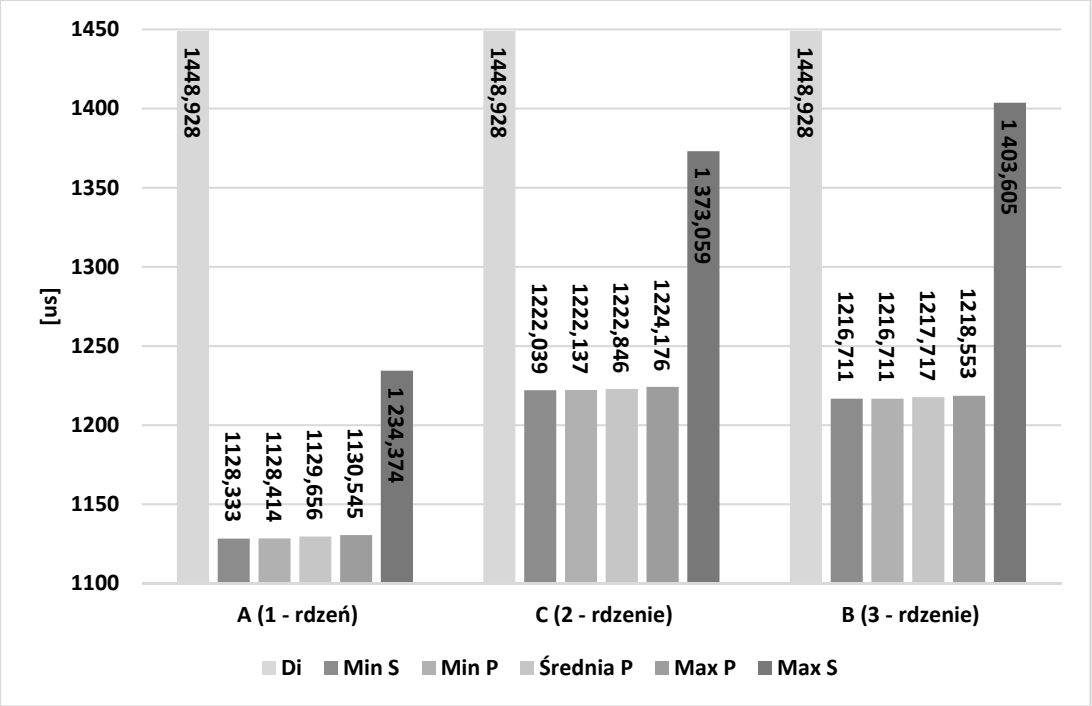
Proces weryfikacji przewidywalności czasowej metodologii harmonogramowania był wykonywany przy pomocy środowiska pomiarowego opisanego w rozdziale 7.2.1 (Opis środowiska pomiarowego, na stronie 159). Testy polegały na uruchomieniu trzech konfiguracji systemu. Następnie każdy z nich pracował i wykonywał zadania przydzielone do zasobów obliczeniowych (rdzeni) przez około godzinę. W tabelach Tab. 26, Tab. 27 oraz Tab. 28 (podrozdział B strona 206) przedstawiono porównanie danych pozyskanych ze środowiska pomiarowego (maksymalny i minimalny czas wykonania każdego zadania) z danymi pozyskanymi za pomocą symulatora oraz parametrem D_i (maksymalnym dopuszczalnym czasem wykonania zadania) zadań. Wszystkie dane przeliczono z taktów zegara na jednostkę czasu [ns], co umożliwiło porównanie z parametrem D_i. Eksperymenty te potwierdzają poprawność koncepcji oraz wykonania metodologii harmonogramowania. Każdy z maksymalnych czasów wykonania zadań jest niższy od parametru D_i. Dodatkowo zweryfikowano działanie symulatora wykonywania zadań. Każdy z maksymalnych i minimalnych czasów wykonania zadań mieści się w przydziale obliczonym przez symulator.

W pierwszym etapie przedstawienia wyników porównano wartości bezwzględne czasów wykonań zadań różnego typu. Z każdego typu zadań wybrano jedno zadanie i porównano czas D_i z maksymalnym i minimalnym czasem wykonania zadania obliczonym przez symulator (Maks S i Min), maksymalnym, minimalnym i średnim czasem wykonania zadania w środowisku pomiarowym (Maks P, Min P oraz Średnia P). Na Rys. 103 przedstawiono pomiary dla zadania typu NCT. Wszystkie pomiary są mniejsze niż czas D_i. Można też zaobserwować, że w każdym scenariuszu różnica ta jest inna. W przypadku tego zadania NCT, różnice między wskazaniami symulatora, a pomiarami są bardzo niewielkie i są na poziomie pojedynczych [us].



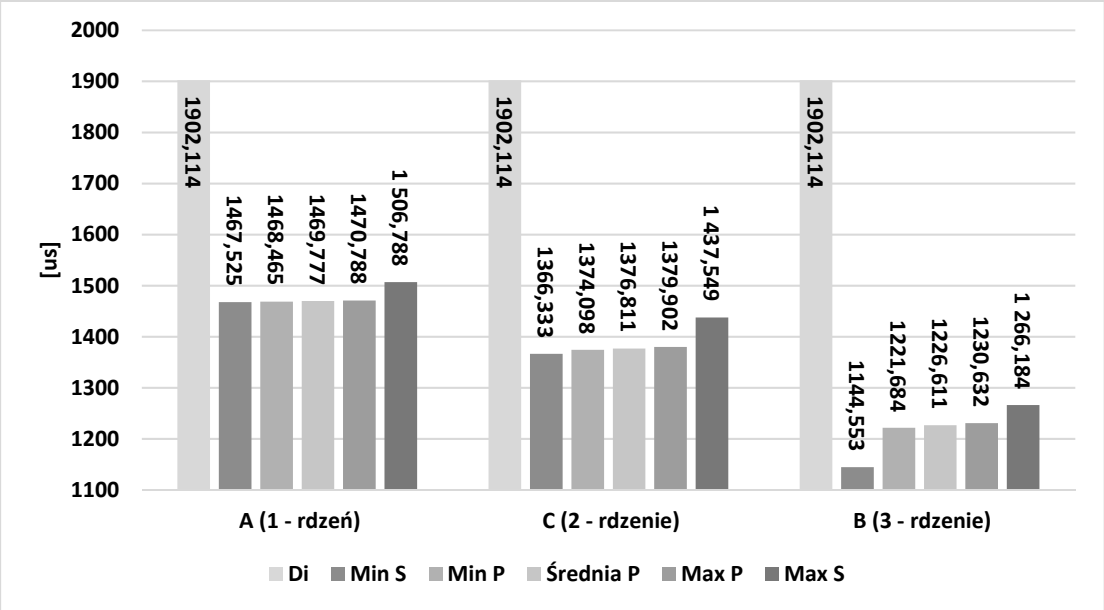
Rys. 103. Pomiary - Zadanie typu NCT (ThID 3)

W przypadku zadania NCT (Rys. 104) można zaobserwować znaczną różnice pomiędzy maksymalnym czasem wykonania zadania obliczonym przez symulator a pomiarem. Różnica ta wynika z dwóch czynników. Pierwszym z nich jest to, że metodologia symulatora zawsze zakłada najgorszy możliwy przypadek, aby zapewnić przewidywalność czasową układu. Druga składowa pochodzi od generatora pseudolosowego używanego w eksperymencie i czasu trwania eksperymentu – nie wszystkie możliwe przypadki mogły zostać wylosowane. Na tym etapie badań można też już stwierdzić, że różnica między czasem wykonania zadania a czasem D_i nie rośnie wraz ze zwiększaniem się liczby rdzeni systemu – jest to czynnik losowy wynikający z procesu harmonogramowania.



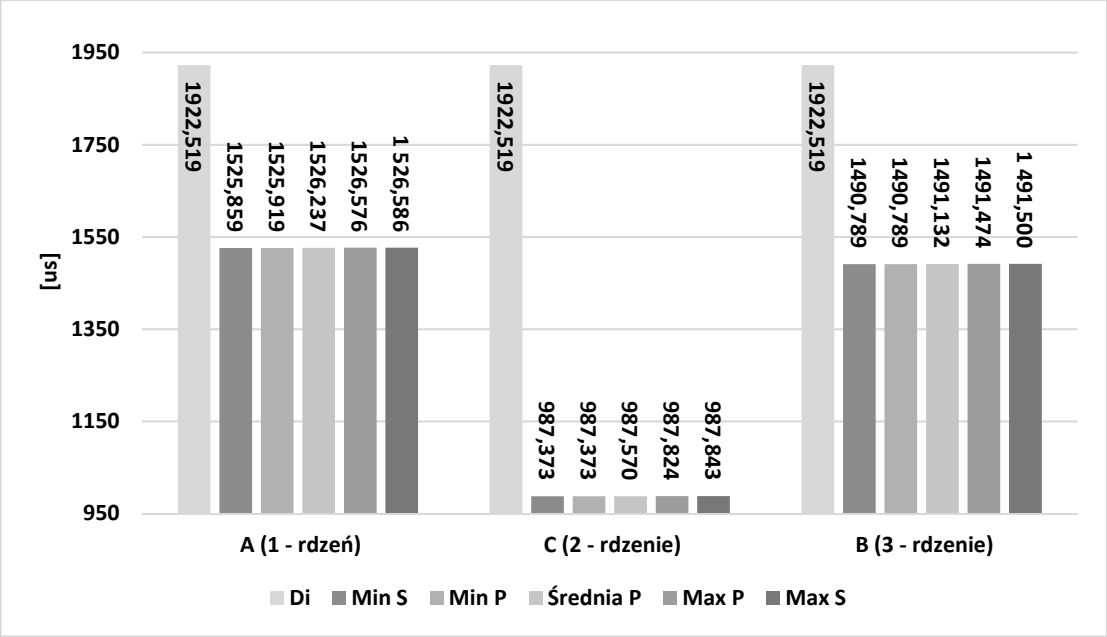
Rys. 104. Pomiary - Zadanie typu CT (ThID 2)

W kolejnym eksperymencie porównano wyniki zadań CTHM. Typ tego zadania został wprowadzony, aby ograniczyć różnice między maksymalnym a minimalnym czasem wykonania zadania typu CT. Jak można zaobserwować na Rys. 105 różnice te zostały zmniejszone.



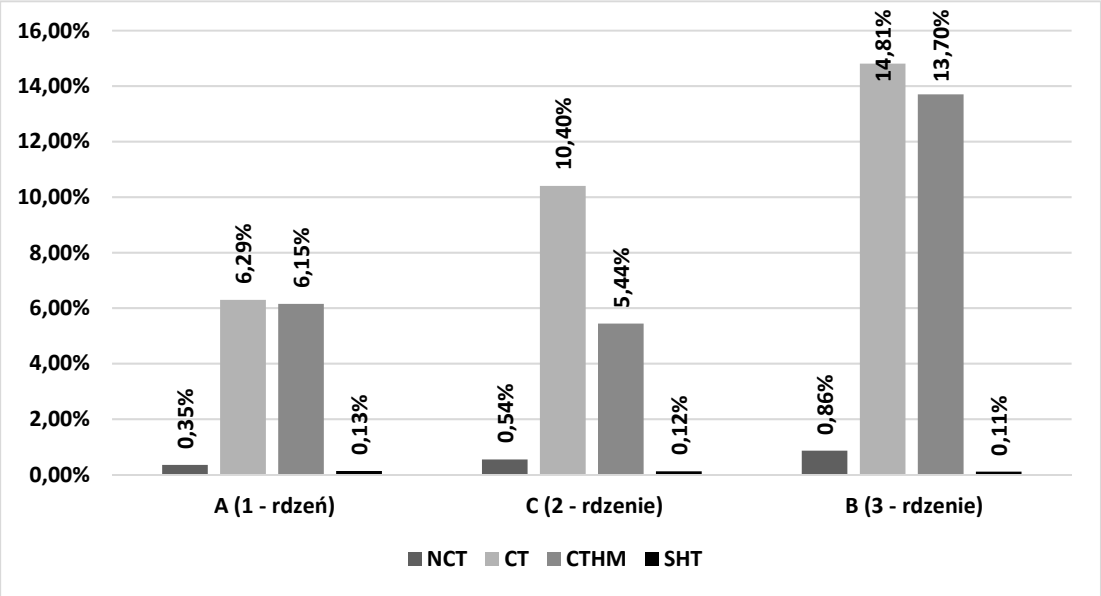
Rys. 105. Pomiary - Zadanie typu CTHM (ThID 1)

Ostatnie porównanie bezwzględne będzie dotyczyć zadania typu SHT. Przedstawiono je na Rys. 106. W przypadku tego zadania, różnice między obliczeniami symulatora a pomiarami są najmniejsze.



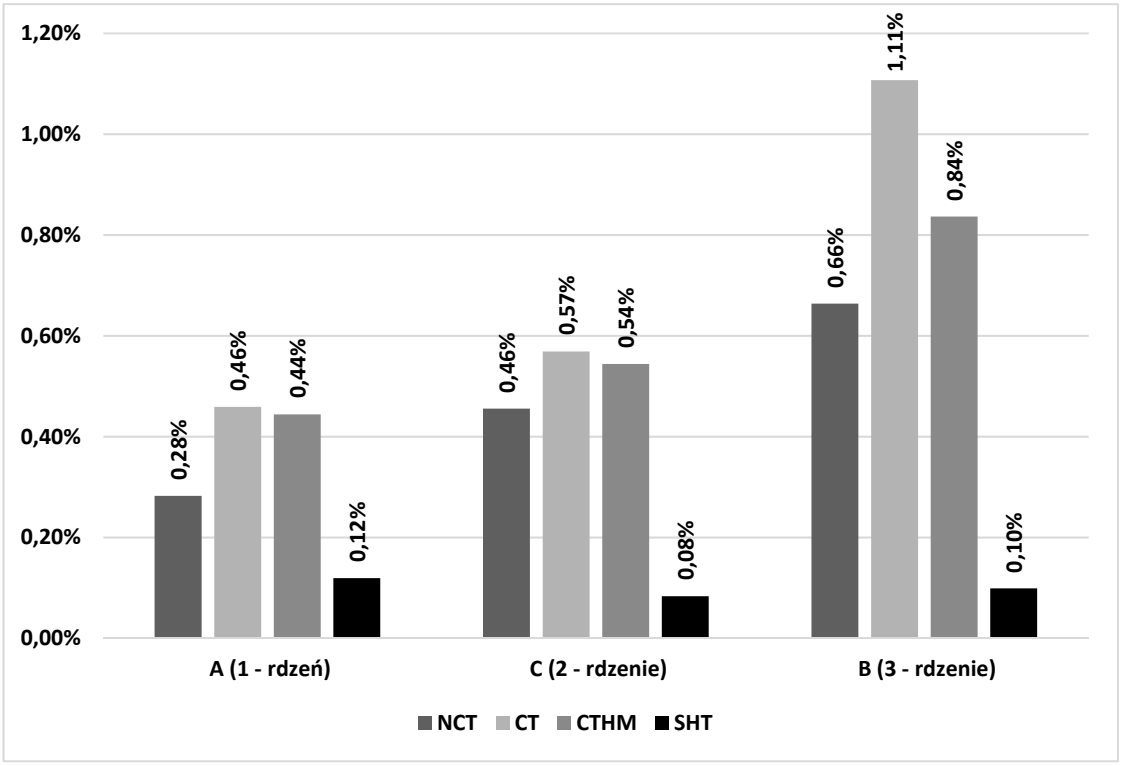
Rys. 106. Pomiary - Zadanie typu SHT (ThID 5)

W celu zwiększenia czytelności analizy większej liczby danych zadania podzielono na grupy według ich typu zadań, a następnie analizowane wartości uśredniono z wartości zadań każdej grupy. Pierwsza analiza będzie dotyczyć różnicy między maksymalnym a minimalnym wykonaniem zadania względem maksymalnego czasu wykonania zadania. Na Rys. 107 przedstawiono takie porównanie względem realizowanych scenariuszy i danych wyliczonych przez symulator. Symulator zawsze zakłada najgorszy możliwy scenariusz, a największe nieprzewidywalności występują podczas wykonywania zadań współpracujących (CT i CTHM). Zadania typu CTHM powinny się cechować mniejszą rozbieżnością niż zadania typu CT. Rozbieżności te w badanym przypadku nie przekraczają 15 [%]. Zadania typu SHT cechują się najmniejszą rozbieżnością (maksymalnie 0,13 [%]) co potwierdza skuteczność metodologii harmonogramowania.



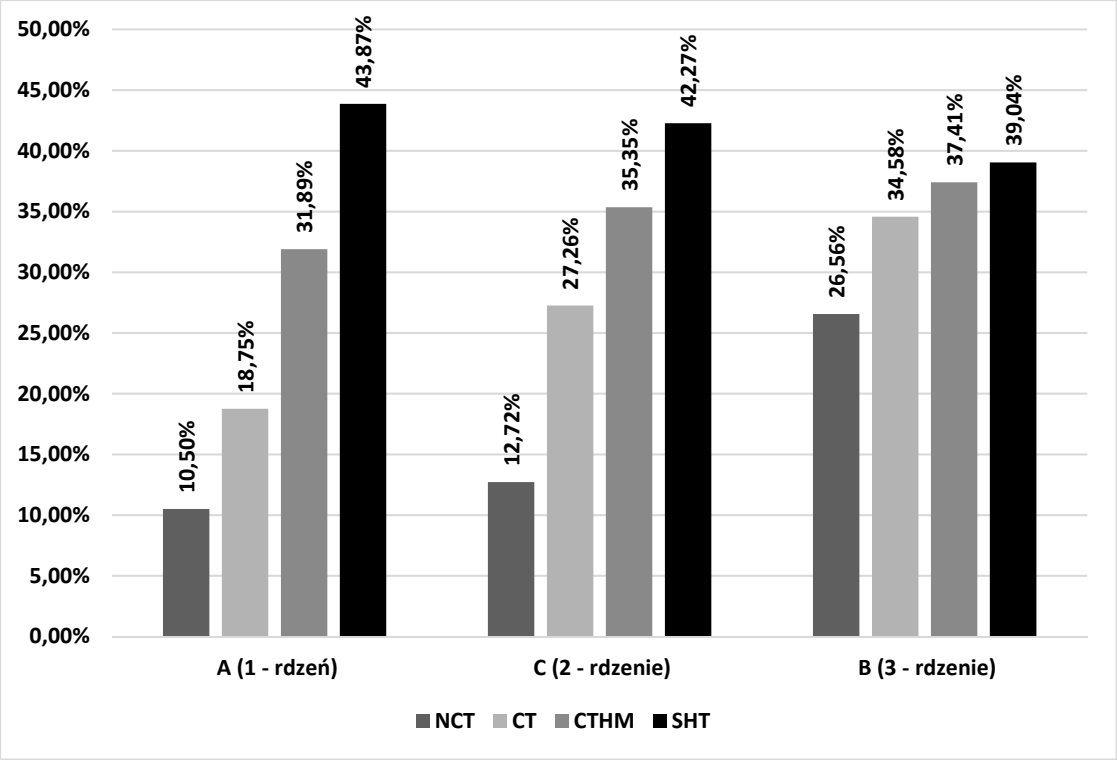
Rys. 107. Rozbieżność między maksymalnym a minimalnym czasem wykonania zadania - symulator

Na Rys. 108 przedstawiono tą samą rozbieżność między maksymalnym a minimalnym czasem wykonania zadania, tym razem jednak są to dane uzyskane za pomocą środowiska pomiarowego. Wartości te są mniejsze i maksymalna rozbieżność wynosi 1,11 [%]. Największe różnice można zaobserwować w grupach zadań CT i CTHM. Wynikają one z największej nieprzewidywalności czasowej tych zadań i metodologii zakładania przez symulator zawsze najgorszego możliwego scenariuszu. Warto tutaj zauważyć, że różnice pomiędzy danymi z symulatora oraz wartościami praktycznymi w przypadku zadań NCT oraz SHT są bardzo niskie. W przypadku zadań NCT maksymalna różnica wynosi 0.15 [%], a w przypadku SHT 0.04 [%].



Rys. 108. Rozbieżność między maksymalnym a minimalnym czasem wykonania zadania - pomiary

Następnie przeprowadzono analizę średnich czasów wykonania zadania uzyskanych za pomocą środowiska pomiarowego oraz czasów D_i. Na Rys. 109 przedstawiono różnice między średnim czasem wykonania zadania a czasem D_i zadania (wyrażony procentowo w czasie D_i zadania). Największe różnice dochodzące do 44 [%] należą do grupy zadań SHT. Oznacza to, że średnio wykonują się one znacznie szybciej niż ustalony na początku harmonogramowania czas D_i. Przyczyną tego jest to, że zadania te są priorytetowe w procesie harmonogramowania. Skutkuje to tym, że często trzeba zwiększyć minimalną częstotliwość systemu ze względu na pozostałe zadania, co również przyśpiesza prace zadań SHT.



Rys. 109. Średni czas wykonania zadania przed czasem D_i względem czasu D_i

8 Podsumowanie

Rozdział ten zostanie podzielony na trzy części. W pierwszej z nich zostanie przeprowadzone porównanie metody autorskiej z innymi metodami dostępnymi w literaturze. W kolejnej części nastąpi zaprezentowanie wniosków powstałych po ukończeniu niniejszej pracy. W ostatniej części autor rozprawy przytoczy, ważne jego zdaniem, informacje dla osób projektujących systemy PRET.

8.1 Porównanie z innymi rozwiązaniami

Najbardziej miarodajną metodą porównania proponowanej metodologii oraz architektury z innymi rozwiązaniami, było by uruchomienie tych samych zestawów programów na wszystkich porównywanych systemach. Systemy powinny być również wykonane w tych samych technologiach. Następnie, po przeprowadzeniu szczegółowej analizy ich wykonania, można by było przeprowadzić takie porównanie. Niestety, takie bezpośrednie porównanie jest bardzo trudne do przeprowadzenia. Pierwsza trudność wynika z różnych architektur sprzętowych wykorzystywanych podczas omawiania takich rozwiązań. Problemem jest również technologia ich wykonania. Niektóre z rozwiązań uruchamiane są na układach ASIC, które są kilkukrotnie szybsze od układów FPGA [81], co dawałoby przewagę niektórym z nich. Autor musiałby więc przygotować modele wszystkich porównywanych systemów (gdzie ich autorzy nie zawsze publikują wszystkie niezbędne dane) oraz napisać odpowiedniki programów testowych działających na wszystkich porównywanych platformach. Taka metodologia porównywania wymaga zbyt dużych nakładów pracy. Dlatego autor zdecydował się na opisową formę porównania.

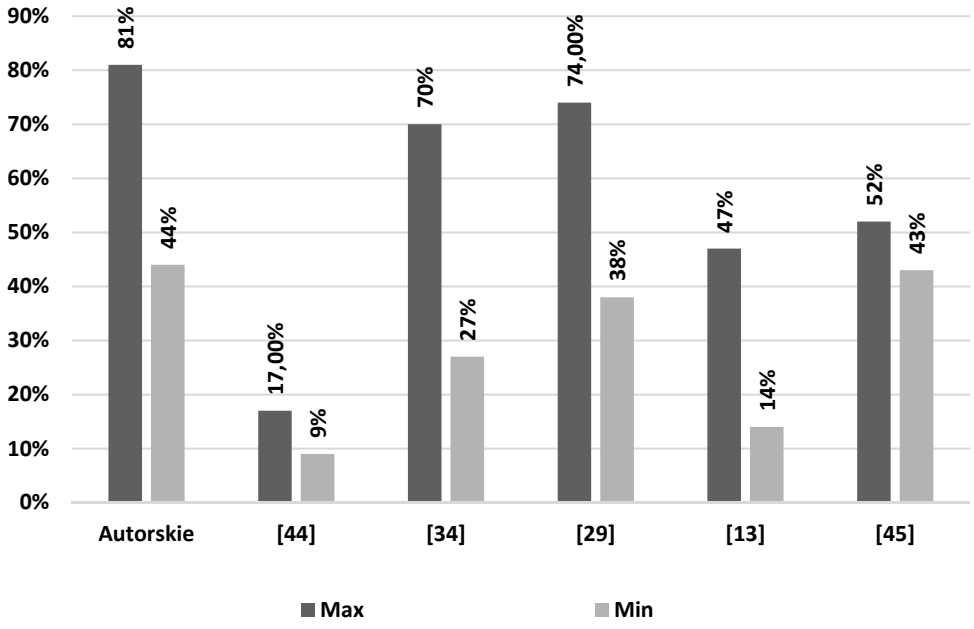
Zaproponowany przez autora pracy model wymiany danych między zadaniami bazuje na technice TDMA (ang. *Time Division Multiple Access*) z autorskim pomysłem wykorzystania metody przeplotu wątków do wymiany danych. W porównaniu do metody koła pamięci [15] rozwiązanie autorskie jest bardziej wydajne. Rozróżnia ono zadania CT i NCT, a protokół wymiany danych nie zakłóca pracy zadań niezaangażowanych w wymianę danych. Tylko zadanie żądające dostępu do pamięci jest zablokowane, a praca pozostałych zadań nie jest w żadnym stopniu sztucznie spowolniania.

Proponowane dotychczas architektury PRET w wielu przypadkach są analizowane i modelowane na wysokim poziomie abstrakcji za pomocą narzędzi SystemC (C++) [15], [42], [82]–[85] lub Java [16], [86]. Często układy te są również implementowane jako układy SoC (ang. *System on Chip*) [17], [60], [82], [87]–[90]. Wykorzystuje się też procesory nieprzystosowane do zadań czasu rzeczywistego, implementując to wsparcie programowo [33], [91]. Podejście zaprezentowane w tej pracy prezentuje sprzętowy model niskiego poziomu opisany za pomocą języka Verilog. Zaprojektowanie modelu niskiego poziomu wymaga dużo więcej pracy, symulacji oraz testów [4], ale otrzymane rozwiązanie jest bliższe sprzętowi. Pozwala ono lepiej wykorzystać dostępne zasoby układu FPGA oraz mieć większą kontrolę nad przewidywalnością działania systemu.

Autorskie metodologie podziału zadań, wraz ze statycznym harmonogramowaniem, pozwalają zapewnić krytyczną przewidywalność czasową oraz dopasować strukturę do aktualnych wymagań projektowych (zużycia zasobów oraz mocy). Harmonogramowanie statyczne wymaga jednak przygotowania systemu oraz znajomości wszystkich wykonywanych zadań jeszcze przed uruchomieniem systemu. W [33] zaproponowano algorytmy oparte na różnych filozofiach (SFLS, SSLF), które umożliwiają kradzież zadań (ang. *task stealing*) i minimalizacje takich parametrów systemu jak zapotrzebowanie na moc. Takie podejście nie może być jednak stosowane w krytycznych systemach czasu rzeczywistego, ponieważ harmonogramowanie dynamiczne nie zapewnia przewidywalności czasowej w każdym możliwym przypadku. Podobny model zadania do proponowanego przez autora pracy analizowany jest w [92], ale w autorskim modelu opóźnienie instrukcji (ang. *instruction latency*) oraz opóźnienie danych (ang. *data latency*) są brane pod uwagę przy założeniu najgorszego możliwego przypadku (WCET) [93]–[95]. Ponadto w [92] harmonogramowanie zadań bazuje na ocenie maksymalnego czasu wykonania zadania, który jest ściśle powiązany z częstotliwością pracy systemu na którym wykonywane jest zadanie. W autorskim modelu poprzez zastosowanie parametru TF_i, parametry zadań są uniezależnione od prędkości przetwarzania systemu.

Metodologia harmonogramowania zaproponowana przez autora pracy bazuje na metodach heurystycznych. W literaturze można znaleźć również metody heurystyczno-ewolucyjne [67] oraz czysto matematyczne (algorytmiczne) [33], [96], [97]. Złożoność obliczeniowa algorytmu harmonogramowania zaprezentowanego w tej pracy jest równa $O(n \cdot \ln(n)) + O(n)$ i jest równa rozwiązaniu [96]. Zdecydowanie niższą złożoność posiada rozwiązanie stosujące metodę ewolucyjną, ale rozwiązanie to nie jest zawsze optymalne.

Na Rys. 110 przedstawiono porównanie minimalnych i maksymalnych uzyskanych zysków energii każdego ze wspomnianych rozwiązań. Należy jednak pamiętać, że są to dane odczytane bezpośrednio z każdej publikacji, każdy pomiar przeprowadzany był w innych warunkach i na innej platformie sprzętowej (układ FPGA, chmura obliczeniowa [96], symulowany w języku Java procesor [73], procesor PC [33] oraz mikroprocesor przeznaczony do telefonu komórkowego [97]) przez co porównanie to nie jest do końca sprawiedliwe. Wynik uzyskany przez autora pracy jest najlepszy, spowodowane jest to projektowaniem systemu na najniższym poziomie abstrakcji.



Rys. 110. Porównanie uzyskanych oszczędności energii

8.2 Wnioski

Zdaniem autora, najważniejszymi efektami niniejszej pracy są:

- Opracowanie i zaimplementowanie parametrycznej architektury systemu czasu rzeczywistego. Dzięki temu przed realizacją systemu, za pomocą parametrów języku opisu sprzętu, można dopasowywać strukturę systemu do przetwarzanych zadań. Najważniejszymi parametrami są: liczba rdzeni systemu oraz liczba obsługiwanych zadań.

- Opracowanie metody oraz protokołu wymiany danych między zadaniami przetwarzanymi w potoku z przeplotem wątków. Protokół ten został tak zaprojektowany, aby minimalizował wszystkie nieprzewidywalności czasowe związane z wymianą danych.
- Opracowanie zestawu algorytmów statycznego harmonogramowania zadań. Metodologia ta, na podstawie zaproponowanego modelu zadania oraz wybranej strategii (minimalizacja zasobów lub minimalizacja energii), pozwala wyznaczyć parametry niezbędne do skonfigurowania modelu systemu tak, aby architektura systemu byłą dopasowana do zadań.
- Opracowanie i praktyczna implementacja symulatora wykonywania zadań, który na podstawie parametrów uzyskanych w procesie harmonogramowania (jeszcze przed właściwą pracą systemu) pozwala wyznaczyć maksymalny i minimalny czas wykonania każdego zadania pracującego w systemie.
- Opracowanie i zaimplementowanie programów ułatwiających korzystanie z systemu, takich jak kompilator zadań czy programy ułatwiające harmonogramowanie zadań.
- Przeprowadzenie szeregu eksperymentów praktycznych, które potwierdziły rozważania teoretyczne oraz przyjętą koncepcję systemu. Eksperymenty dotyczyły zużycia mocy oraz zasobów w różnych konfiguracjach systemu oraz weryfikacji przewidywalności czasowej systemu.

Opis zaprojektowanego modelu systemu, zaprezentowany w rozdziale 5 (Budowa oraz implementacja systemu - strona 37) wraz z opisem zaproponowanej metodologii harmonogramowania w rozdziale 6 (Metodologia harmonogramowania zadań – strona 87), został zweryfikowany za pomocą eksperymentów praktycznych w rozdziale 7 (Weryfikacja działania systemu – strona 141). Powyższe rozważania i badania z całą pewnością pozwalają udowodnić tezę główną tj.:

Metodologia dopasowywania struktury oraz zasobów systemu PRET do wymagań czasowych wykonywanych zadań, pozwala na efektywną implementację sprzętową krytycznie przewidywalnego systemu czasu rzeczywistego

Przedstawiona w tej pracy metodologia pozwala na dopasowanie struktury sprzętowej w taki sposób, aby minimalizować energię potrzebną do funkcjonowania

systemu lub ilość potrzebnych zasobów, zapewniając przy tym krytyczną przewidywalność czasową systemu. Zaproponowana metoda symulacji wykonywania zadań pozwala jeszcze przed uruchomianiem systemu przewidzieć maksymalne i minimalne czasy wykonywania zadań. Podział na różne typy zadań pozwala dopasować różnice między maksymalnym a minimalnym czasem wykonania zadania do wymagań aplikacji zadania. Parametryczny model systemu został opisany za pomocą języka opisu sprzętu Verilog w taki sposób, aby była możliwość konfiguracji systemu bez użycia zewnętrznych programów generujących kod w języku Verilog. Wszystkie parametry systemu wraz ze strukturą sprzętową ustawiana jest w jednym głównym pliku Verilog. Wykorzystanie metody przeplotu wątków pozwoliło uzyskać przewidywalność czasową systemu.

Przedstawiony w rozdziale 5.2 (Konfigurowalność systemu – strona 48) opis implementacji konfigurowalności systemu wraz z wynikami eksperymentów (7.2.2 Przewidywalność czasowa mikroarchitektury systemu- strona 160) oraz metodologią podziału zadań (6.3 Podział zadań – strona 90) pozwalają udowodnić pierwszą tezę pomocniczą:

Język opisu sprzętu Verilog pozwala na opisanie parametrycznego modelu systemu PRET, w którym konfigurowalna jest struktura systemu wynikająca z parametrów zadań

Przedstawione eksperymenty w rozdziale 7.3.3 (Weryfikacja przewidywalności czasowej metodologii harmonogramowania – strona 179) wraz z opisem metodologii harmonogramowania (6 Metodologia harmonogramowania zadań – strona 87) pozwalają udowodnić drugą tezę pomocniczą:

Dopasowana struktura sprzętowa systemu PRET pozwala wyznaczyć minimalny i maksymalny czas wykonania zadania który jest niższy od zakładanego czasu deadline zadania

Omawiana idea wykorzystania przeplotu wątków do wymiany danych między zadaniami w rozdziale 4.5 (Pamięć systemu oraz komunikacja wątków – strona 30) wraz z opisem implementacji tej metody w rozdziale 5.3 (Magistrala wymiany danych między wątkami – strona 54) zweryfikowana w rozdziale 7.2.2.2 oraz 7.2.2.3 (Zadania współpracujące CTHM oraz Zadania współpracujące CT – strona 165) pozwalają udowodnić trzecią tezę pomocniczą:

Metoda przeplotu wątków pozwala wyeliminować nieprzewidywalności czasowe wynikające z dostępu do danych innych zadań

Przedstawiona metodologia podziału zadań (6.3 Podział zadań – strona 90) wraz z eksperymentami potwierdzającymi jej poprawność w podrozdziałach 7.1.3, 7.1.3 (Badanie zapotrzebowania systemu na zasoby – strona 143, Badania zapotrzebowania systemu na energie – strona 143) oraz podrozdział 7.3.2 (Badania zapotrzebowania na moc i zasoby systemu – strona 177) pozwala udowodnić czwartą tezę pomocniczą:

Metodologia dopasowywania struktury oraz zasobów systemu PRET, pozwala na minimalizacje zużytych zasobów sprzętowych lub minimalizacje energii potrzebnej do funkcjonowania systemu

Przedstawione w pracy rozważania mogą być wykorzystane do implementacji krytycznych czasowo systemów wbudowanych o dużej liczbie wykonywanych zadań, gdzie system ten może być alternatywą dla dużej liczby standardowych procesorów. Omawiany system pozwala na wymianę danych pomiędzy zadaniami w systemie oraz wg badań przeprowadzonych w rozdziale 7.1.4 (Porównanie systemu wielozadaniowego z klasycznym systemem jednozadaniowym – strona 154) pozwala na większą oszczędność zasobów niż rozwiązanie wieloprocesorowe. Bardzo ważnym wnioskiem pochodzącym z tych rozważań jest również Rys. 98 (strona 158). Wykres ten pokazuje zależność między pojemnością używanego układu FPGA (Viretx-7) a minimalną mocą zużywaną przez każde zadanie. Minimalizacja energii potrzebnej do funkcjonowania systemu wiąże się z maksymalizacją zasobów systemów. Założenie to działa również w drugą stronę: Minimalizacja zasobów powoduje maksymalizacje energii potrzebnej do funkcjonowania systemu.

Dalsze badania powinny dotyczyć przewidywalnych czasowo technik programowania oraz prac nad kompilatorem zadań ułatwiającym proces tworzenia takich programów. Niezbędną częścią takiego systemu jest metodologia umożliwiająca szybkie i proste programowanie przewidywalnych czasowo programów. Kolejnym pomysłem jest zaimplementowanie każdego z rdzeni zaproponowanego systemu PRET jako układu GALS (ang. *Globally asynchronous locally synchronous*) [98]. Pozwoli to na to, aby każdy z rdzeni pracował z inną częstotliwością zegara, lecz będzie wymagało opracowania innego protokołu wymiany danych na magistrali. W przyszłości jest

również planowany proces implementacji omawianego prototypu z układu FPGA do układu ASIC oraz wypróbowanie systemu w rzeczywistej krytycznie czasowej aplikacji.

8.3 Zalecenie dla projektantów systemów PRET

Autor pracy po zdobytym doświadczeniu podczas projektowania i implementacji systemu PRET sformułował następujące zalecenia dla osób rozpoczynających projektowanie takie systemu:

1. Najbardziej energooszczędnym systemem wielozadaniowym, niewymagającym komunikacji między zadaniami będzie system, w którym każde zadanie pracuje w pojedynczym procesorze, zoptymalizowanym pod względem zużycia energii (wykorzystującym techniki bramkowania zegara). Rozwiązanie takie zminimalizuje liczbę połączeń oraz zredukuje minimalną częstotliwość zegara potrzebną do prawidłowego funkcjonowania systemu, powodując minimalizacje energii potrzebnej do funkcjonowania systemu.
2. Najbardziej energooszczędnym systemem wielozadaniowym wymagającym komunikacji między zadaniami, będzie system z jak największą możliwą liczbą rdzeni. Duża liczba rdzeni wymusi równoległą pracę systemu, zmniejszając minimalną częstotliwość zegara potrzebną do właściwego funkcjonowania systemu, powodując minimalizację energii potrzebnej do funkcjonowania systemu.
3. Systemem o najmniejszym zapotrzebowaniu na zasoby, będzie system wielordzeniowy o minimalnej, możliwej liczbie rdzeni. Mała liczba rdzeni redukuje liczbę potoków przetwarzania pracujących w systemie. Powoduje to zmniejszenie ilości zasobów wymaganych do właściwego funkcjonowania systemu.

9 Spis literatury

[1] T. A. Henzinger, „Two challenges in embedded systems design: predictability and robustness”, *Philos. Trans. R. Soc. Math. Phys. Eng. Sci.*, t. 366, nr 1881, s. 3727–3736, paź. 2008, doi: 10.1098/rsta.2008.0141.

[2] M. Paolieri *i in.*, „A Software-Pipelined Approach to Multicore Execution of Timing Predictable Multi-threaded Hard Real-Time Tasks”, w *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Newport Beach, CA, USA, mar. 2011, s. 233–240. doi: 10.1109/ISORC.2011.36.

[3] T. Ungerer *i in.*, „Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability”, *IEEE Micro*, t. 30, nr 5, s. 66–75, wrz. 2010, doi: 10.1109/MM.2010.78.

[4] D. D. Gajski, Red., *Embedded system design: modeling, synthesis and verification*. Dordrecht ; New York: Springer, 2009.

[5] E. A. Lee, „Absolutely positively on time: what would it take? [embedded computing systems”, *Computer*, t. 38, nr 7, s. 85–87, lip. 2005, doi: 10.1109/MC.2005.211.

[6] S. Andalam, P. S. Roop, A. Girault, i C. Traulsen, „A Predictable Framework for Safety-Critical Embedded Systems”, *IEEE Trans. Comput.*, t. 63, nr 7, s. 1600–1612, lip. 2014, doi: 10.1109/TC.2013.28.

[7] J. Ulasiewicz, *Systemy czasu rzeczywistego QNX6 Neutrino*. Warszawa: Wydawnictwo BTC, 2007.

[8] D. Broman *i in.*, „Precision timed infrastructure: design challenges”, Austin, Texas, USA, May 31-June 1, s. 1–6.

[9] T. Henzinger i C. Kirsch, „The Embedded Machine: Predictable, Portable Real-Time Code”, *ACM Trans. Program. Lang. Syst.*, t. 29, grudz. 2002, doi: 10.1145/512529.512567.

[10] E. A. Lee, „The Problem with Threads”, *Computer*, t. 39, nr 5, s. 33–42, maj 2006, doi: 10.1109/MC.2006.180.

[11] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, i M. Valero, „Predictable performance in SMT processors: synergy between the OS and SMTs”, *IEEE Trans. Comput.*, t. 55, nr 7, s. 785–799, lip. 2006, doi: 10.1109/TC.2006.108.

[12] S. A. Edwards i E. A. Lee, „The Case for the Precision Timed (PRET) Machine”, w *2007 44th ACM/IEEE Design Automation Conference*, San Diego, CA, USA, cze. 2007, s. 264–265. doi: 10.1109/DAC.2007.375165.

[13] L. Thiele i R. Wilhelm, „Design for Timing Predictability”, *Real-Time Syst.*, t. 28, nr 2, s. 157–177, lis. 2004, doi: 10.1023/B:TIME.0000045316.66276.6e.

[14] N. J. H. Ip i S. A. Edwards, „A Processor Extension for Cycle-Accurate Real-Time Software”, w *Embedded and Ubiquitous Computing*, t. 4096, E. Sha, S.-K. Han, C.-Z. Xu, M.-H. Kim, L. T. Yang, i B. Xiao, Red. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, s. 449–458. doi: 10.1007/11802167_46.

[15] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, i E. A. Lee, „Predictable programming on a precision timed architecture”, w *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems - CASES '08*, Atlanta, GA, USA, 2008, s. 137. doi: 10.1145/1450095.1450117.

[16] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, i C. W. Probst, „Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach”, Grenoble, France, 2011, s. 11–21. Dostęp: 12 listopad 2019. [Online]. Dostępne na: http://drops.dagstuhl.de/opus/volltexte/2011/3077

[17] M. Schoeberl *i in.*, „T-CREST: Time-predictable multi-core architecture for embedded systems”, *J. Syst. Archit.*, t. 61, nr 9, s. 449–471, paź. 2015, doi: 10.1016/j.sysarc.2015.04.002.

[18] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, i F. J. Cazorla, „Assessing the suitability of the NGMP multi-core processor in the space domain”, w *Proceedings of the tenth ACM international conference on Embedded software - EMSOFT '12*, Tampere, Finland, 2012, s. 175–184. doi: 10.1145/2380356.2380389.

[19] A. Alhammad i R. Pellizzoni, „Time-predictable execution of multithreaded applications on multicore systems”, w *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, Dresden, Germany, 2014, s. 1–6. doi: 10.7873/DATE.2014.042.

[20] S. Moulik, R. Devaraj, i A. Sarkar, „COST: A Cluster-Oriented Scheduling Technique for Heterogeneous Multi-cores”, w *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Miyazaki, Japan, paź. 2018, s. 1951–1957. doi: 10.1109/SMC.2018.00337.

[21] R. Pathan, P. Voudouris, i P. Stenstrom, „Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms”, *IEEE Trans. Parallel Distrib. Syst.*, t. 29, nr 4, s. 915–928, kwi. 2018, doi: 10.1109/TPDS.2017.2777449.

[22] D. Kim, Y.-B. Ko, i S.-H. Lim, „Energy-Efficient Real-Time Multi-Core Assignment Scheme for Asymmetric Multi-Core Mobile Devices”, *IEEE Access*, t. 8, s. 117324–117334, 2020, doi: 10.1109/ACCESS.2020.3005235.

[23] J. Chen, C. Du, P. Han, i Y. Zhang, „Sensitivity Analysis of Strictly Periodic Tasks in Multi-Core Real-Time Systems”, *IEEE Access*, t. 7, s. 135005–135022, 2019, doi: 10.1109/ACCESS.2019.2941958.

[24] Y. Liu, G. Xie, Y. Tang, i R. Li, „Improving Real-Time Performance Under Reliability Requirement Assurance in Automotive Electronic Systems”, *IEEE Access*, t. 7, s. 140875–140888, 2019, doi: 10.1109/ACCESS.2019.2944204.

[25] S. L. Tripathi, Red., *Design and development of efficient energy systems*. Hoboken, NJ, USA: Wiley-Scrivener, 2021.

[26] F. Glaser, G. Tagliavini, D. Rossi, G. Haugou, Q. Huang, i L. Benini, „Energy-Efficient Hardware-Accelerated Synchronization for Shared-L1-Memory Multiprocessor Clusters”, *IEEE Trans. Parallel Distrib. Syst.*, t. 32, nr 3, s. 633–648, mar. 2021, doi: 10.1109/TPDS.2020.3028691.

[27] M. Lohstroh, H. Kim, J. C. Eidson, C. Jerad, B. Osyk, i E. A. Lee, „On Enabling Technologies for the Internet of Important Things”, *IEEE Access*, t. 7, s. 27244–27256, 2019, doi: 10.1109/ACCESS.2019.2901509.

[28] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, i M. Schoeberl, „A disruptive computer design idea: Architectures with repeatable timing”, w *2009 IEEE International Conference on Computer Design*, paź. 2009, s. 54–59. doi: 10.1109/ICCD.2009.5413177.

[29] A. Pułka i A. Milik, „Dynamic Rescheduling of Tasks in Time Predictable Embedded Systems”, *IFAC Proc. Vol.*, t. 45, nr 7, s. 305–310, 2012, doi: 10.3182/20120523-3-CZ-3015.00058.

[30] Jinkyu Lee i K. G. Shin, „Preempt a Job or Not in EDF Scheduling of Uniprocessor Systems”, *IEEE Trans. Comput.*, t. 63, nr 5, s. 1197–1206, maj 2014, doi: 10.1109/TC.2012.279.

[31] S. Moulik, R. Devaraj, i A. Sarkar, „HETERO-SCHED: A Low-Overhead Heterogeneous Multi-core Scheduler for Real-Time Periodic Tasks”, Exeter, United Kingdom, cze. 2018, s. 659–666. doi: 10.1109/HPCC/SmartCity/DSS.2018.00117.

[32] L. Bo, Y. Yunhong, P. Lizhang, C. Yunlong, i J. Yan, „A hard real-time scheduling algorithm based on elastic model”, w *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, Shanghai, China, lis. 2014, s. 320–333. doi: 10.1109/ICSAI.2014.7009309.

[33] S. I. Kim i J.-K. Kim, „A Method to Construct Task Scheduling Algorithms for Heterogeneous Multi-Core Systems”, *IEEE Access*, t. 7, s. 142640–142651, 2019, doi: 10.1109/ACCESS.2019.2944238.

[34] E. Lee i D. Messerschmitt, „Pipeline interleaved programmable DSP’s: Architecture”, *IEEE Trans. Acoust. Speech Signal Process.*, t. 35, nr 9, s. 1320–1333, wrz. 1987, doi: 10.1109/TASSP.1987.1165274.

[35] M. M. Mano Kime, Charles R. i A. K. Pulka, *Podstawy projektowania układów logicznych i komputerów*. Warszawa: Wydawnictwa Naukowo-Techniczne, 2007.

[36] I. Liu, J. Reineke, D. Broman, M. Zimmer, i E. A. Lee, „A PRET microarchitecture implementation with repeatable timing and competitive performance”, w *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Montreal, QC, Canada, wrz. 2012, s. 87–93. doi: 10.1109/ICCD.2012.6378622.

[37] I. Liu, J. Reineke, i E. A. Lee, „A PRET architecture supporting concurrent programs with composable timing properties”, w *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, USA, lis. 2010, s. 2111–2115. doi: 10.1109/ACSSC.2010.5757922.

[38] Ł. Golly, „Multitask real-time architecture supporting predictable management and memory system”, *Pomiary Autom. Kontrola*, t. R. 60, nr 6, 2014, Dostęp: 19 sierpień 2022. [Online]. Dostępne na: http://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-e2cca740-2a4d-4ea5-a7c8-da31cccd7238

[39] W. Stallings i J. Szporko, *Organizacja i architektura systemu komputerowego: projektowanie systemu a jego wydajnosc*. Warszawa: Wydawnictwa Naukowo-Techniczne, 2000.

[40] Y.-H. Chen i Y.-Y. Liu, „Dual-addressing Memory Architecture for Two-dimensional Memory Access Patterns”, w *Design, Automation & Test in Europe*

Conference & Exhibition (DATE), 2013, Grenoble, France, 2013, s. 71–76. doi: 10.7873/DATE.2013.029.

[41] Ł. Golly, A. Milik, i A. Pulka, „High Level Model of Time Predictable Multitask Control Unit", *IFAC-Pap.*, t. 48, nr 4, s. 348–353, 2015, doi: 10.1016/j.ifacol.2015.07.059.

[42] J. Reineke, I. Liu, H. D. Patel, S. Kim, i E. A. Lee, „PRET DRAM controller: bank privatization for predictability and temporal isolation", w *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '11*, Taipei, Taiwan, 2011, s. 99. doi: 10.1145/2039370.2039388.

[43] Jun Yan i Wei Zhang, „Time-predictable multicore cache architectures", w *2011 3rd International Conference on Computer Research and Development*, Shanghai, China, mar. 2011, s. 1–5. doi: 10.1109/ICCRD.2011.5764232.

[44] A. Pyka, M. Rohde, i S. Uhrig, „Extended performance analysis of the time predictable on-demand coherent data cache for multi- and many-core systems", w *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, Agios Konstantinos, Samos, Greece, lip. 2014, s. 107–114. doi: 10.1109/SAMOS.2014.6893201.

[45] L. M. AlBarakat, P. V. Gratz, i D. A. Jimenez, „MTB-Fetch: Multithreading Aware Hardware Prefetching for Chip Multiprocessors", *IEEE Comput. Archit. Lett.*, t. 17, nr 2, s. 175–178, lip. 2018, doi: 10.1109/LCA.2018.2847345.

[46] A. F. Lorenzon, M. C. Cera, i A. C. S. Beck, „Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy", *J. Parallel Distrib. Comput.*, t. 95, s. 107–123, wrz. 2016, doi: 10.1016/j.jpdc.2016.04.003.

[47] Y. Liu i W. Zhang, „Exploiting multi-level scratchpad memories for time-predictable multicore computing", w *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Montreal, QC, Canada, wrz. 2012, s. 61–66. doi: 10.1109/ICCD.2012.6378618.

[48] A. Prakash i H. D. Patel, „An Instruction Scratchpad Memory Allocation for the Precision Timed Architecture", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, t. 32, nr 11, s. 1819–1823, lis. 2013, doi: 10.1109/TCAD.2013.2269768.

[49] N. R. Shah, „Memory Issues in PRET Machines", 2008, doi: 10.7916/D80C53NG.

[50] „Vivado Design Suite 7 Series FPGA Libraries Guide". Xilinx. Dostęp: 23 lipiec 2921. [Online]. Dostępne na: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug953-vivado-7series-libraries.pdf

[51] A. K. Pulka, *Techniki heurystyczne w modelowaniu i weryfikacji systemów elektronicznych: wybrane zagadnienia*. Gliwice: Wydawnictwo Politechniki Slaskiej, 2012.

[52] B. Lickly, B. Burgers, i E. A. Lee, „A Timing Requirements-Aware Scratchpad Memory Allocation Scheme for a Precision Timed Architecture", EECS Department, University of California, Berkeley, UCB/EECS-2008-115, wrz. 2008. [Online].

Dostępne na: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-115.html

[53] E. Antolak i A. Pulka, „Flexible hardware approach to multicore time predictable systems design", *IET Circuits Devices Syst.*, mar. 2020, doi: 10.1049/iet-cds.2019.0521.

[54] „Verilog Reference Guide". Dostęp: 29 lipiec 2021. [Online]. Dostępne na: http://in.ncu.edu.tw/ncume_ee/digilogi/vhdl/Verilog_Reference_Guide.pdf

[55] Z. Hajduk, *Wprowadzenie do języka VERILOG*. Legionowo: Wydawnictwo BTC, 2009.

[56] S. Sutherland i D. Mills, *Verilog and System Verilog gotchas: 101 common coding errors and how to avoid them*. New York: Springer, 2007.

[57] „VC707 Evaluation Board for the Virtex-7 FPGA". https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf (dostęp 12 listopad 2019).

[58] S. Churiwala, Red., *Designing with Xilinx® FPGAs: Using Vivado*, 1st ed. 2017. Cham: Springer International Publishing : Imprint: Springer, 2017. doi: 10.1007/978-3-319-42438-5.

[59] S. Furber, *ARM system-on-chip architecture*, 1. publ. Harlow Munich: Addison-Wesley, 2000.

[60] G. C. Buttazzo, *Hard Real-Time Computing Systems*, t. 24. Boston, MA: Springer US, 2011. doi: 10.1007/978-1-4614-0676-1.

[61] E. L. Lamie, *Real-time embedded multithreading: using ThreadX and ARM*. San Francisco, CA : Berkeley, CA: CMP Books ; Distributed to the book trade in the U.S. by Publishers Group West, 2005.

[62] E. Antolak i A. Pułka, „The hierarchical model of a flexible predictable system for time critical data processing", w *Networking issues in innovative applications based on cyber-physical systems paradigm*, 2020, s. 9–22.

[63] A. S. Tanenbaum i T. Austin, *Structured computer organization*, 6th ed. Boston: Pearson, 2013.

[64] J. L. Hennessy, D. A. Patterson, i D. A. Patterson, *Computer architecture: a quantitative approach*. San Francisco, CA: Morgan Kaufmann Publishers, 2003.

[65] W. Stallings, „Reduced instruction set computer architecture", *Proc. IEEE*, t. 76, nr 1, s. 38–55, sty. 1988, doi: 10.1109/5.3287.

[66] Z. Wang, Q. Zuo, i J. Li, „An Intelligent Multi-Port Memory", w *2008 International Symposium on Intelligent Information Technology Application Workshops*, Shanghai, China, grudz. 2008, s. 251–254. doi: 10.1109/IITA.Workshops.2008.231.

[67] H. Bahn i K. Cho, „Evolution-Based Real-Time Job Scheduling for Co-Optimizing Processor and Memory Power Savings", *IEEE Access*, t. 8, s. 152805–152819, 2020, doi: 10.1109/ACCESS.2020.3017014.

[68] M. Pedram, „Power minimization in IC design: principles and applications", *ACM Trans. Des. Autom. Electron. Syst.*, t. 1, nr 1, s. 3–56, sty. 1996, doi: 10.1145/225871.225877.

[69] M. Tahir i K. Javed, *ARM microprocessor systems: cortex-M architecture, programming, and interfacing*. Boca Raton: CRC Press, Taylor & Francis Group, CRC Press is an imprint of the Taylor & Francis Group, an informa business, 2017.

[70] „ARM7TDMI-S Data Sheett, ARM Instruction Set". Dostęp: 29 lipiec 2021. [Online]. Dostępne na: https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf

[71] „Specyfikacja języka C# 6.0". Dostęp: 29 lipiec 2021. [Online]. Dostępne na: https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/language-specification/introduction

[72] P. Puschner, „The single-path approach towards WCET-analysable software", w *IEEE International Conference on Industrial Technology, 2003*, Maribor, Slovenia, 2003, s. 699–704. doi: 10.1109/ICIT.2003.1290740.

[73] G. Xie, G. Zeng, X. Xiao, R. Li, i K. Li, „Energy-Efficient Scheduling Algorithms for Real-Time Parallel Applications on Heterogeneous Distributed Embedded Systems", *IEEE Trans. Parallel Distrib. Syst.*, t. 28, nr 12, s. 3426–3442, grudz. 2017, doi: 10.1109/TPDS.2017.2730876.

[74] „Mälardalen: WCET benchmark programs". http://www.mrtc.mdh.se/projects/wcet/benchmarks.html (dostęp 12 kwiecień 2022).

[75] E. Antolak i A. Pulka, „Energy-Efficient Task Scheduling in Design of Multithread Time Predictable Real-Time Systems", *IEEE Access*, t. 9, s. 121111–121127, 2021, doi: 10.1109/ACCESS.2021.3108912.

[76] „Vivado Design Suite Tutorial: Power Analysis and Optimization", 2022. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug997-vivado-power-analysis-optimization-tutorial.pdf (dostęp 24 kwiecień 2022).

[77] M. Geier, M. Brandle, i S. Chakraborty, „Insert & Save: Energy Optimization in IP Core Integration for FPGA-based Real-time Systems", w *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Nashville, TN, USA, maj 2021, s. 80–91. doi: 10.1109/RTAS52030.2021.00015.

[78] E. Antolak i A. Pułka, „An Analysis of the Impact of Gating Techniques on the Optimization of the Energy Dissipated in Real-Time Systems", *Appl. Sci.*, t. 12, nr 3, Art. nr 3, sty. 2022, doi: 10.3390/app12031630.

[79] S. Wimer i I. Koren, „Design Flow for Flip-Flop Grouping in Data-Driven Clock Gating", *IEEE Trans. Very Large Scale Integr. VLSI Syst.*, t. 22, nr 4, s. 771–778, kwi. 2014, doi: 10.1109/TVLSI.2013.2253338.

[80] E. Bezati, S. Casale-Brunet, M. Mattavelli, i J. W. Janneck, „Clock-Gating of Streaming Applications for Energy Efficient Implementations on FPGAs", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, t. 36, nr 4, s. 699–703, kwi. 2017, doi: 10.1109/TCAD.2016.2597215.

[81] I. Kuon i J. Rose, „Measuring the Gap Between FPGAs and ASICs", *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, t. 26, nr 2, s. 203–215, luty 2007, doi: 10.1109/TCAD.2006.884574.

[82] Ł. Golly, „Concurrent modeling of time predictable multitask electronic systems in SystemC language", PhD thesis (in Polish), Silesian University of Technology, Gliwice, 2016.

[83] A. Pulka, L. Golly, i A. Milik, „SystemC hardware-software design and simulation platform based on AMBA bus”, sty. 2011.

[84] L. Golly i A. Pulka, „Multitask real-time systems modeling in SystemC”, w *2012 International Conference on Signals and Electronic Systems (ICSES)*, Wroclaw, wrz. 2012, s. 1–6. doi: 10.1109/ICSES.2012.6382231.

[85] Ł. Golly i A. Pułka, „Analysis of timing resources for highly predictable real-time systems models”, w *Proceedings of the 20th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2013*, cze. 2013, s. 497–502.

[86] M. Schoeberl, „A Java processor architecture for embedded real-time systems”, *J. Syst. Archit.*, t. 54, nr 1–2, s. 265–286, sty. 2008, doi: 10.1016/j.sysarc.2007.06.001.

[87] B. Akesson i K. Goossens, *Memory Controllers for Real-Time Embedded Systems*, t. 2. New York, NY: Springer New York, 2012. doi: 10.1007/978-1-4419-8207-0.

[88] „The Multi-objective Design Space Exploration of Multiprocessor SOC Architectures for Embedded Multi-media Applications (MULTICUBE) project”. http://www.multicube.eu/ (dostęp 12 listopad 2019).

[89] C. Silvano *i in.*, „MULTICUBE: Multi-objective Design Space Exploration of Multi-core Architectures”, w *2010 IEEE Computer Society Annual Symposium on VLSI*, Lixouri, Greece, lip. 2010, s. 488–493. doi: 10.1109/ISVLSI.2010.67.

[90] B. Forsberg, L. Benini, i A. Marongiu, „HePREM: Enabling predictable GPU execution on heterogeneous SoC”, w *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, mar. 2018, s. 539–544. doi: 10.23919/DATE.2018.8342066.

[91] K. Li, „Energy and time constrained task scheduling on multiprocessor computers with discrete speed levels”, *J. Parallel Distrib. Comput.*, t. 95, s. 15–28, wrz. 2016, doi: 10.1016/j.jpdc.2016.02.006.

[92] J. Li, G. Zheng, H. Zhang, i G. Shi, „Task Scheduling Algorithm for Heterogeneous Real-time Systems Based on Deadline Constraints”, w *2019 IEEE 9th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Beijing, China, lip. 2019, s. 113–116. doi: 10.1109/ICEIEC.2019.8784641.

[93] J. Whitham i N. Audsley, „Time-Predictable Out-of-Order Execution for Hard Real-Time Systems”, *IEEE Trans. Comput.*, t. 59, nr 9, s. 1210–1223, wrz. 2010, doi: 10.1109/TC.2010.109.

[94] R. Wilhelm, „Real time spent on real time”, *Commun. ACM*, t. 63, nr 10, s. 54–60, wrz. 2020, doi: 10.1145/3375545.

[95] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nelis, i J. Reineke, „An extensible framework for multicore response time analysis”, *Real-Time Syst.*, t. 54, nr 3, s. 607–661, lip. 2018, doi: 10.1007/s11241-017-9285-4.

[96] A. U. Rehman *i in.*, „Dynamic Energy Efficient Resource Allocation Strategy for Load Balancing in Fog Environment”, *IEEE Access*, t. 8, s. 199829–199839, 2020, doi: 10.1109/ACCESS.2020.3035181.

[97] H. Chniter, O. Mosbahi, M. Khalgui, M. Zhou, i Z. Li, „Improved Multi-Core Real-Time Task Scheduling of Reconfigurable Systems With Energy Constraints”, *IEEE Access*, t. 8, s. 95698–95713, 2020, doi: 10.1109/ACCESS.2020.2990973.

[98] M. Heath i I. Harris, „A deterministic globally asynchronous locally synchronous microprocessor architecture”, w *Proceedings. 4th International Workshop on Microprocessor Test and Verification - Common Challenges and Solutions*, Austin, TX, USA, 2003, s. 119–124. doi: 10.1109/MTV.2003.1250272.

Dodatki

A. Opublikowane wzory i algorytmy

$$TF_i = \frac{C_i + M_i \cdot M_{dur}}{D_i} \text{ [53], [75]} \tag{32}$$

gdzie:

$TF_i[MHz]$ – Minimalna częstotliwość przetwarzania zadania, pozwalająca wykonać je przed czasem D_i

C_i – Liczba standardowych instrukcji, potrzebnych do wykonania zadania i (bez M_i)

M_i – Liczba instrukcji wymiany danych z innym wątkiem, potrzebnych do wykonania zadania i

M_{dur} – Liczba cykli zegara, potrzebnych do wykonania instrukcji wymiany danych z innym wątkiem

$D_i[\mu s]$ – Maksymalny dopuszczalny czas wykonania zadania i w (deadline)

Algorithm 1: BLIS - tasks scheduling

Given a set of N hardware tasks $\mathbb{T} = \{T_1, T_2, .. T_N\}$ described by pairs of parameters, i.e. $T_i = \{C_i, D_i\}$ and the hardware platform with the system clock *Sys_clk*.

The goal: determine necessary resources (number of processing cores) and allocate all tasks to them.

PHASE 1 (Initial allocation)

Step 1. Basing on the matrix $\mathbb{T}$ **determine** the matrix $\mathbb{TF}$ of task frequencies; Find the total sum of all elements:

$$TS = \sum_{i=1}^{N} TF_i$$

Step 2. Find the minimal number of processing cores, i.e.:

$$CN = \left\lceil \frac{TS}{Sys_clk} \right\rceil$$

Step 3. Find the mean value TF_{MEAN} of elements in the matrix $\mathbb{TF}$

Step 4. Re-arrange the matrix $\mathbb{TF}$ in ascending order of the task frequency parameters (TF_i)

Step 5. for $k = 0$ to $CN–1$ **do loop**

5.1. Take the core # k and allocate to it tasks by jumping from the right to the left edge of the matrix $\mathbb{TF}_{sorted}$, i.e. take the first task with highest TF_i and lowest TF_i alternately. During these jumps control the *SUM* of TF_i of the selected tasks

5.2. If *SUM* > *Sys_clk* **move back one jump** (neglect the last jump and try to take the next task with the lower value of TF_i)

5.3. Remove all already allocated tasks from the matrix $\mathbb{TF}_{sorted}$

When $\mathbb{TF} = \varnothing$ **end loop**.

PHASE 2 (Load balancing)

Step 6. Find the relative difference between the most loaded (ML) and the least loaded (LL) cores δL:

$$\delta L = \left\lceil \frac{SUM_{ML} - SUM_{LL}}{Sys_clk} \right\rceil \cdot 100\%$$

Step 7. Until $\delta L > \varepsilon$

Step 8. Flip tasks between the most and the least loaded core

Step 9. End.

Rys. 111. Algorytm BLIS [53]

Algorithm 2: COTAS - tasks scheduling

Given a set of N hardware tasks $\mathbb{T} = \{T_1, T_2, .. T_N\}$ described by the triple parameters, i.e. $T_i = \{C_i, M_i, D_i\}$ and the hardware platform with the system clock *Sys_clk*.

The goal: determine necessary resources (number of processing cores) and allocate all tasks to them.

Step 1. **Assume** the minimal number of cores *CN* with their $Idle_dur_i$ parameters (particularly if there is no task has a dedicated pipeline and all cores have identical structures *CN*=1)

Step 2. **Evaluate** the *M_dur* from the equation (4)

Step 3. **Determine** the matrix $\mathbb{TF}$ of task frequencies; Find the total sum of all elements:

$$TS = \sum_{i=1}^{N} TF_i$$

Step 4. **Check if** $CN \times Sys_clk > TS$,
If not $CN = CN+1$ and **goto Step 2**
otherwise continue

Step 5. **Find** the mean value TF_{MEAN} of elements in the matrix $\mathbb{TF}$

Step 6. **Re-arrange** the matrix $\mathbb{TF}$ in ascending order of the task frequency parameters (TF_i)

Step 7. **for** $k = 0$ to $CN-1$ **do loop**

7.1. Take the core # *k* and allocate to it tasks by jumping from the right to the left edge of the matrix $\mathbb{TF}_{sorted}$, i.e. take the first task with highest TF_i and lowest TF_i alternately. During these jumps control the *SUM* of TF_i of the selected tasks

7.2. If $SUM > Sys_clk$ **move back one jump** (neglect the last jump and try to take the next task with the lower value of TF_i)

7.3. Remove all already allocated tasks from the matrix $\mathbb{TF}_{sorted}$

When $\mathbb{TF} = \varnothing$ **end loop**.

Step 8. **Execute** Phase 2 of the BLIS Algorithm

Step 9. **End.**

Rys. 112. Algorytm COTAS [53]

Algorithm I (II*)): BLIS II (COTAS II*)) – tasks scheduling

Given a set of N hardware tasks $\mathbb{T} = \{T_1, T_2, .. T_N\}$ described by pairs of parameters, i.e., $T_i = \{C_i, D_i\}$ and the hardware platform with the set of processing cores $\mathbb{PE} = \{pe_1, pe_2, .. pe_k\}$.

The goal: determine necessary resources (number of processing cores) and allocate all tasks to them.

PHASE 1 (Initial allocation)

{*)**Step 1** In case of cooperating tasks use the COTAS II version **Calculate** the *M_dur* from equation (3)}

Step 1'Based on the matrix $\mathbb{T}$, **determine** the matrix $\mathbb{TF}$, of task frequencies; find the total sum of all elements:

$$TS = \sum_{i=1}^{N} TF_i$$

Step 2 Calculate the processing frequency of the system:

$$F_{sys} = (1 + F_{margin}) \cdot \frac{\sum_{i=1}^{N} TF_i}{CN}$$

Step 3 Check if the processing frequency $F_{sys} \leq F_{max}$:
If not increase CN and go return to Step 2

Step 4 Find the mean value, TF_{MEAN}, of elements in the matrix $\mathbb{TF}$

Step 5 Re-arrange the matrix $\mathbb{TF}$ in ascending order of the task frequency parameters (TF_i)

Step 6 for $k = 0$ to $CN–1$ **do loop**

6.1. Take the core # k, and allocate to it tasks with the highest and lowest value of TF from $\mathbb{TF}_{sorted}$, i.e., take the last task with highest TF_i and the first one with the lowest TF_i, alternately. During these jumps, control the SUM of TF_i of the selected tasks

6.2. If $SUM > F_{sys}$ **move back one jump** (neglect the last jump and try to take the next task with the lower value of TF_i)

6.3. Remove all of the already allocated tasks from the matrix $\mathbb{TF}_{sorted}$

If $\mathbb{TF} = \varnothing$ **end loop else repeat Step 6**.

PHASE 2 (Load balancing)

Step 7 Find the relative difference between the most loaded (ML) and the least loaded (LL) cores δL:

$$\delta L = \frac{\lceil SUM_{ML} - SUM_{LL} \rceil}{F_{SYS}} \cdot 100\%$$

Step 8 Until $\delta L > \varepsilon$

Step 9 Flip tasks between the most and the least loaded core

Step 10 End.

Rys. 113. Algorytm BLIS II i COTAS II [75]

Methodology I: SFERA

Step 1 **Set** the frequency of the system clock (use the reasonable, i.e. possible parameter for your platform)

Step 2 According to the specificity of the application and tasks executed by the system **run one** of the scheduling algorithms (BLIS II, COSTAS II or STODER II)

Step 3 Estimate total power consumed by the system:

$$P_{total} = \sum_{i}^{N} \boldsymbol{P}(pe_i)$$

Step 4 If $\boldsymbol{P_{total} \leq P_{constraint}}$ **terminate**

Step 5 If not increase *CN* and **go back** to Step 1

Rys. 114. Metodologia SFERA[75]

Methodology II: MAXPRO

Given a set of *N* hardware tasks $\mathbb{T} = \{T_1, T_2, .. T_N\}$ described by pairs of parameters, i.e. $T_i = \{C_i, D_i\}$ and the hardware platform with the set of processing cores $\mathbb{PE} = \{pe_1, pe_2, .. pe_k\}$.

The goal: determine the most economical implementation of the system in terms of energy consumption.

Step 1 **Analyze** a set of *N* hardware tasks $\mathbb{T} = \{T_1, T_2, .. T_N\}$ and find the maximal number of cores (find initial mapping),

Step 2 Depending on the tasks' and cores' structure run appropriate scheduling algorithm **BLIS II (COTAS II)** or **STODER II**

Step 3 For the most loaded core (ML) find the sum of TF_i task assigned to it, i.e.:

$$SUM_{ML} = \sum_{i=1}^{N_ML} TF_i$$

Step 4 Calculate the processing frequency of system, i.e:

$$F_{sys} = SUM_{ML}$$

Step 5 Estimate total power dissipated the system:

$$P_{total} = \sum_{i}^{N} \boldsymbol{P}(pe_i)$$

Step 6 **End.**

Rys. 115. Metodologia MAXPRO[75]

B. Dane z procesu weryfikacji przewidywalności czasowej metodologii harmonogramowania

Tab. 26. Dane z procesu weryfikacji - Scenariusz A (MINRES)

ThID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
1	1902114	1467525,25	1506787,88	1468464,65	1470787,88	1469777,09	2510870
2	1448928	1128333,33	1234373,74	1128414,14	1130545,45	1129656,42	3259998
3	1280929	1211212,12	1215848,48	1211212,12	1214464,65	1213284,13	3036551
4	855631	331060,61	406818,18	334383,84	337989,90	505181,82	7310647
5	1922519	1525858,59	1526585,86	1525919,19	1526575,76	1526236,99	2417875
6	1581164	1307343,43	1309545,45	1307353,54	1308979,80	1308114,44	2817410
7	1980426	1871040,40	1883292,93	1871101,01	1881717,17	1878642,13	1963054
8	1612365	1425595,96	1427111,11	1425666,67	1426929,29	1426541,27	2583200
9	1088925	914727,27	918272,73	914797,98	918030,30	916480,46	4017393
10	1878523	1114676,77	1181757,58	1134636,36	1148262,63	1715580,81	7310647
11	1311330	1107575,76	1109434,34	1107707,07	1109191,92	1108548,91	3323028
12	599793	533121,21	535848,48	533202,02	535414,14	534423,19	6866146
13	1882301	1016171,72	1117313,13	1016202,02	1020111,11	1017614,19	3617956
14	720337	643474,75	645808,08	643474,75	645292,93	644457,52	5706289
15	367229	306414,14	309282,83	306444,44	308939,39	307680,25	11897186
16	989144	730101,01	810080,81	744828,28	747373,74	1119787,88	7310647
17	1251289	1143505,05	1166484,85	1145313,13	1149585,86	1722242,42	7310647
18	1225434	1074070,71	1075717,17	1074070,71	1075404,04	1074704,34	3433072
19	1016979	901959,60	903292,93	901959,60	903141,41	902476,66	4077482
20	1602334	1367313,13	1368060,61	1367313,13	1368010,10	1367594,23	2694277
21	769698	689858,59	690222,22	689858,59	690212,12	690029,02	5326089
22	1099207	921787,88	925202,02	921787,88	924919,19	923245,11	3985508
23	1256100	1134686,87	1137252,53	1134737,37	1136606,06	1135678,27	3242432

TID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
24	902913	572060,61	574282,83	572131,31	574121,21	860186,87	7310647
25	422956	371636,36	373828,28	371727,27	373515,15	372350,51	9841992
26	1900332	1689737,37	1692131,31	1689757,58	1691868,69	1691010,45	2179677
27	1891852	1311252,53	1430202,02	1311343,43	1313919,19	1312596,36	2789856
28	1924117	1509252,53	1576191,92	1514111,11	1516858,59	1515495,84	2432182
29	1192492	1148151,52	1155434,34	1148262,63	1153454,55	1151375,29	3199293
30	989968	835434,34	837787,88	835545,45	837565,66	836607,06	4399835
31	1442498	1290494,95	1292404,04	1290575,76	1291969,70	1291276,22	2853262
32	1034294	1000505,05	1002656,57	1000545,45	1002131,31	1001290,76	3678476
33	1415204	1189444,44	1191747,47	1189454,55	1191464,65	1190363,78	3095761
34	1034817	346222,22	346949,49	346272,73	346929,29	346685,38	10539457
35	974619	845636,36	847616,16	845757,58	847303,03	846599,29	4347526
36	1162232	1029121,21	1031161,62	1029121,21	1030929,29	1030100,09	3576440
37	1250616	792797,98	899020,20	841242,42	843525,25	842355,71	4368191
38	1378492	758000,00	822545,45	758000,00	761929,29	759853,98	4843500
39	187292	167858,59	170151,52	167868,69	169616,16	168936,17	21505141
40	1927832	1386666,67	1440171,72	1388737,37	1391202,02	1389646,19	2651913
41	1553470	1103717,17	1140333,33	1106444,44	1109323,23	1107878,17	3324578
42	1094883	949828,28	952575,76	949929,29	952414,14	950990,58	3870925
43	1834138	865979,80	870262,63	866010,10	869010,10	868068,14	4239031
44	487666	454737,37	457464,65	454757,58	457151,52	456094,86	8050285
45	543915	462393,94	538646,46	503989,90	507959,60	505722,10	7274291
46	492035	423787,88	437414,14	429141,41	432525,25	431212,89	8502861
47	1267142	1118575,76	1120626,26	1118646,46	1120383,84	1119537,30	3288930
48	1551826	935838,38	1062808,08	951525,25	955262,63	953458,68	3862745
49	987253	953323,23	956757,58	953343,43	956181,82	955253,05	3855426
50	1821961	1224767,68	1225494,95	1224808,08	1225464,65	1225184,94	3006845
51	817140	767848,48	770505,05	767939,39	770010,10	768956,44	4784086
52	1624262	1185676,77	1227878,79	1188030,30	1191959,60	1190151,26	3096263

ThID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
53	731547	663707,07	665939,39	663737,37	665474,75	664626,04	5533214
54	1003412	926777,78	928595,96	926888,89	928101,01	927494,81	3971670
55	1725713	987666,67	1063545,45	1009535,35	1014000,00	1011741,38	3642312
56	990527	926303,03	929474,75	926343,43	928494,95	927858,53	3970242
57	1446128	1163676,77	1210606,06	1169101,01	1171878,79	1170556,75	3146338
58	768467	486545,45	527222,22	515787,88	519696,97	517411,79	7093467
59	1573557	1226000,00	1267777,78	1244232,32	1247333,33	1245808,94	2957764
60	1813628	1712323,23	1717707,07	1712454,55	1716383,84	1715148,22	2149960

Tab. 27. Dane z procesu weryfikacji - Scenariusz B (MAXPRO)

ThID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
1	1902114	1144552,63	1266184,21	1221684,21	1230631,58	1226611,10	4349745
2	1448928	1216710,53	1403605,26	1216710,53	1218552,63	1217717,15	4380308
3	1280929	737236,84	748447,37	737289,47	746000,00	742284,28	7169222
4	855631	243263,16	410421,05	281947,37	291526,32	286128,07	18389131
5	1922519	1490789,47	1491500,00	1490789,47	1491473,68	1491132,22	3581259
6	1581164	1160631,58	1163868,42	1160631,58	1163526,32	1161907,93	4588762
7	1980426	1076710,53	1105000,00	1076868,42	1094447,37	1088568,82	4896054
8	1612365	1196105,26	1202000,00	1196184,21	1201000,00	1199622,78	4443003
9	1088925	648973,68	654605,26	649210,53	653605,26	651229,05	8160090
10	1878523	827263,16	927657,89	827315,79	844157,89	836483,86	6363431
11	1311330	1028921,05	1031763,16	1028947,37	1031105,26	1029929,12	5181010
12	599793	472815,79	475342,11	472815,79	474921,05	473868,40	11174862
13	1882301	730631,58	885605,26	792842,11	804789,47	799481,61	6652416
14	720337	523500,00	526473,68	523500,00	525947,37	524739,25	10102969
15	367229	287473,68	291473,68	287552,63	290052,63	288932,07	18138716

ThID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
16	989144	710631,58	862105,26	781263,16	786605,26	784041,37	6783668
17	1251289	1126789,47	1163552,63	1141842,11	1144710,53	1143239,74	4662909
18	1225434	972815,79	975552,63	972815,79	975078,95	974057,69	5471914
19	1016979	855921,05	857605,26	855921,05	857236,84	856596,81	6217212
20	1602334	1277052,63	1281684,21	1277131,58	1280552,63	1278442,44	4175894
21	769698	449394,74	449631,58	449394,74	449605,26	449500,00	11768294
22	1099207	693210,53	698394,74	693421,05	697052,63	695377,58	7649114
23	1256100	916236,84	919078,95	916236,84	918552,63	917219,53	5805033
24	902913	343131,58	423473,68	347131,58	360973,68	353379,35	14926490
25	422956	304052,63	308368,42	304052,63	307210,53	305400,80	17097334
26	1900332	1374289,47	1379947,37	1374315,79	1379026,32	1376768,67	3884605
27	1891852	1076368,42	1319868,42	1139552,63	1146289,47	1142432,72	4662909
28	1924117	1283105,26	1424973,68	1322710,53	1330552,63	1326972,63	4020464
29	1192492	661131,58	674210,53	661447,37	670315,79	667128,57	7960622
30	989968	721052,63	723763,16	721052,63	723315,79	722120,20	7377758
31	1442498	1146289,47	1149078,95	1146368,42	1148605,26	1147504,65	4662909
32	1034294	877157,89	879447,37	877184,21	878894,74	878006,66	6046238
33	1415204	1081315,79	1086210,53	1081315,79	1085105,26	1082803,16	4921959
34	1034817	523894,74	524736,84	523894,74	524710,53	524139,68	10102969
35	974619	809026,32	810947,37	809026,32	810763,16	809709,97	6577580
36	1162232	880236,84	893315,79	880368,42	890526,32	885491,31	6022923
37	1250616	715500,00	953736,84	849710,53	854736,84	852193,96	6217212
38	1378492	660105,26	879973,68	720052,63	728289,47	724047,90	7338613
39	187292	114973,68	123842,11	114973,68	122447,37	118710,50	42723761
40	1927832	1027500,00	1186684,21	1068789,47	1082026,32	1074428,14	4970315
41	1553470	954605,26	1073394,74	978000,00	988973,68	982805,87	5420487
42	1094883	678973,68	685973,68	679157,89	683868,42	681172,71	7771515
43	1834138	704447,37	944210,53	819263,16	827973,68	823492,44	6463598
44	487666	451263,16	453368,42	451263,16	453105,26	452264,68	11657273

ThID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
45	543915	500552,63	500789,47	500552,63	500763,16	500657,89	10569260
46	492035	467447,37	486657,89	483473,68	485710,53	484652,97	10880121
47	1267142	1021026,32	1025447,37	1021052,63	1024578,95	1023163,44	5212096
48	1551826	806947,37	1025710,53	872394,74	881131,58	876695,19	6062933
49	987253	902157,89	904236,84	902157,89	903894,74	903136,18	5906351
50	1821961	1418210,53	1419052,63	1418552,63	1419026,32	1418798,06	3763049
51	817140	544605,26	544842,11	544605,26	544815,79	544710,41	9714394
52	1624262	906236,84	1033315,79	942315,79	954368,42	947901,81	5618517
53	731547	551078,95	552789,47	551105,26	552368,42	551859,19	9636678
54	1003412	792342,11	794342,11	792342,11	793684,21	792928,46	6709984
55	1725713	862605,26	1123000,00	996921,05	1005763,16	1001773,76	5320897
56	990527	712157,89	715973,68	712184,21	715394,74	714078,89	7440698
57	1446128	1200605,26	1280473,68	1231131,58	1233763,16	1232272,00	4329021
58	768467	457447,37	521921,05	481552,63	486105,26	483587,12	10905449
59	1573557	1098368,42	1191605,26	1147947,37	1156447,37	1152311,17	4629880
60	1813628	1010894,74	1016578,95	1010921,05	1015868,42	1013234,51	5261315

Tab. 28. Dane z procesu weryfikacji - Scenariusz C (SFERA)

ThID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
1	1902114	1366333,33	1437549,02	1374098,04	1379901,96	1376810,93	3225326
2	1448928	1222039,22	1373058,82	1222137,25	1224176,47	1222846,40	3631311
3	1280929	1056196,08	1063686,27	1056196,08	1062607,84	1059838,28	4186339
4	855631	314431,37	454764,71	325705,88	331588,24	328994,14	13326388
5	1922519	987372,55	987843,14	987372,55	987823,53	987569,96	4495025
6	1581164	1348274,51	1350901,96	1348313,73	1350627,45	1349431,58	3293753
7	1980426	1815019,61	1829000,00	1815078,43	1826450,98	1822353,29	2440873

TID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
8	1612365	1302784,31	1306862,75	1302823,53	1306411,76	1304390,68	3404080
9	1088925	870980,39	876803,92	870980,39	875705,88	873542,48	5078817
10	1878523	1323607,84	1331921,57	1323607,84	1329254,90	1325704,26	3351031
11	1311330	1125941,18	1129294,12	1126098,04	1128686,27	1127523,95	3936753
12	599793	490196,08	495686,27	490235,29	495000,00	492533,48	8956943
13	1882301	920647,06	1016784,31	930176,47	938823,53	934392,48	4743803
14	720337	604725,49	607686,27	604784,31	606941,18	605791,41	7294200
15	367229	305137,25	309843,14	305372,55	309294,12	307225,14	14319852
16	989144	833490,20	985215,69	862274,51	866901,96	864791,15	5119689
17	1251289	1162019,61	1193490,20	1170019,61	1175529,41	1172768,45	3786036
18	1225434	1126137,25	1128588,24	1126196,08	1128039,22	1127160,73	3936552
19	1016979	951058,82	953235,29	951196,08	952823,53	952112,94	4661762
20	1602334	1370666,67	1373215,69	1370725,49	1372588,24	1371609,35	3241866
21	769698	446431,37	446666,67	446470,59	446647,06	446540,95	9841381
22	1099207	891764,71	895666,67	891803,92	895313,73	893809,78	4962040
23	1256100	1090313,73	1092960,78	1090509,80	1092490,20	1091619,66	4052333
24	902913	532803,92	577392,16	532941,18	538352,94	535614,85	8266759
25	422956	347411,76	351843,14	347450,98	351117,65	349533,60	12515920
26	1900332	1468901,96	1472843,14	1469039,22	1471960,78	1470565,76	3020372
27	1891852	1221490,20	1387647,06	1251000,00	1257568,63	1254153,60	3543111
28	1924117	1459411,76	1588705,88	1492627,45	1497843,14	1495582,04	2972676
29	1192492	1016764,71	1025960,78	1016901,96	1024450,98	1021497,79	4341785
30	989968	847333,33	849098,04	847352,94	848686,27	848099,48	5210142
31	1442498	1419745,10	1423431,37	1419745,10	1422647,06	1421289,83	3126085
32	1034294	1027549,02	1029294,12	1027607,84	1028843,14	1028202,60	4318629
33	1415204	1151058,82	1152901,96	1151078,43	1152058,82	1151764,58	3859365
34	1034817	516784,31	517725,49	517000,00	517647,06	517337,14	8518170
35	974619	898352,94	900333,33	898509,80	899862,75	899181,55	4920690
36	1162232	973117,65	975411,76	973117,65	975058,82	974185,75	4550191

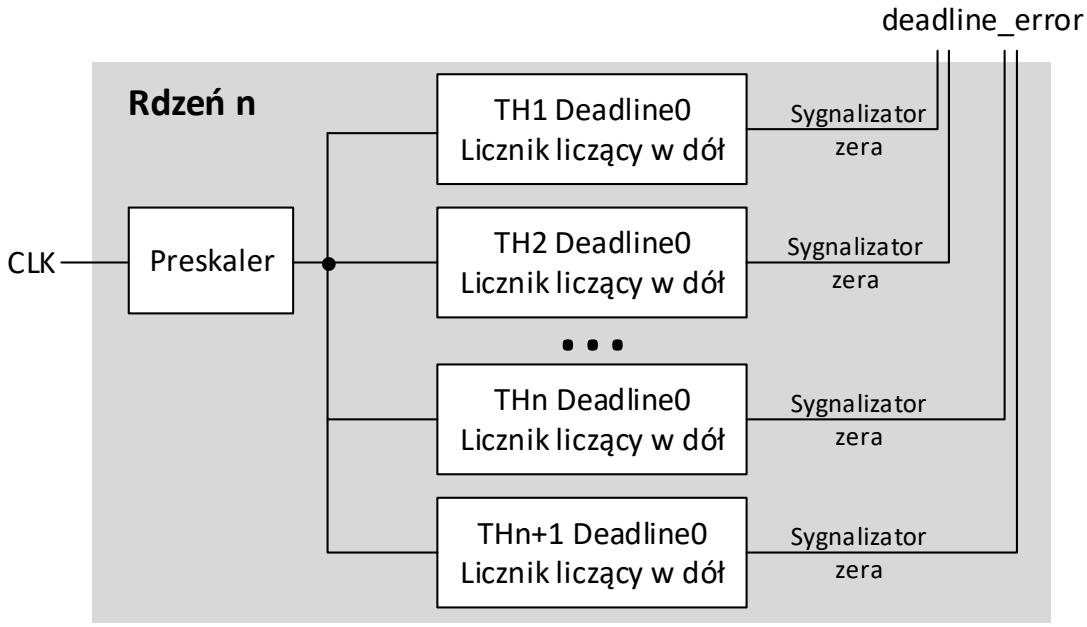
ThID	D_i[ns]	Symulator		Pomiar			
		Min[ns]	Max[ns]	Min[ns]	Max[ns]	Średnie wykonanie[ns]	Liczba wykonań
37	1250616	778235,29	985098,04	823980,39	830117,65	827280,49	5354868
38	1378492	766705,88	957980,39	803745,10	810078,43	806474,24	5483514
39	187292	145588,24	150509,80	145764,71	150137,25	148410,30	28655786
40	1927832	1250392,16	1347941,18	1269509,80	1276019,61	1272172,98	3473428
41	1553470	1124588,24	1195176,47	1138039,22	1144588,24	1140996,10	3896029
42	1094883	921588,24	925098,04	921588,24	924725,49	923034,31	4810698
43	1834138	809058,82	811196,08	809098,04	811019,61	810036,88	5458244
44	487666	475627,45	477725,49	475666,67	477470,59	476408,48	9262476
45	543915	459137,25	459372,55	459156,86	459333,33	459249,55	9551929
46	492035	436784,31	448784,31	445980,39	446627,45	446216,84	9841381
47	1267142	1131019,61	1133294,12	1131098,04	1132823,53	1131889,74	3921612
48	1551826	883549,02	1062039,22	920764,71	925686,27	922922,57	4810805
49	987253	909235,29	911980,39	909352,94	911333,33	910285,10	4862800
50	1821961	1585019,61	1585960,78	1585215,69	1585862,75	1585540,30	2805461
51	817140	794901,96	797019,61	795019,61	796666,67	795743,63	5557485
52	1624262	1163196,08	1282058,82	1177333,33	1182137,25	1179416,78	3762881
53	731547	692607,84	696215,69	692607,84	695666,67	694356,31	6390063
54	1003412	911764,71	915803,92	912039,22	915294,12	913532,55	4851007
55	1725713	1058117,65	1133431,37	1061725,49	1067823,53	1064812,93	4168114
56	990527	819352,94	824666,67	819372,55	824196,08	822042,80	5399251
57	1446128	1249000,00	1316921,57	1265745,10	1270666,67	1268436,85	3497550
58	768467	455901,96	526666,67	480901,96	486901,96	483621,47	9117750
59	1573557	1197117,65	1238058,82	1214431,37	1219156,86	1216611,85	3650316
60	1813628	1534058,82	1540254,90	1534058,82	1539843,14	1536660,24	2894523

C. Mechanizm kontroli czasu wykonania zadania

Każdy z wątków pracujących w systemie, oprócz zestawu banków rejestrów posiada również zestaw maksymalnie 16 rejestrów odliczających czas jego pracy. Liczba tych rejestrów jest konfigurowalna dla każdego rdzenia. W standardowej konfiguracji przewidziano jeden taki rejestr.

Rejestry te (Rys. 116) odliczają w dół, aż do osiągnięcia wartości zera. Gdy stan takiego rejestru wynosi zero, na sygnale wyjściowym takiego rejestru ustawiany jest wysoki stan logiczny. Wszystkie takie sygnały wyjściowe zostają złączone w jedną magistralę nazwaną „*deadline_error”*. Za pomocą kolejności sygnałów magistrali można określić do którego wątku i licznika należy dany sygnał. Sygnały wyjściowe są sygnałami wyjściowymi rdzenia systemu.

Liczniki odliczają takty zegara przechodzące przez preskaler. Ustawiając parametry rdzenia DEAD_PRESC (dzielnik preskalera) oraz DEAD_PRESC_WIDTH (liczba bitów dzielnika preskalera) można w bardzo szeroki sposób dostosować zliczane jednostki czasu.



Rys. 116. Liczniki czasu pracy wątków

Standardowe instrukcje ISA uzupełniono o instrukcje ustawiania wartości omawianych wyżej rejestrów. Instrukcja ta, to STD (set deadline), której argumentami są numer licznika (C) oraz rejestr (R), którego wartością ma być załadowany dany licznik.

Przykład programu z użytą instrukcją STD pokazano na Rys. 117. W linii 3 i 4 do pierwszego licznika czasu wykonania wpisywana jest wartość 120. Zakładając, że preskaler jest wyłączony (parametr DEAD_PRESC jest równy 0, tzn. liczniki zliczają takty zegara CLK), a program pracuje w pierwszym wątku rdzenia pierwszego. Czas wykonania instrukcji od linii 5 do 9 musi być mniejszy niż 120 cykli zegara. W przeciwnym razie magistrala „*deadline_error”* zwróci błąd czasu wykonania zadania, licznika pierwszego, wątku pierwszego oraz rdzenia pierwszego. Aby błąd nie został zwrócony, instrukcje od linii 10 do linii 14 muszą być wykonane przed czasem równym 200 cyklom zegara.

```
1   MOV  .R0, 0x02
2   LDR  .R0, [R0]
3   MOV  .R1, #120
4   STD  .C0 , R1
5
6   //-- 1 Fragment kodu programu --
7
8   MOV  .R1, #200
9   STD .C0, R1
10
11  //-- 2 Fragment kodu programu --
12
13  MOV  .R1, #50
14  STD .C0, R1
```

Rys. 117. Przykład instrukcji STD

Projektowany system jest przewidywalny czasowo podczas poprawnego pisania programów oraz konfiguracji systemu, powyższy mechanizm nie jest potrzebny. Został on jednak zaimplementowany jako dodatkowa kontrola poprawności konfiguracji systemu. Dzięki takiemu mechanizmowi można sprawdzać długość wykonywania całych wątków, jak i również krytycznych czasowo fragmentów zadań. Liczniki te można również odczytywać i używać ich wartości w wykonywanych programach.