PRACA MAGISTERSKA

„Implementacja modułu predykcji skoków mikroprocesora RISC”

Michał GOLD
Nr albumu 293468

Kierunek: Mikroinformatyka Systemów Cyfrowych
Specjalność: Projektowanie Systemów Cyfrowych

PROWADZĄCY PRACĘ
dr hab. inż. Robert Czerwiński Prof. PŚ
KATEDRA Systemów Cyfrowych
Wydział Automatyki, Elektroniki i Informatyki

GLIWICE Rok 2024

Tytuł pracy:

Implementacja modułu predykcji skoków mikroprocesora RISC

Streszczenie:

Celem pracy było zaprojektowanie, implementacja, oraz zbadanie efektywności modułu predykcji skoków dla mikroprocesora opartego o specyfikację RISC-V. Model zintegrowano z istniejącym już projektem procesora, stworzonym w ramach innych prac magisterskich. W procesorze wykryte zostały liczne błędy, które w większości zostały przez autora niniejszej pracy naprawione. Stworzone zostało środowisko testowe, pozwalające na weryfikację funkcjonalną systemu procesorowego, jak i czytelny podgląd wewnętrznych sygnałów. Zaimplementowane zostały łącznie 7 różnych algorytmów predykcji. Sam moduł predykcji skoków zaimplementowano w sposób modularny, co pozwoliło na łatwe podmienianie testowanych algorytmów predykcji. Przeprowadzono eksperymenty symulacyjne, wyznaczając efektywność predykcji każdego z zaimplementowanych algorytmów.

Słowa kluczowe:

RISC, RISC-V, Predyktor Skoków, Potok Mikroprocesora, Mikroprocesor

Thesis title:

Implementation of branch prediction module of RISC microprocessor

Abstract:

The objective of this thesis was to design, implement, and measure the efficiency of a branch prediction module for a microprocessor based on the RISC-V specification. The model was integrated with an existing processor project created as part of other master's theses. Numerous errors were detected in the processor, most of which were corrected by the author of this thesis. A test environment was created, allowing for functional verification of the processor system as well as a clear view of internal signals. A total of 7 different prediction algorithms were implemented. The branch prediction module itself was implemented in a modular manner, which facilitated easy replacement of the tested prediction algorithms. Simulation experiments were conducted to determine the prediction efficiency of each implemented algorithm.

Keywords:

RISC, RISC-V, Jump Predictor, Microprocessor Pipeline, Microprocessor

Spis treści

1. Wstęp

1.1. Wprowadzenie

Praca magisterska dotyczy modułu predyktora skoków dla mikroprocesora RISC (Reduced Instruction Set Computing).

Co do zasady, praca dowolnego procesora sprowadza się do 3 elementów: wczytania instrukcji do wykonania, wykonania obliczeń związanych z instrukcją oraz aktualizacji stanu systemu procesorowego. Instrukcje występują jednak w wielu różnych rodzajach. Mogą się różnić zarówno czasem potrzebnym na wykonanie, jak i wymaganymi do zaangażowania podmodułami mikroprocesora. Sprawia to, że wiele komponentów mikroprocesora pozostaje przez długi czas w stanie bezczynności, podczas gdy potencjalnie mogłyby już zaczynać przetwarzać kolejną instrukcję. Szczególnie widać to w procesorach typu RISC, gdzie instrukcje skonstruowane są z reguły w sposób bardzo modularny, przez co łatwo określić które podmoduły i kiedy będą wymagane do jej wykonania. Struktura mikroprocesora implementująca podział wykonywanych instrukcji na mniejsze, niezależne od siebie etapy, pozwalające na przetwarzanie wielu instrukcji naraz, nazywana jest strukturą potokową. Rozwiązanie to pozwala przyśpieszyć pracę procesora wielokrotnie, zależnie od liczby wydzielonych podetapów.

Świat nie jest jednak taki kolorowy. Przy opisie idei potoku, przyjęto ważne założenie, że instrukcje są od siebie niezależne. Istnieją jednak instrukcje, od których wyniku zależy to, którą instrukcję mikroprocesor powinien wykonać jako kolejną. Takie operacje nazywane są skokami. Zastosowanie potoku sprawia, że mikroprocesor może zacząć wykonywać kolejne instrukcje, podczas gdy jeszcze nie jest znany wynik skoku. Oznacza to, że procesor może wykonać operacje, które nigdy nie powinny mieć miejsca.

W celu minimalizacji szansy na wystąpienie takiej niechcianej sytuacji, niezbędne jest zaimplementowanie modułu predykcji skoków. Moduł taki powinien, na podstawie odpowiednio wybranej heurystyki, przewidzieć sekwencję wczytywanych instrukcji. Im większą miałby dokładność, tym mniej czasu pracy mikroprocesora zostałoby zmarnowane na niepoprawne instrukcje.

1.2. Cel i zakres pracy

Celem pracy magisterskiej jest zaprojektowanie, implementacja i zbadanie efektywności modułu predykcji skoków dla mikroprocesora RISC, bazującego na popularnej specyfikacji RISC-V [1]. W ramach pracy zostanie zaimplementowanych kilka różnych, konfigurowalnych wersji predyktora skoków. Każda z nich charakteryzować się będzie innymi cechami, które ujawniają się w różnych sytuacjach. Prace badawcze będą obejmowały rozbudowane porównanie tych wariantów.

Szczegółowy zakres pracy obejmuje szereg kluczowych zadań. W pierwszej kolejności zostanie wykonana analiza aktualnego stanu wiedzy na temat algorytmów predykcji skoków z wybranymi implementacjami owego modułu w istniejących procesorach.

Druga część analizy będzie obejmowała implementację mikroprocesora RISC-V. Ta implementacja powstała w wyniku realizacji projektu PBL oraz innych prac magisterskich [2][3] i nie obejmowała modułu predykcji skoków. Zadaniem autora jest dokładna analiza działania otrzymanego projektu z usunięciem niewykrytych dotychczas błędów oraz rozbudowa go o właśnie moduł predykcji skoków.

Projekt predyktora ma być wykonany w sposób możliwie modularny. Oznacza to, że w razie wystąpienia chęci zmiany sposobu działania algorytmu predykcji potrzebna będzie minimalna modyfikacja projektu, jedynie w części odpowiedzialnej stricte za algorytm.

W celu przetestowania zaimplementowanych modułów należy stworzyć mechanizm weryfikacji funkcjonalnej projektu. W tym celu zostanie opracowany model referencyjny mikroprocesora, będący jego bardzo uproszczoną wersją, bez cech takich jak przetwarzanie potokowe i predykcja skoków. Ważnym aspektem w pracy jest stworzenie rozbudowanego środowiska testowego pozwalającego na wykonanie dowolnego testu i uruchomienia symulacji, dla dowolnego programu testowego, przy dowolnej konfiguracji projektu mikroprocesora, przy użyciu jednej komendy, bez potrzeby na modyfikację jakiegokolwiek pliku źródłowego i ręcznego uruchamiania kolejnych pomniejszych skryptów.

2. Analiza tematu

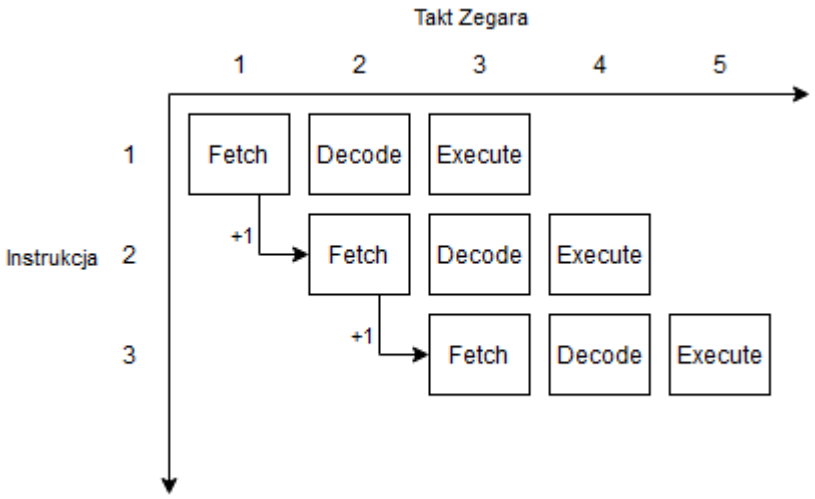
2.1. Motywacja dla predyktorów skoków

Na początku historii komputerów wielkim osiągnięciem była w ogóle sama budowa jednostki pozwalającej na przetwarzanie informacji cyfrowych. Nie trudno jednak zauważyć, że wraz z postępem technologicznym, zapotrzebowanie na szybkie przetwarzanie informacji wzrasta. Dotyczy to bardzo wielu dziedzin – logistyki, finansów, analizy dużych zbiorów danych, badań naukowych oraz rozrywki (głównie gier komputerowych). Sercem prawie każdego systemu przetwarzania danych jest procesor, ponieważ to właśnie on odpowiada za wykonywanie zadanych mu instrukcji. Im szybciej wykona zadany zbiór instrukcji, tym lepiej.

Zwiększenie wydajności procesora można przeprowadzić na 2 ogólne sposoby: horyzontalny (scale-out) i wertykalny (scale-up) [4]. Horyzontalność oznacza zwiększenie jednostek przetwarzających dane, czyli zwiększenie liczby procesorów albo rdzeni, oddelegowanie pewnych operacji do osobnych modułów, ogólnie zwiększenie równoległości systemu. Jest to bardzo dobra metoda na przyśpieszenie przetwarzania danych, ponieważ pozwala zwielokrotnić moc obliczeniową, proporcjonalnie do ilości zużytych zasobów. Charakteryzuje się jednak jedną znaczącą wadą – z reguły takie systemy wymagają odpowiednio przygotowanego programu. Jeśli zastosowany program nie zawiera elementów, które są od siebie niezależne, pozwalając na ich jednoczesne wykonywanie na osobnych jednostkach, to więcej rdzeni ani procesorów nie wpłynie w żaden sposób na wydajność systemu.

Wertykalne rozbudowanie systemu określa bezpośrednie polepszenie jego parametrów. Może to być np. zwiększenie częstotliwości taktowania zegara albo zmiana wewnętrznej implementacji poszczególnych modułów. Taki rodzaj najczęściej nie wymaga specjalnych programów wspierających współbieżność. Z reguły każdy program który działa na jednym procesorze, zadziała również na szybszym wariancie, ale szybciej. Problemem w takim ulepszaniu procesora jest jednak to, że nie można tego robić w nieskończoność. Ograniczeniem są zasady fizyki. Aktualny stan technologi dosięga już granicy minimalnego rozmiaru tranzystora, co wpływa również na maksymalną możliwą częstotliwość taktowania zegara. Projektanci, chcąc więc przyśpieszyć działanie procesora, muszą wykazać się kreatywnością. Jednym z powszechnie stosowanych rozwiązań tego problemu jest właśnie moduł predykcji skoków.

Jeśli niemożliwym jest skrócenie czasu wykonywania pojedynczej instrukcji, to trzeba zastosować inny rodzaj optymalizacji. Większość mikroprocesorów RISC posiada strukturę potokową [5].

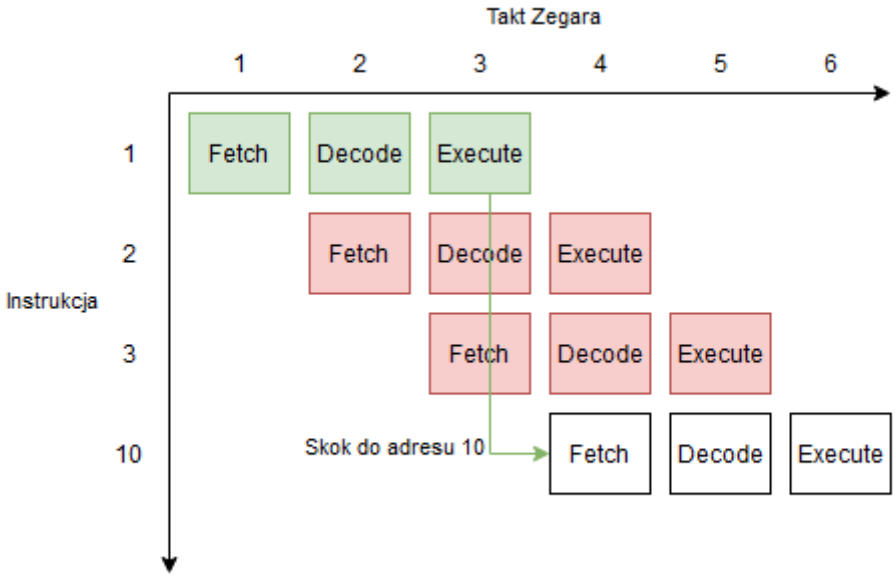


Rys.2.1. *Diagram przykładowego prostego potoku*

Struktura potokowa oznacza, że wykonanie pojedynczej instrukcji rozdzielane jest na różne fazy, wzajemnie od siebie niezależne. Prostym rodzajem potoku, pozwalającym na łatwe zobrazowanie tego aspektu, jest podział na fazy Fetch, Decode i Execute. Podczas fazy Fetch, mikroprocesor wczytuje z pamięci programu daną instrukcję. Podczas fazy Decode wczytana instrukcja zamieniana jest na sygnały kontrolne i wartości operandów, które zostaną użyte w obliczeniach. W fazie Execute wykonywane są właśnie obliczenia, jak i przeprowadzane zostają wszelkie efekty uboczne operacji, jak np. zapisanie wyniku operacji do rejestru albo pamięci RAM (Random Access Memory). Na Rys. 2.1 zaprezentowano przykładowy przebieg wykonania trzech kolejnych instrukcji. W pierwszym takcie sygnału zegarowego mikroprocesor wczytuje pierwszą instrukcję. W kolejnym jednocześnie dekoduje pierwszą instrukcję oraz wczytuje kolejną instrukcję – z adresu o 1 większego. W kolejnym takcie jednocześnie wykonuje pierwszą, dekoduje drugą i wczytuje jeszcze kolejną. Ponieważ każda z tych faz może być wykonywana jednocześnie, wykonanie wszystkich trzech instrukcji zajmuje jedynie 5 taktów sygnału zegara. Gdyby nie można było wykonywać tych instrukcji współbieżnie, to mikroprocesor potrzebowałby 9 taktów.

Jednym z rodzajów instrukcji, jakie powszechnie implementowane są w mikroprocesorach, są skoki. Skok oznacza, że adres kolejnej instrukcji jest zależny od wyniku obliczeń aktualnej instrukcji. Wyróżnia się skoki bezwarunkowe i warunkowe. Skoki warunkowe są, w przeciwieństwie do bezwarunkowych, zależne od wartości jakiegoś warunku, np. wyniku operacji porównania dwóch liczb. Mikroprocesor musi wtedy obliczyć nie tylko adres docelowy, ale i wartość warunku wykonania skoku.

Wykonanie instrukcji skoku w systemie z potokiem zobrazowane jest na Rys. 2.2, gdzie pierwszą instrukcją jest właśnie skok (oznaczony na zielono).



Rys.2.2. P*rzykładowy przebieg wykonania skoku w mikroprocesorze z potokiem*

Ponieważ instrukcja 1 jest skokiem na adres 10, mikroprocesor powinien wykonać najpierw instrukcję 1, i od razu potem 10. Ponieważ jednak adres docelowy skoku jest znany dopiero po 3 takcie zegara, zmarnowane zostaną 2 takty, w których mogłyby się zacząć wykonywać kolejne instrukcje. Marnowane są całe 2 fazy potoku (oznaczone na czerwono). Ze względu na skok, wszystkie operacje wykonane w tych fazach powinny się w ogóle nie wykonać. Wymagane jest więc albo wstrzymanie ich działania do momentu decyzji o wykonaniu skoku, lub zastosowanie mechanizmu zapewniającego powrót do prawidłowego stanu mikroprocesora. W zaprezentowanym przypadku implementacja potoku nie daje żadnej przewagi nad prostym, bez-potokowym mikroprocesorem.

Dokładnie ten problem ma rozwiązywać moduł predykcji skoków. Powinien on, na podstawie jakiegoś algorytmu, „zgadnąć” jaki będzie adres kolejnej instrukcji, zanim otrzymany zostanie wynik fazy Execute. Dzięki temu możliwe będzie wcześniejsze wczytanie odpowiedniej instrukcji do potoku, nie pozostawiając fragmentu potoku w bezczynności. Należy pamiętać oczywiście, że adres kolejnej instrukcji jest jedynie sugestią, wynikającą najprawdopodobniej z analizy statystycznej poprzednio wykonanych instrukcji. Predykcja skoku nie może być w 100% poprawna. Gdyby była, to musiałaby działać tak samo jak cały fragment mikroprocesora odpowiedzialny za fazę Execute, nie dając tym samym żadnej przewagi w czasie wykonywania programu. Instrukcja skoku musi zostać wykonana do końca, aby móc porównać wynik fazy Execute z poprzednimi wartościami zwróconymi przez predyktor. Jeśli predyktor się pomylił, niezbędne jest

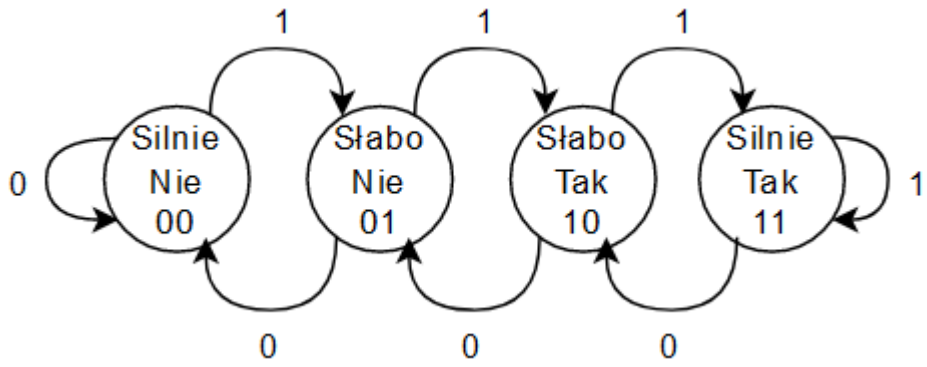
przywrócenie mikroprocesora do stanu sprzed rozpoczęcia wykonywania instrukcji z przewidzianego adresu. Ta cecha nazywana jest wykonywaniem spekulatywnym [5].

2.2. Standardowe techniki predykcji skoków

Algorytmy predykcji skoków dzielą się na 2 rodzaje: statyczne i dynamiczne.

Predyktory statyczne są niezależne od danych przetwarzanych przez mikroprocesor. Konfiguracja predyktora jest niezmienna, bez bezpośredniej ingerencji w niego. Najprostszym przykładem takiego statycznego predyktora, jest np. zawsze przewidywanie, że skok nie zostanie wykonany. Tego typu predyktory mają sens jedynie wtedy, jeśli posiadamy pewne uprzednie założenia co do struktury programów, jakie będą wykonywane na mikroprocesorze. Przykładowo, częstą techniką statycznego przewidywania skoków, jest przewidywanie wszystkich skoków do przodu (na adres większy niż aktualny) jako niewykonane, a skoków do tyłu (na adres mniejszy niż aktualny) jako wykonane. Jeśli programista uważa, że dany skok częściej się wykona, powinien go zaimplementować jako skok do tyłu, a jeśli się nie wykona, to jako skok do przodu. Można też potencjalnie zaprojektować zbiór instrukcji mikroprocesora, w których wybrany bit determinuje wynik predykcji, co daje jeszcze większą swobodę programiście.

Predykcja dynamiczna jest zależna od przetwarzanych danych. Jeśli warunek wykonania skoku zależy od danych znajdujących się w pamięci, których rozkład nie jest znany programiście, predyktory statyczne w niczym nie pomogą. Należy zaprojektować algorytm, który będzie na bieżąco analizował wykonywane instrukcje, i na ich podstawie determinował wyniki kolejnych skoków.

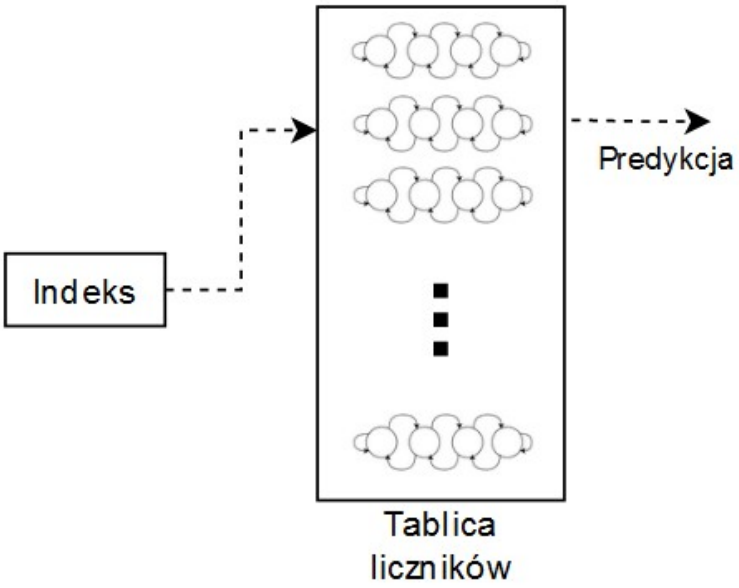


Rys.2.3. *Maszyna stanu 2-bitowego licznika nasycenia*

W wielu różnych algorytmach predykcji dynamicznej wykorzystywane są tzw. liczniki nasycenia, którego przykład zaprezentowano na rysunku Rys 2.3 [6]. Jest to zwyczajny 2-bitowy licznik, który potrafi liczyć w górę oraz w dół, ale jeśli miałby się przepełnić, to jego wartość się nie zmienia. Wraz z każdym wykonanym skokiem, wartość licznika się

zwiększa, a wraz z niewykonaniem skoku warunkowego, zmniejsza. Jeśli wartość najbardziej znaczącego bitu licznika wynosi 1, to kolejny skok przewidziany zostanie jako wykonany, a jeśli 0, jako niewykonany. Pozostałe bity licznika reprezentują nasycenie, czyli ile razy jeszcze licznik musi dokonać błędnej predykcji, aby ją zmienił.

Najprostszym dynamicznym predyktorem jest zastosowanie jednego, globalnego licznika nasycenia. Wraz z każdym napotkanym skokiem jego wartość jest aktualizowana. Takie rozwiązanie miałoby sens w małych programach, gdzie jest niewiele skoków, ale w przypadku bardziej rozbudowanych, składających się z wielu niezależnych komponentów, których skoki są całkowicie od siebie niezależne, nie będzie dawał najlepszych rezultatów. Należy tak zaprojektować predyktor, aby niezależne fragmenty programu nie interferowały ze sobą wewnątrz predyktora. Stosuje się więc najczęściej tablicę liczników z saturacją, nazywaną najczęściej BHT (Branch History Table) [6].

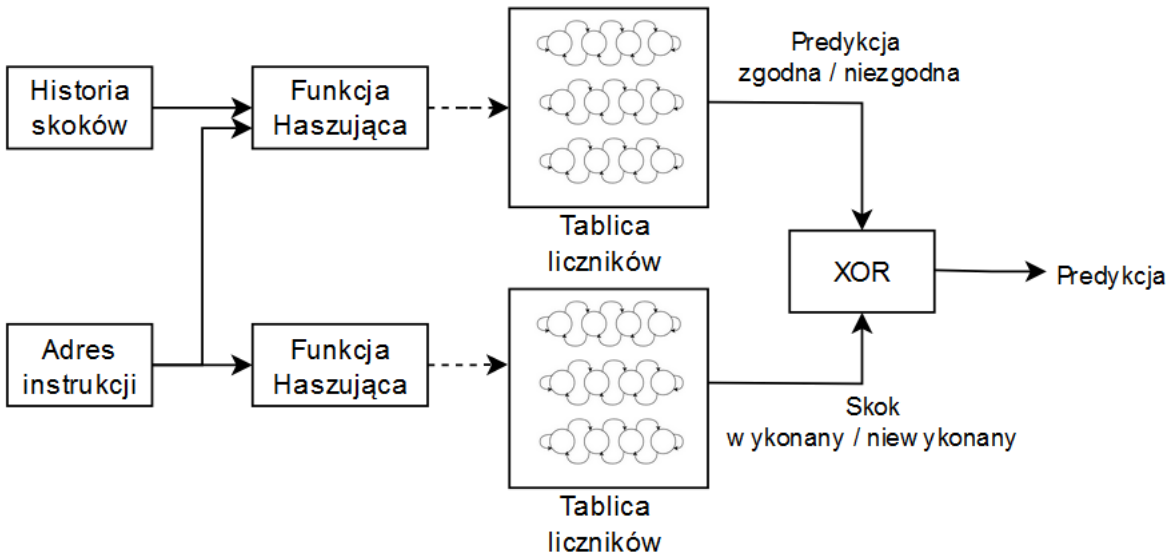


Rys.2.4. *Tablica liczników nasycenia*

W predyktorze implementowanych jest wiele liczników. To, który licznik powinien zostać zaktualizowany / użyty do predykcji, zależy od danego indeksu, jak to zostało zobrazowane na Rys. 2.4. Indeks powinien reprezentować różne instrukcje skoku lub przynajmniej podzbiory istniejących instrukcji skoku, tak, żeby niezależne od siebie ścieżki wykonywania programu, a co za tym idzie niezależne od siebie skoki nie wpływały na siebie. Idealną sytuacją byłoby, gdyby istniał dedykowany licznik na każdy możliwy skok w programie. Niestety, jest to raczej nieosiągalne. Jeśliby założyć, że adres instrukcji ma 64 bity i każda z tych instrukcji może być potencjalnie skokiem, predyktor musiałby posiadać 2^{64} liczników, do których czas dostępu musiałby wynosić co najwyżej małą wielokrotność okresu taktowania sygnału zegarowego mikroprocesora, co jest absurdalnym wymogiem projektowym. Dlatego należy zastosować bardziej wyszukany

sposób na generację indeksu, na wzór funkcji *haszującej* (ang. hash function – rodzaj funkcji, dla której mała zmiana w argumencie powoduje duże i trudno przewidywalne zmiany w wyniku) aktualny PC (Program Counter), tak aby zminimalizować szansę na interferencję niezależnych skoków. PC to rejestr przechowujący adres aktualnej instrukcji. Najczęściej stosuje się po prostu określoną liczbę bitów PC, licząc od najmniej znaczącego. Często wykorzystywany jest też jakiś dynamiczny parametr, np. historia wykonania określonej liczby poprzednio napotkanych instrukcji skoków warunkowych. Wtedy jako indeks wykorzystywana jest nie tylko pozycja w kodzie programu, ale także w pewnym sensie droga, określająca w jaki sposób program w danym miejscu się znalazł. Algorytm wykorzystujący takie podejście nazywany jest Gshare [7].

Powyższe techniki można w dalszy sposób łączyć i komplikować. Przykładem bardziej zaawansowanej techniki jest tak zwany Predyktor Zgodności (ang. Agree Predictor), przedstawiony na Rys. 2.5. [6]



Rys.2.5. *Schemat Predyktora Zgodności*

Predyktor zgodności wykorzystywany był między innymi w mikroprocesorze Intel Pentium 4. Stosowane są tutaj 2 tablice liczników, gdzie jedna z nich na podstawie adresu instrukcji przewiduje, czy skok zostanie wykonany, a druga na podstawie zarówno adresu, jak i historii poprzednich skoków przewiduje, czy ten poprzedni licznik da błędny wynik. Na podstawie tej informacji generowana jest ostateczna predykcja. Taka konfiguracja pozwala na poprawną predykcję zarówno skoków, których wynik jest ściśle związany z historią, jak i takich gdzie nie jest.

Wraz z rozwojem dziedziny sztucznej inteligencji i uczenia maszynowego, zaczęto wykorzystywać sieci neuronowe jako predyktory skoków. Nawet najbardziej skomplikowane algorytmy bazujące na wcześniej opisanych technikach nie są w stanie wykryć bardzo skomplikowanych wzorców, aby poprawnie przewidzieć skoki. Sieci

neuronowe pozwalają na automatyczną detekcję bardzo skomplikowanych wzorców, nawet silnie zaszumionych, jeśli odpowiednio dużo czasu poświęcone zostanie na ich nauczenie. Prowadzone są aktualnie badania nad głębokimi sieciami neuronowymi, lecz faktyczne komercyjne dzisiejsze procesory wykorzystują co najwyżej pojedyncze perceptrony [8], zamiast liczników nasycających. Dają one dużo większe możliwości predykcyjne, będąc skalowalnymi liniowo względem długości historii skoków, lecz algorytm perceptronu jest dużo bardziej wymagający od zwykłego licznika, co czyni tę technikę niekoniecznie pożądaną.

Dotychczas zaprezentowane zostały techniki przewidywania, czy skok wystąpi czy nie. Jest to jednak jedynie połowa sukcesu, ponieważ jeśli skok ma wystąpić, to wymagana jest też wiedza, na jaki adres skok zostanie wykonany. W tym celu wykorzystywany jest BTB (Branch Target Buffer) [6]. Jest to pamięć *cache*, która służy do powiązania adresu instrukcji skoku do potencjalnego adresu docelowego. Najczęściej tym adresem docelowym jest adres, na jaki skok został wykonany przy poprzednim wykonaniu instrukcji. Jeśli instrukcja skoku jest względna, możliwe jest przechowywanie więcej niż jednego ostatniego adresu docelowego, z dodaniem mechanizmu predykcji, który z możliwych adresów najprawdopodobniej zostanie użyty. To właśnie BTB zużywa najwięcej zasobów spośród wszystkich komponentów predyktora, ponieważ musi przechowywać cały adres, dla jak największej liczby napotkanych skoków. Najczęściej tablice liczników nasycenia implementuje się właśnie wewnątrz BTB, ponieważ obie te informacje są tak samo ważne dla poprawnej predykcji skoku. Możliwe jest jednak pominięcie użycia BTB w niektórych przypadkach. Predykcję można wykonać na tyle późno w potoku, aby wiadomy był już potencjalny adres docelowy, podczas gdy jeszcze nie jest wiadomy fakt, czy skok zostanie w ogóle wykonany. Oszczędność czasowa wykonywania programu z takim predyktorem jest jednak oczywiście gorsza, niż jakby predykcja wykonywana była jak najszybciej. Predyktor byłby też wtedy bezużyteczny w przypadku skoków bezwarunkowych, gdzie moment poznania adresu docelowego jest jednoznaczny z możnością wykonania skoku, i nie ma czego przewidywać.

Ostatnim aspektem pozostałym do omówienia jest wykonywanie spekulatywne. Niestety, trudno poruszyć ten temat bez konkretnego projektu mikroprocesora, ponieważ technika ta jest bardzo silnie zależna od konkretnej implementacji. Każdy mikroprocesor może posiadać całkowicie inaczej skonstruowany potok, inaczej obsługiwać efekty uboczne operacji. Może wspierać przerwania lub inne specjalne instrukcje, które muszą współgrać z mechanizmem przywracania stanu mikroprocesora. W nowoczesnych mikroprocesorach może to być operacja bardzo kosztowna, wymagająca wielu dodatkowych cykli zegara, niż w przypadku, gdyby żadna predykcja nie została wykonana. W prostych mikroprocesorach z prostym potokiem operacja ta może być wręcz „darmowa”, nieodróżnialna od po prostu wstrzymania działania mikroprocesora do

momentu aż wyznaczony zostanie wynik instrukcji skoku, co jest sytuacją identyczną do niezastosowania żadnego predyktora.

2.3. Specyfikacja RISC-V

Algorytmy predykcji skoków silnie zależą od zbioru instrukcji i możliwości omawianych mikroprocesorów. Ponieważ w pracy magisterskiej implementacja wykonana została na mikroprocesorze zgodnym ze standardem RISC-V, warto ten standard dokładnie przybliżyć, by zrozumieć jak traktuje on instrukcje skoku.

RISC-V to ISA (Instruction Set Architecture), czyli zbiór instrukcji, jaki mikroprocesor musi implementować [1]. Jest to otwarty i darmowy standard, przystosowany do zastosowań zarówno przemysłowych, jak i badawczo-edukacyjnych. ISA zaprojektowana została w sposób pozwalający na łatwą syntezę zarówno na FPGA (Field Programable Grid Array), jak i Full-Custom ASIC (Application-Specific Integrated Circuit), bez narzucania żadnych większych wymagań dotyczących struktury mikroprocesora. Zdefiniowane jest również wiele oficjalnych rozszerzeń, np. Rozszerzenie M, definiujące instrukcje operacji mnożenia i dzielenia liczb całkowitych, albo Rozszerzenie C, definiujące wsparcie dla skompresowanych 16-bitowych instrukcji (w przeciwieństwie do standardowego rozmiaru 32 bitów). Pozostawia to dużą dowolność projektantowi w implementacji poszczególnych modułów mikroprocesora, jak i jego dodatkowych funkcjonalności, przy jednoczesnej pełnej kompatybilności wspieranych programów między sobą. Ponieważ w praktyce każdy mikroprocesor RISC-V jest inny, implementacje predyktorów skoków są równie różnorodne.

Wspólnym elementem RISC-V jest zbiór instrukcji. W Tab. 2.1 przedstawiona została lista wszystkich instrukcji związanych ze skokami, a co za tym idzie z predykcją skoków.

Tab.2.1. *Format instrukcji skoków RISC-V*

31 20	19 15	14 12	11 7	6 0	Nazwa
imm			rd	1101111	**JAL**
imm	rs1	000	rd	1100111	**JALR**
imm rs2	rs1	000	imm	1100011	**BEQ**
imm rs2	rs1	001	imm	1100011	**BNE**
imm rs2	rs1	100	imm	1100011	**BLT**
imm rs2	rs1	101	imm	1100011	**BGE**
imm rs2	rs1	110	imm	1100011	**BLTU**
imm rs2	rs1	111	imm	1100011	**BGEU**

Oznaczenie *imm* oznacza wartość *immediate* (natychmiastową), czyli wartość operandu bezpośrednio zawartą w instrukcji. Pola *rs1* i *rs2* oznaczają numery rejestrów źródłowych instrukcji, a *rd* numer rejestru docelowego. RISC-V definiuje 32 rejestry ogólnego przeznaczenia, przy czym rejestr x0 (o numerze 0) posiada stałą, nienadpisywanlną wartość zero. Wszystkie instrukcje są 32-bitowe.

RISC-V rozróżnia 2 główne rodzaje skoków – bezwarunkowe (JAL i JALR) oraz warunkowe (BEQ, BNE, BLT, BGE, BLTU, BGEU). Skoki bezwarunkowe są wykonywane zawsze, a warunkowe jedynie w przypadku spełnienia danego warunku.

JAL (Jump And Link) wykonuje skok na adres sumy aktualnego PC i wartości *immediate*. Jeśli *rd* jest równe 0, to jest to wszystko co dana instrukcja robi. Jeśli nie, to do rejestru docelowego wpisywany jest adres aktualnego PC powiększony o 4 (rozmiar jednej instrukcji w bajtach).

JALR (Jump And Link Register) działa identycznie do JAL, z różnicą taką, że zamiast PC, do wyznaczania adresu docelowego skoku wykorzystywany jest wybrany rejestr *rs1*. Ponieważ ani adres instrukcji, ani wartości *immediate* instrukcji nie są w stanie się zmienić podczas działania programu, instrukcja JAL nie może zmienić adresu docelowego skoku w trakcie wykonywania programu. Są to więc skoki pośrednie bezwzględne. Instrukcja JALR już takiej cechy nie posiada, ponieważ wartości przechowywane w rejestrach mogą być dowolne. Jest to skok pośredni względny.

Pozostałe instrukcje to skoki warunkowe. Wartości w rejestrach *rs1* i *rs2* są porównywane, i na podstawie wyniku podejmowana jest decyzja o wykonaniu skoku. Przykładowo, BEQ wykonuje skok jeśli wartości są równe, BLT jeśli wartość w pierwszym rejestrze jest mniejsza od drugiej, itd. Instrukcje z przyrostkiem U oznaczają porównanie liczb bez znaku (unsigned). Domyślnie wykonywane porównania są ze znakiem (signed). Adres docelowy obliczany jest tak samo, jak w instrukcji JAL, czyli jako suma PC i *immediate*. Są to więc zawsze skoki bezwzględne. Litera B w nazwie tych instrukcji oznacza Branch (po polsku gałąź), czyli inne określenie na skok warunkowy. Ponieważ wszystkie te instrukcje mają prawie identyczne działanie, w dalszej części pracy stosowana będzie nazwa BR na określenie ich wszystkich.

Trzeba jeszcze wspomnieć o tak zwanej Standardowej Konwencji Wywoływania (Standard Calling Convention) dla RISC-V [1]. Domyślnie wszystkie rejestry mikroprocesora są ogólnego przeznaczenia i mogą być stosowane zamiennie (oczywiście za wyjątkiem rejestru zerowego). Istnieje jednak ustandaryzowane, preferowane przeznaczenie każdego z rejestrów. Programy są najczęściej podzielone na mniejsze fragmenty, nazywane funkcjami lub podprogramami. Konwencja wywoływania określa ogólne założenia, jakie powinni respektować programiści, podczas wywoływania oraz powracania z podprogramów. Przykładowo, rejestr x2 powinien przechowywać wskaźnik

początku stosu, rejestry od x10 do x17 powinny przechowywać argumenty do wywołań funkcji, itd. Najbardziej interesujące w przypadku predyktora skoków są rejestry x1 oraz x5.

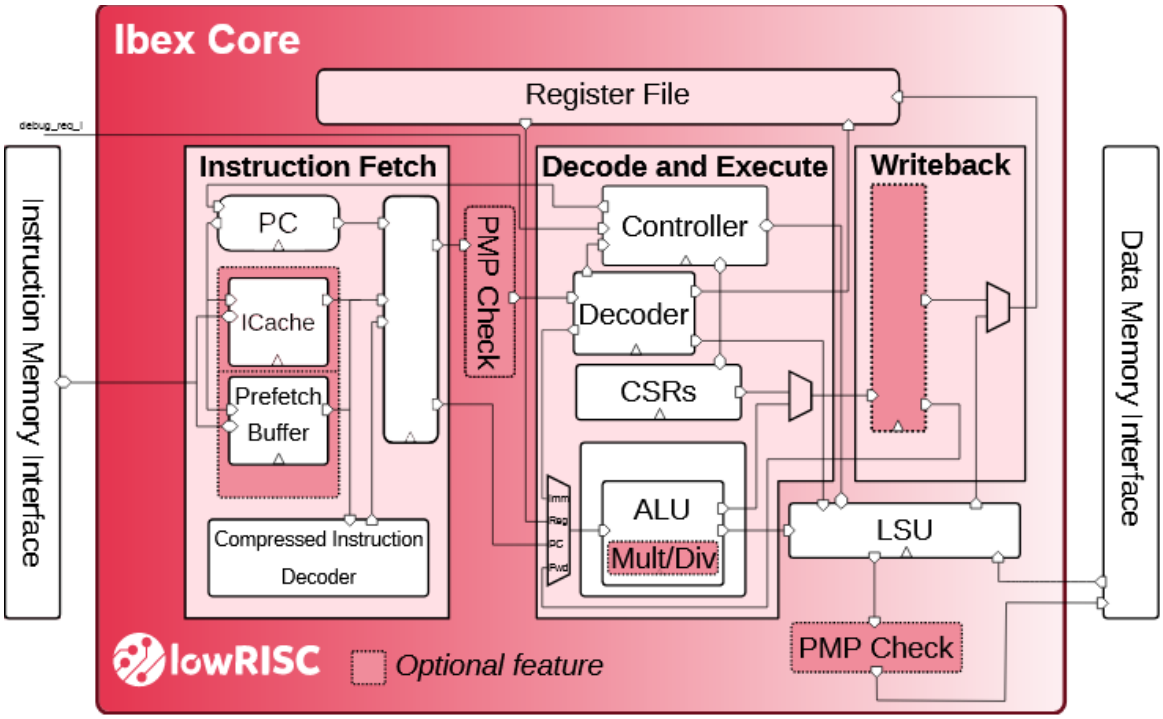
Większość popularnych architektur mikroprocesorów implementuje dedykowane instrukcje CALL (wywołanie funkcji) oraz RET (powrót do miejsca, z którego ostatnia funkcja została wywołana). RISC-V nie definiuje takich instrukcji. Zamiast tego, wykorzystywane są JAL oraz JALR. Rożnica CALL i RET w stosunku do zwykłych skoków to automatyczne zarządzanie adresem powrotnym. Według standardowej konwencji RISC-V rejestr x1 powinien być wykorzystywany jako domyślny rejestr przechowujący adres powrotu, a rejestr x5 jako alternatywny. Można wtedy traktować instrukcje JAL z rejestrem docelowym równym x1 lub x5 jako CALL, oraz JALR z rejestrem źródłowym równym x1 lub x5 jako RET. Istnienie dwóch rejestrów adresów powrotnych wynika ze wsparcia tak zwanych „coroutines”, czyli współprogramów. Wykonanie instrukcji JALR z rejestrem źródłowym x1 i x5 jako docelowym, albo na odwrót, powinno być interpretowane jako jednoczesny CALL i RET, wspierając proste wykonywanie dwóch dowolnych współprogramów niezależnie od siebie, z możliwością przekazywania kontroli między sobą w dowolnym momencie.

2.4. Analiza przykładowych mikroprocesorów RISC-V

W celu dokładniejszej analizy stosowanych w praktyce predyktorów skoków, wybrano 3 projekty mikroprocesorów typu RISC, opartych o specyfikację RISC-V, o różnym poziomie skomplikowania.

2.4.1. IBEX

IBEX [9] to parametryzowalny mikroprocesor 32-bitowy spełniający specyfikację RISC-V. Projekt mikroprocesora stworzony został w języku SystemVerilog. Kod dostępny jest pod licencją Apache License 2.0. Diagram reprezentujący wewnętrzną strukturę mikroprocesora przedstawiony został na Rys. 2.6.



Rys.2.6. *Diagram mikroprocesora RISC* [9]

Mikroprocesor posiada jedynie 2-fazowy potok. Wyróżnia się fazę IF (Instruction Fetch) oraz ID/EX (Instruction Decode and EXecute). Możliwe jest zastosowanie konfiguracji dodającej dodatkową fazę do potoku – Writeback.

IF odpowiada za wczytywanie kolejnych instrukcji z pamięci. Instrukcje są stale wczytywane z pamięci programu z kolejnych adresów do bufora „Prefetch", będącego kolejką FIFO (First In First Out), aż się zapełni. Mikroprocesor wspiera rozszerzenie specyfikacji RISC-V o nazwie „C". Oznacza to, że wspiera skompresowane warianty instrukcji, zajmujące jedynie 16 bitów (zamiast standardowych 32). IF dekoduje wczytane instrukcje skompresowane, o ile takie wystąpiły, a następnie wczytane instrukcje przesyła do kolejnej fazy potoku, w liczbie maksymalnie 1 instrukcji na cykl zegara. W przypadku wystąpienia skoku lub wyjątku, bufor instrukcji jest zerowany.

Faza ID/EX stanowi prawie całość funkcjonalności mikroprocesora. Dekoduje instrukcje otrzymane z poprzedniej fazy na sygnały kontrolne, wczytuje wartości operandów z rejestrów, przekierowuje argumenty do odpowiedniego modułu odpowiedzialnego za dane rodzaje instrukcji itd. Mikroprocesor posiada wsparcie do wszystkich standardowych dla RISC-V rejestrów kontrolnych i statusowych (CSR – Control Status Register). Za ich pomocą definiowane jest zachowanie systemu w razie wystąpienia wyjątku, przerwań systemowych, działanie modułu ochrony pamięci (PMP – phisical memory protection) i tym podobnych. Wiele tych cech nie jest niezbędnych do poprawnego działania mikroprocesora, lecz jest krytycznych dla przeciętnych zastosowań przez potencjalnych konsumentów, zapewniając bezpieczeństwo. Faza ta pozwala na wykonywanie instrukcji wielocyklowych (jak np. mnożenie). Zaimplementowana jest

maszyna stanów, powiązana z dekodowaniem sygnałów kontrolnych, która określa kiedy instrukcja zakończy się wykonywać i będzie można wczytać kolejną z fazy IF.

W trakcie analizy mikroprocesora okazało się, że dokumentacja jest w wielu aspektach dość uboga. Przykładowo, w przypadku opcjonalnej fazy Writeback opisane jest jedynie to, że ma duży wpływ na wydajność. Jaki jest to wpływ nie zostało sprecyzowane.

Wiedząc już jak wygląda potok w mikroprocesorze, można przejść do analizy predyktora skoków. Predyktor jest opcjonalny, należy go włączyć podczas konfiguracji mikroprocesora. Niestety w tym przypadku dokumentacja również jest uboga, więc warto zajrzeć do kodu źródłowego projektu mikroprocesora (Rys. 2.7).

```
    // Determine if the instruction is a branch or a jump

  // Uncompressed branch/jump
  assign instr_b = opcode_e'(instr[6:0]) == OPCODE_BRANCH;
  assign instr_j = opcode_e'(instr[6:0]) == OPCODE_JAL;

  // Compressed branch/jump
  assign instr_cb = (instr[1:0] == 2'b01) & ((instr[15:13] == 3'b110) |
(instr[15:13] == 3'b111));
  assign instr_cj = (instr[1:0] == 2'b01) & ((instr[15:13] == 3'b101) |
(instr[15:13] == 3'b001));

  // Select out the branch offset for target calculation based upon the
instruction type
  always_comb begin
    branch_imm = imm_b_type;

    unique case (1'b1)
      instr_j  : branch_imm = imm_j_type;
      instr_b  : branch_imm = imm_b_type;
      instr_cj : branch_imm = imm_cj_type;
      instr_cb : branch_imm = imm_cb_type;
      default : ;
    endcase
  end

  // Determine branch prediction, taken if offset is negative
  assign instr_b_taken = (instr_b & imm_b_type[31]) | (instr_cb &
imm_cb_type[31]);

  // Always predict jumps taken otherwise take prediction from `instr_b_taken`
  assign predict_branch_taken_o = fetch_valid_i & (instr_j | instr_cj |
instr_b_taken);
  // Calculate target
  assign predict_branch_pc_o    = fetch_pc_i + branch_imm;
```

Rys.2.7. *Kod źródłowy predyktora skoków w mikroprocesorze IBEX* [9]

Predykcja ma miejsce podczas fazy IF. Instrukcja wczytana z pamięci programu jest dekompresowana i częściowo dekodowana, aby określić czy instrukcja ta jest skokiem czy nie. Dekodowany jest także fragment instrukcji określający wartość natychmiastową (*immediate*). Sygnał *predict_branch_taken_o* określa, czy skok przewidziany zostanie jako wykonany, czy nie, a sygnał *predict_branch_pc_o* określa adres docelowy skoku.

Predyktor IBEX reaguje jedynie na skoki bezwzględne. Skoki bezwarunkowe zawsze przewidywane są jako wykonane, a warunkowe jedynie wtedy, kiedy ich wartość

immediate jest ujemna. Ponieważ nieobsługiwane są skoki względne, adres docelowy zawsze wyznaczany jest jako suma aktualnego adresu instrukcji i *immediate*. Jest to zatem predyktor w pełni statyczny. Za każdym razem kiedy instrukcja zostanie wczytana z pamięci programu, jeśli okazała się ona skokiem bezwzględnym, to adres kolejnej instrukcji do wczytania jest wyznaczany właśnie przez predyktor. Ponieważ predykcja odbywa się już po wstępnym dekodowaniu instrukcji oraz cała informacja o potencjalnym adresie docelowym jest w niej zawarta, niepotrzebny jest BTB. Mikroprocesor IBEX jest ciekawym przypadkiem, ponieważ posiada bardzo krótki potok. W większości zaawansowanych mikroprocesorów potok rozbity jest na dużo więcej faz, przez co niemożliwym jest tak wczesne dekodowanie instrukcji. Taka sytuacja występuje między innymi w mikroprocesorze rozwijanym w niniejszej pracy magisterskiej.

Jeśli predyktor uznał, że skok nie będzie wykonany, to jest to sytuacja identyczna do takiej, w której predyktor jest wyłączony. Jeśli z fazy ID/EX otrzymany został sygnał o wykonaniu skoku, to faza ID/EX wstrzymywana jest do momentu, aż IF przekaże kolejną instrukcję. Trwa to zawsze co najmniej 1 cykl zegara, potencjalnie dłużej jeśli adres docelowy nie znajduje się w buforze Prefetch ani w cache'u instrukcji. Jeśli predyktor jest włączony i przewiduje, że skok będzie wykonany, wewnątrz fazy ID/EX informacja ta porównywana jest z rzeczywistym wykonaniem skoku. Jeśli predykcja była poprawna, żaden dodatkowy skok nie jest inicjowany i mikroprocesor pracuje nieprzerwanie. Jeśli jednak była niepoprawna, wymuszany w fazie IF jest dodatkowy skok, który powraca na adres sprzed dokonania predykcji. Wywołuje to analogiczne opóźnienie pracy mikroprocesora co zwykły skok przy wyłączonym predyktorze. Zestawienie potencjalnie utraconych cykli zegara przedstawiono na Tab. 2.2.

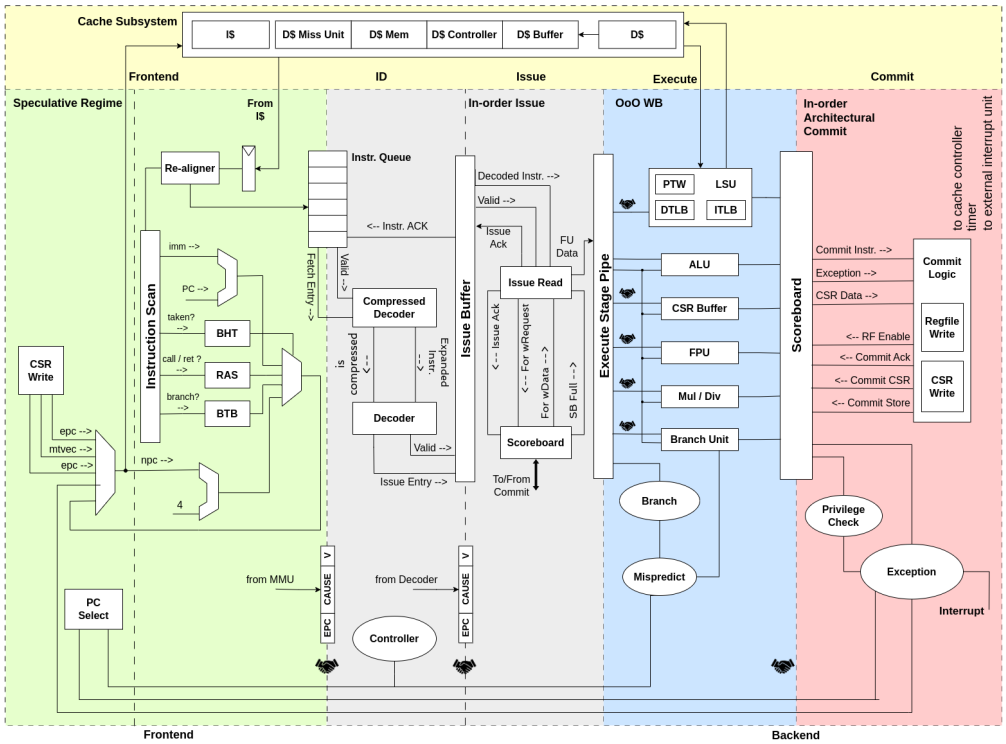
Tab.2.2. *Zależność liczby bezczynnych cykli zegara zależnie od predykcji i skoku*

	Predykcja Poprawna	Predykcja Niepoprawna	Brak Predyktora
Skok wykonany	0	≥ 1	≥ 1
Skok niewykonany	0	≥ 1	0

Jeśli więc wykonywany program posiada wiele skoków warunkowych na adres wcześniejszy, które rzadko są wykonywane (jest to jedyny rodzaj skoku, jaki może być błędnie przewidziany jako wykonany), to bardziej efektywnym byłoby w ogóle niestosowanie predyktora. Jest to jednak mało prawdopodobna sytuacja, a w każdym innym przypadku zastosowanie predyktora jest nie gorsze niż jego niestosowanie. System nie musi zapewniać specjalnego mechanizmu wykonywania spekulatywnego i cofania efektów ubocznych niepoprawnie przewidzianych instrukcji, ponieważ jedyna faza która przetwarza instrukcje spekulatywnie to IF. Faza IF nie posiada żadnych efektów ubocznych.

2.4.2. CV32A65X

CV32A65X [10] to mikroprocesor 32-bitowy, należący do rodziny CORE-V CVA6, będącej wysoce sparametryzowanym wzorcem procesorów, spełniających specyfikację RISC-V. Projekt mikroprocesora stworzony został w języku SystemVerilog. Kod dostępny jest pod zmodyfikowaną licencją Apache License 2.0. Diagram reprezentujący wewnętrzną strukturę mikroprocesora przedstawiony został na Rys. 2.8.



Rys.2.8. *Schemat mikroprocesora CVA6* [10]

Jeśli chodzi o wsparcie rozszerzeń standardu RISC-V, to mikroprocesor CVA6 jest podobny do wcześniej omawianego IBEX. Wspiera jednak bardziej rozbudowany standard Privileged Architecture (Architektury Uprzywilejowanej), który pozwala na pełne wsparcie nowoczesnego systemu operacyjnego opartego na Unix'ie, a więc dowolnej dystrybucji Linuxa.

Najważniejsza różnica, względem IBEX, od razu rzucająca się w oczy, to znacznie dłuższy potok. Posiada on 6 faz. Pierwsze 2 fazy zgrupowane są pod nazwą Frontend.

W fazie PCGen generowany jest kolejny adres kolejnej instrukcji programu. Na wartość tą wpływają między innymi BTB oraz BHT. Faza ta ma więc duże powiązanie z wykonywaniem spekulatywnym oraz predykcją skoków.

Faza Fetch odpowiada za wczytanie kolejnej instrukcji do modułu cache. Wczytane instrukcje przekazywane są do kolejnej fazy potoku, maksymalnie jedna na cykl zegara, poprzez kolejkę FIFO.

Faza Decode przetwarza instrukcję otrzymaną z Fetch na sygnały kontrolne. Jeśli instrukcja była skompresowana, to ją również dekompresuje.

Faza Issue otrzymuje zdekodowane instrukcje z fazy Decode, a także polecenia z kolejnych faz – Execute oraz Commit. Wewnątrz tej fazy następuje odczyt i zapis rejestrów mikroprocesora. Wszystkie otrzymane polecenia są rozprowadzane do odpowiednich modułów funkcyjnych, wraz z wymaganymi wartościami, odczytanymi z rejestrów.

W fazie Execute dokonywane są faktyczne obliczenia. Odpowiednie podmoduły otrzymują polecenia z fazy Issue. Wewnątrz tej fazy przeprowadzana jest również interakcja z interfejsem pamięci.

Faza Commit odpowiada za wszystkie skutki uboczne instrukcji wykonywanych przez mikroprocesor. Inicjuje zapisy do rejestrów, zatwierdza wykonanie zapisów do pamięci oraz przeprowadza obsługę wszelkich wyjątków, jakie wystąpiły w którejkolwiek z poprzednich faz. Zarządza ewentualnym zatrzymywaniem pracy i czyszczeniem wszystkich komponentów mikroprocesora.

Przechodząc do analizy predykcji skoków – proces ten odbywa się wewnątrz faz we Frontend'zie. Instrukcja wczytana w fazie Fetch jest częściowo dekodowana, podobnie jak miało to miejsce w poprzednio omawianym mikroprocesorze IBEX. W przeciwieństwie do IBEX, mikroprocesor CV32A65X posiada BTB, więc wspiera skoki względne. Każdy skok JAL i BR jest automatycznie przewidywany na odpowiedni adres, na podstawie wartości *immediate* instrukcji.

Jeśli napotkana zostanie instrukcja JALR i BTB zwróci odpowiadający tej instrukcji adres docelowy, wykonywana jest odpowiednia predykcja. Jeśli JALR jest skokiem względem rejestru x1 lub x5, a więc jest to RET, w predykcji wykorzystywany jest RAS (Return Address Stack). RAS to stos LIFO (Last In First Out). Wraz z każdym wczytanym CALL'em, do RAS wkładany jest adres powrotny funkcji, i z każdym wczytanym RET'em jest ściągany i uznawany za przewidziany adres docelowy. W przeciwnym wypadku predykcja nie jest wykonywana i PC jest inkrementowany.

Jeśli napotkaną instrukcją jest BR, to konsultowany jest BHT, indeksowany 10 najmniej znaczącymi bitami adresu przewidywanej instrukcji. Domyślnie każdy rekord wewnątrz BHT jest pusty. Przy pierwszym wykonanym skoku, odnoszącym się do danego rekordu, licznik inicjalizowany jest na 2. Jeśli rekord jest pusty, predykcja dokonywana jest zależnie od znaku wartości *immediate*, dokładnie tak jak miało to miejsce w IBEX (skoki do tyłu wykonane, do przodu niewykonane).

Niestety, nie jest w żaden sposób udokumentowany mechanizm przywracania mikroprocesora do poprawnego stanu podczas wykrycia błędnej predykcji. Większość

skutków ubocznych wymaga, aby dotarła w potoku do fazy Commit, a wykrycie błędnej predykcji występuje fazę wcześniej – w Execute. Wystarczy więc w takim wypadku wyzerować wszystkie operacje zakolejkowane w fazach Fetch, Decode oraz Issue (pomijając te zainicjowane przez Commit). Jednak trzeba zauważyć, że BTB, BHT oraz RAS aktualizowane są bezpośrednio wewnątrz Frontend, zanim dojdą do fazy Commit. Mechanizm nadpisywania stanu BTB oraz BHT przedstawiony jest na Rys. 2.9.

```
  assign bht_update.valid = resolved_branch_i.valid
                          & (resolved_branch_i.cf_type == ariane_pkg::Branch);
  assign bht_update.pc = resolved_branch_i.pc;
  assign bht_update.taken = resolved_branch_i.is_taken;
  // only update mispredicted branches e.g. no returns from the RAS
  assign btb_update.valid = resolved_branch_i.valid
                            & resolved_branch_i.is_mispredict
                            &(resolved_branch_i.cf_type == ariane_pkg::JumpR);
  assign btb_update.pc = resolved_branch_i.pc;
  assign btb_update.target_address = resolved_branch_i.target_address;
```

Rys.2.9. *Kod mikroprocesora CV32A65X aktualizujący BTB i BHT* [10]

Jak widać na Rys. 2.9., wszystkie sygnały odpowiadające za aktualizowanie BTB i BHT są zależne od *resolved_branch*. Jest to sygnał pochodzący z modułu Execute, będący faktycznym wynikiem wykonania skoku, nie predykcją. Oznacza to, że BTB i BHT w fazie Frontend są jedynie odczytywane. Aktualizacja następuje jedynie podczas faktycznych skoków, nie podczas wykonywania spekulatywnego.

Inaczej jest jednak z RAS, jak widać na Rys. 2.10.

```
      unique case ({
        is_branch[i], is_return[i], is_jump[i], is_jalr[i]
      })
// ..
        4'b0100: begin
          ras_pop = ras_predict.valid & instr_queue_consumed[i];
          ras_push = 1'b0;
          predict_address = ras_predict.ra;
          cf_type[i] = ariane_pkg::Return;
        end
// …
      if (is_call[i]) begin
        ras_push   = instr_queue_consumed[i];
        ras_update = addr[i] + (rvc_call[i] ? 2 : 4);
      end
```

Rys.2.10. *Kod mikroprocesora CV32A65X aktualizujący RAS* [10]

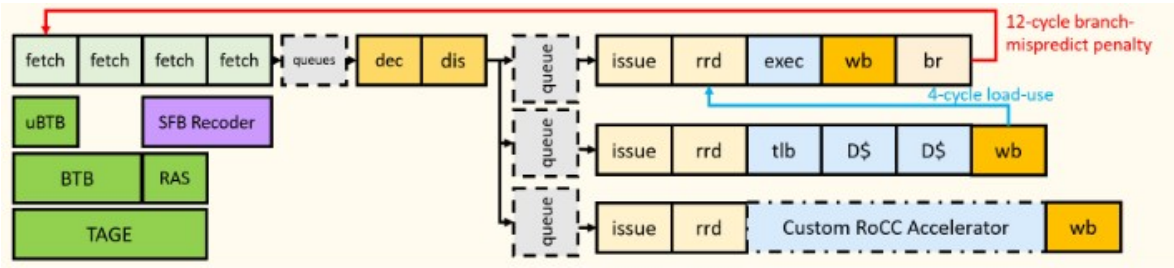
Sygnały *ras_pop*, *ras_push* oraz *ras_update* odpowiadają za aktualizację RAS. Jak widać, sygnały te są ustawiane, jeśli *is_return[i]* lub *is_call[i]* są włączone. Sygnały te pochodzą z zdekodowanej, właśnie wczytanej instrukcji, nie z instrukcji aktualnie wykonywanej w Execute. Warto zauważyć, że zaraz obok zarządzania RAS, dokonywana w kodzie jest sama predykcja (sygnał *predict_address*), co nie miałoby sensu, gdyby było inaczej. W każdym razie, oznacza to, że RAS jest aktualizowany podczas wykonywania spekulatywnego. Nie ma mechanizmu przywracania poprzedniego stanu RAS. Jeśli więc

mikroprocesor wykona choć jedną instrukcję JAL lub JALR, niezgodną z konwencją RISC-V, czyli wykorzystującą rejestry x1 lub x5 w roli innej niż adresu powrotu, albo stos RAS się przepełni, RAS potencjalnie nigdy już nie przewidzi poprawnie żadnej instrukcji, do momentu restartu całego mikroprocesora. Jest to według autora pracy poważna wada mikroprocesora, której projektanci albo nie przewidzieli, albo nie udokumentowali w żaden sposób.

Niestety, koszt nieprawidłowej predykcji skoków również nie jest nigdzie udokumentowany, a analiza jego na podstawie kodu źródłowego jest zbyt złożona aby ją przeprowadzić.

2.4.3. SonicBOOM

SonicBOOM (Berkeley Out-of-Order Machine) [11] to mikroprocesor stworzony na Uniwersytecie Kalifornijskim w Berkeley, podobnie jak sama specyfikacja RISC-V. W przeciwieństwie do poprzednio omawianych mikroprocesorów, stworzony został w języku Chisel, pozwalającym na szeroką parametryzację projektu i łatwą integrację na wszelkich systemach układów scalonych. Projekt mikroprocesora jest na otwartej licencji BSD 3-Clause. Wyróżnia się tym, że SonicBOOM jest najszybszym publicznie dostępnym otwartym jednordzeniowym mikroprocesorem, mierząc pod względem IPC (Instructions Per Cycle), czyli instrukcji na cykl zegara. Na Rys. 2.11. przedstawiono schemat potoku mikroprocesora.



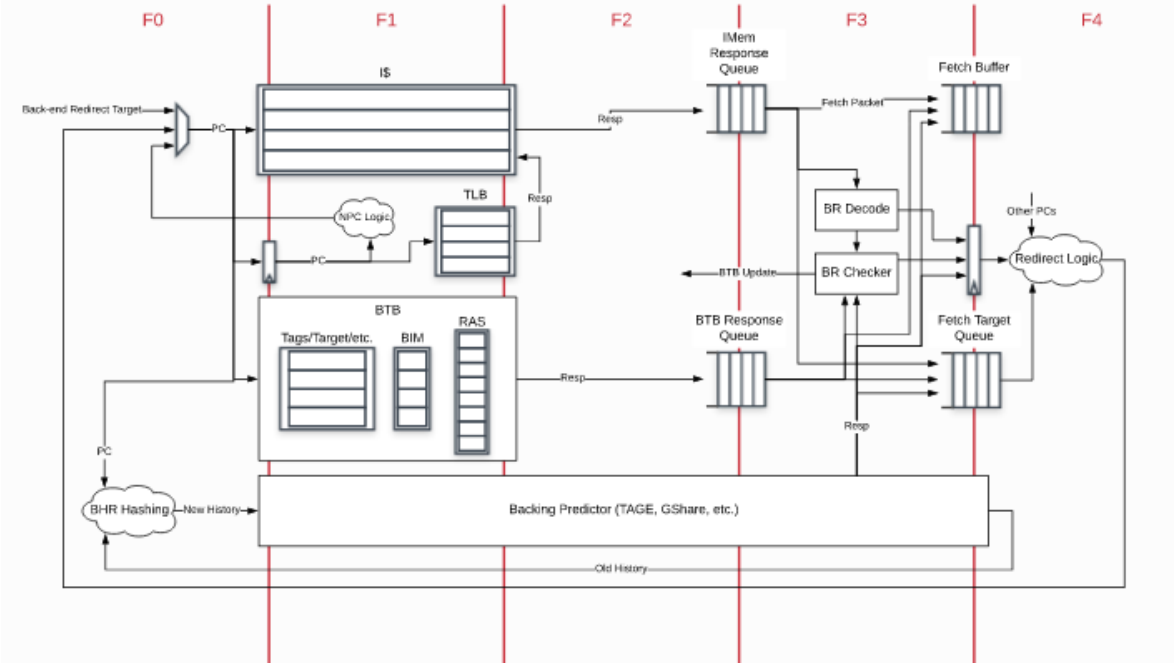
Rys.2.11. *Schemat potoku mikroprocesora SonicBOOM* [11]

W mikroprocesorze wyróżnić można aż 10 faz potoku: Fetch, Decode, Register Rename, Dispatch, Issue, Register Read, Execute, Memory, Writeback and Commit. Wiele z nich pełni role analogiczne do podobnie nazwanych faz w poprzednio omawianych mikroprocesorach, więc nie zostaną one obszernie omówione.

Cechą wyróżniającą SonicBOOM od poprzednich mikroprocesorów jest możliwość wykonywania instrukcji poza kolejnością. Moduł Decode nie tylko zamienia instrukcje na sygnały kontrolne, ale kompiluje je na mikroinstrukcje, które następnie zostają w module Dispatch rozdzielone do odpowiednich kolejek, do odpowiednich rozgałęzień potoku. Wszystkie gałęzie są od siebie w większości niezależne, co pozwala na wykonywanie kolejnych instrukcji, zanim poprzednia się zakończy na innej gałęzi. Do takiego działania

wymagane jest zastosowanie abstrakcji na poziomie rejestrów. Ten sam numer rejestru nie zawsze odnosi się do tego fizycznego rejestru. Za rozdzielanie rejestrów mikroinstrukcjom odpowiada moduł Register Rename.

Inną ważną cechą jest to, że faza Fetch na wykonanie potrzebuje aż 4 cykli zegara. Fetch podzielony jest na 5 podfaz (F0 do F4), z czego faza F4 jest wykonywana jednocześnie z F0. Długość tej fazy potoku wynika między innymi z zastosowania zaawansowanego algorytmu predykcji skoków. Faza Fetch dokładniej przedstawiona jest na Rys. 2.12.



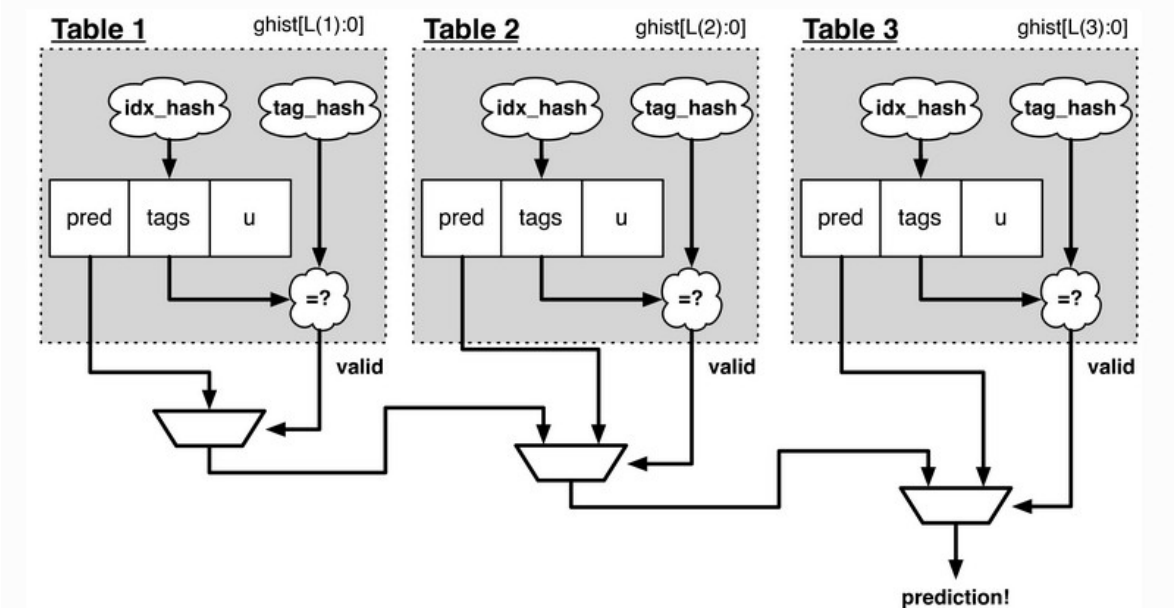
Rys.2.12. Diagram f*azy Fetch mikroprocesora SonicBOOM* [12]

SonicBOOM posiada tak naprawdę 2 predyktory – szybszy nazwany NLP (Next Line Predictor) oraz wolniejszy, ale dokładniejszy BPD (Backing Predictor).

NLP dokonuje predykcji w czasie 1 cyklu zegara. W jego skład wchodzą BTB, RAS oraz BHT złożona z 2-bitowych liczników nasycenia, tutaj nazywana BIM (Bi-Modal Table). Predykcja w NLP dokonywana jest wyłącznie na podstawie aktualnego PC, żadna wczytywana instrukcja nie jest w żaden sposób dekodowana. Rekordy wewnątrz BTB składają się z wartości reprezentującej adres docelowy skoku oraz flag, określających, czy skok będzie warunkowy i czy będzie to CALL lub RET. Adresy docelowe przechowywane są w osobnej pamięci w postaci skompresowanej (najczęściej wykonywane są krótkie skoki, czyli PC i adres docelowy różnią się tylko na mniej znaczących bitach, co pozwala na zastosowanie kompresji). Jeśli ostatnio wczytana instrukcja dla danego PC to był RET, adres docelowy wczytywany jest z RAS zamiast z owej pamięci. Cały predyktor NLP mieści się w podfazie F1. Jeśli NLP nie posiada rekordu odpowiadającego aktualnej instrukcji, predykcja nie jest dokonywana. W pełni poprawnie przewidujący NLP pozwala

na nieprzerwane działanie całego potoku mikroprocesora. Jest to więc mechanizm bardzo podobny do całego modułu predykcji skoków obecnego w CV32A65X.

Przewidywanie skoków warunkowych na podstawie 2-bitowych liczników nasycenia jest rozwiązaniem prostym i popularnym, ale nie pozwala predyktorowi na wykrycie bardziej skomplikowanych wzorców. Ponieważ SonicBOOM zaprojektowany został z myślą o maksymalnej możliwej wydajności, zaimplementowano predyktor BPD działający równolegle z NLP. BPD nie przewiduje w żaden sposób adresu docelowego skoku. Wartość ta pobrana musi zostać z NLP, lub w podfazy F3, gdzie dokonywane jest częściowe dekodowanie instrukcji, co pozwala na odczyt wartości *immediate* i wyznaczenie adresu docelowego skoku. BPD podejmuje wyłącznie decyzje o wykonaniu bądź niewykonaniu skoku warunkowego. Możliwe jest do wyboru kilka konfiguracji BPD, z czego preferowaną i najbardziej zaawansowaną jest algorytm TAGE (TAgged GEometric) [13], którego schemat działania przedstawia Rys. 2.13.



Rys.2.13. *Schemat działania predyktora TAGE* [12]

Na predyktor składa się kolekcja tablic, przechowujących liczniki nasycaenia predykcji, liczniki użyteczności, oraz tagi. Przechowywana jest też informacja o historii wykonanych skoków warunkowych (ciąg zer i jedynek reprezentujący wykonanie poprzednich skoków). W celu wykonania predykcji, aktualny PC oraz historia skoków jest przetwarzana przez funkcję *hashującą*, otrzymując hash indeksu oraz hash tagu. Każda kolejna tablica TAGE wykorzystuje coraz dłuższy fragment historii skoków. Jeśli rekord odpowiadający wygenerowanemu hashowi indeksu posiada wpis o tagu równym hashowi tagu, to odczytywany jest przypisany mu licznik nasycenia jako predykcja. Taka sama operacja wykonywana jest na każdej tabeli. Jeśli więcej niż jedna tabela dokona predykcji, to priorytet bierze tablica o dłuższej historii. Pozostaje jeszcze kwestia aktualizacji stanu tablic. Wraz z każdą poprawnie dokonaną predykcją inkrementowany jest odpowiadający

jej licznik użyteczności, a dekrementowany z każdą niepoprawną predykcją. Jeśli tablica nie posiada wpisu z tagiem odpowiadającym wyliczonemu, to tworzony jest nowy wpis na miejscu innego wpisu, którego licznik użyteczności wynosi 0. Jeśli żaden licznik w rekordzie nie wynosi 0, nowy wpis nie jest tworzony, ale wszystkie liczniki użyteczności w rekordzie zostają dekrementowane. Mechanizm ten zapewnia, aby TAGE dokładniej dokonywał predykcji dla częściej występujących skoków, nie pozwalając rzadziej wykorzystywanym skokom na zaśmiecenie tablic. Ponieważ algorytm wykorzystuje wiele tablic o różnej długości historii, pozwala na jednoczesne uczenie się wzorców o bardzo skomplikowanych i długich historiach, i szybkim wykrywaniu krótkich wzorców.

Wykonywanie spekulatywne i cofanie niechcianych efektów ubocznych również jest rozwiązane w bardzo przemyślany sposób. Większość struktur w predyktorze skoków (RAS, TAGE) są aktualizowane bezpośrednio wewnątrz fazy Fetch. Jednak wraz z każdą predykcją tworzony jest tzw. „migawka” (ang. snapshot) aktualizowanych rekordów. W przypadku wykrycia błędnej predykcji, moduły te przywracane są do stanu z odpowiedniej migawki, sprzed wykonania błędnej predykcji. Przy wysyłaniu wczytanych instrukcji do kolejnej fazy potoku z fazy Fetch, do instrukcji dopisywany jest tag reprezentujący konkretną predykcję. Tagi te propagują w głąb potoku. W momencie wykrycia błędnej predykcji, na podstawie tego właśnie tagu dokonywana jest decyzja w fazie Commit, czy należy efekty instrukcji anulować.

3. Rozbudowywany mikroprocesor

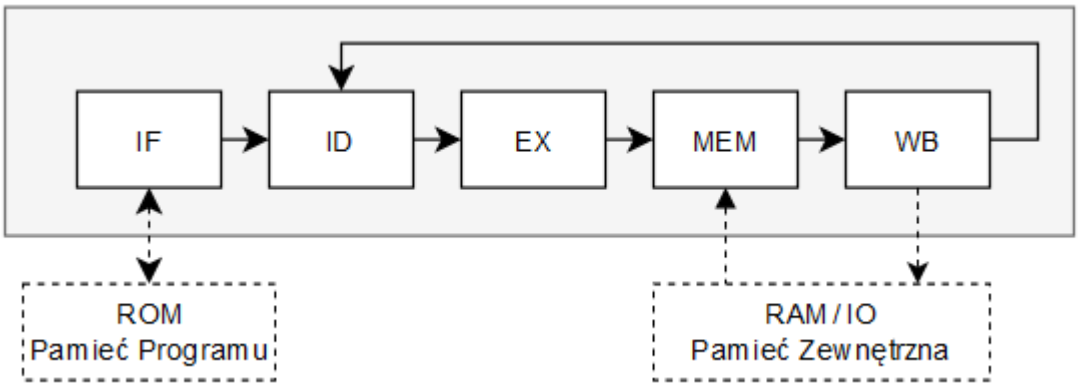
Moduł predyktora stanowi formę rozbudowy istniejącego projektu mikroprocesora. Mikroprocesor ten został zaprojektowany pierwotnie w ramach projektu PBL [2], a następnie rozwinięty w ramach kilku prac magisterskich [3]. Jest to mikroprocesor RISC, implementujący specyfikację RISC-V. W przeciwieństwie do wszystkich wcześniej omawianych mikroprocesorów w rozdziale 2, mikroprocesor ten implementuje konkretnie standard RV64I, który jest 64-bitową wersją podstawowego standardu RV32I. Wszystkie rejestry posiadają szerokość 64 bitów, oraz do zbioru instrukcji wprowadzono kilka dodatkowych, operujących na danych 64-bitowych. Przykładem jest np. instrukcja wczytująca i zapisująca do pamięci operacyjnej słowa 64-bitowe, zamiast 32-bitowych, oraz dwa zestawy instrukcji arytmetycznych, gdzie jedne operują na 64-bitowych danych, a drugie na 32-bitowych z rozszerzeniem bitu znaku. Nie implementuje on żadnych dodatkowych rozszerzeń specyfikacji. Nie ma więc wsparcia dla instrukcji skompresowanych, instrukcji mnożenia itd. Projekt mikroprocesora został wykonany w języku opisu sprzętu SystemVerilog.

Podczas analizy otrzymanego projektu mikroprocesora, odkryto, że niestety posiada on bardzo pewne braki w funkcjonalności, i kilka niewykrytych wcześniej błędów, uniemożliwiających przeprowadzenie miarodajnych badań. Autor pracy magisterskiej zmuszony został więc do stworzenia własnego środowiska testowego, aby odkryć wszystkie niuanse w działaniu mikroprocesora i go naprawić. Odkryto wiele jawnych niezgodności z dostarczoną specyfikacją. Pomimo swojej pozornej prostoty, doprowadzenie mikroprocesora do stanu używalności zajęło znaczną część pracy. Większość wykrytych błędów zostanie omówiona w odpowiednich sekcjach tego rozdziału.

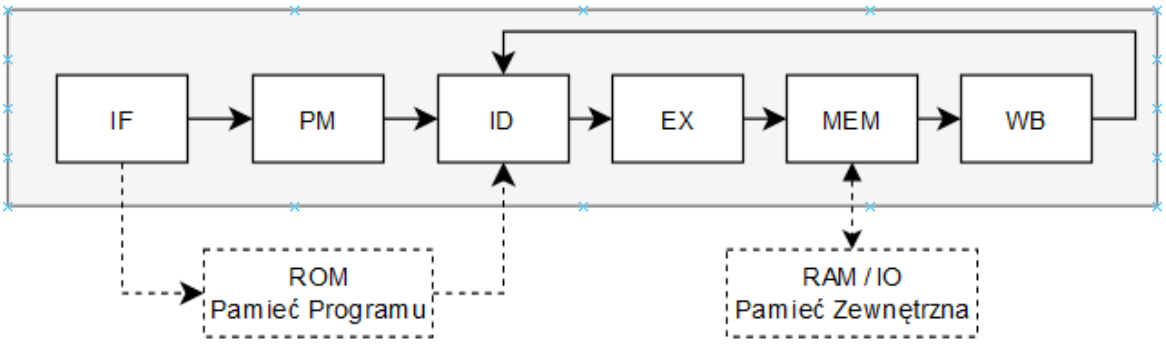
3.1. Potok

Według dostarczonej wraz z mikroprocesorem specyfikacji, potok mikroprocesora ma być podzielony na 5 prostych faz (Instruction Fetch, Instruction Decode, Execute, Memory, Writeback), zgodnych z Rys. 3.1. Zrealizowanie całej instrukcji przez cały potok powinno tym samym zająć maksymalnie 5 cykli sygnału zegarowego (z pominięciem ewentualnego oczekiwania na pamięć). Niestety, nie mogło to być stwierdzenie dalsze od prawdy. W trakcie testów odkryto, że faz jest 6, i niektóre z nich zachowują się niezgodnie

ze specyfikacją. Nową fazę nazwano Program Memory (PM) i umiejscowiono ją pomiędzy Fetch oraz Decode. Bliższy stanowi faktycznemu obraz potoku przedstawiono na Rys. 3.2



Rys.3.1. *Diagram potoku mikroprocesora według dokumentacji*



Rys.3.2. *Diagram faktycznego potoku mikroprocesora*

Podczas fazy IF (Instruction Fetch) kolejna instrukcja powinna zostać wczytana z pamięci programu, podobnie jak miało to miejsce w dowolnym wcześniej omawianym mikroprocesorze. Tak się jednak nie dzieje. W rzeczywistości, jedyną funkcją wykonywaną przez fazę IF jest inkrementacja PC i nadpisywanie go podczas wykonywania instrukcji skoku. Pamięć programu jest zrealizowana jako, osobna od pamięci zewnętrznej, pamięć ROM (Read Only Memory), wbudowana w rdzeń mikroprocesora. Pamięć ROM taktowana jest tym samym zegarem co reszta mikroprocesora, i czas jej odpowiedzi wynosi 1 cykl. Faza IF ustawia jedynie wejście do ROM'u, jako adres z którego należy wczytać instrukcję.

Ponieważ opóźnienie ROM'u wynosi jedynie 1 cykl, wczytana instrukcja mogłaby być dostępna już w kolejnej fazie potoku. Niestety, faza IF wprowadza własne opóźnienie poprzez przerzutnik typu D. Adres wejściowy dociera więc do ROM'u o cykl później, więc nie można jeszcze odczytać kolejnej instrukcji. Z tego powodu wprowadzono do potoku dodatkową fazę, nazwaną przez autora pracy magisterskiej PM, (Program Memory). Jej jedynym zadaniem jest opóźnienie PC o jeden cykl, w oczekiwaniu na zwrócenie odpowiedniej wartości przez ROM.

Dopiero w kolejnej fazie, czyli ID (Instruction Decode) można nareszcie wczytać instrukcję. Instrukcja ta jest dekodowana na sygnały kontrolne. Dodatkowo, wszystkie rejestrowe operandy wejściowe są w tej fazie odczytywane. Zdekodowana instrukcja, wraz z wartościami wszystkich argumentów, przekazywana jest do kolejnej fazy.

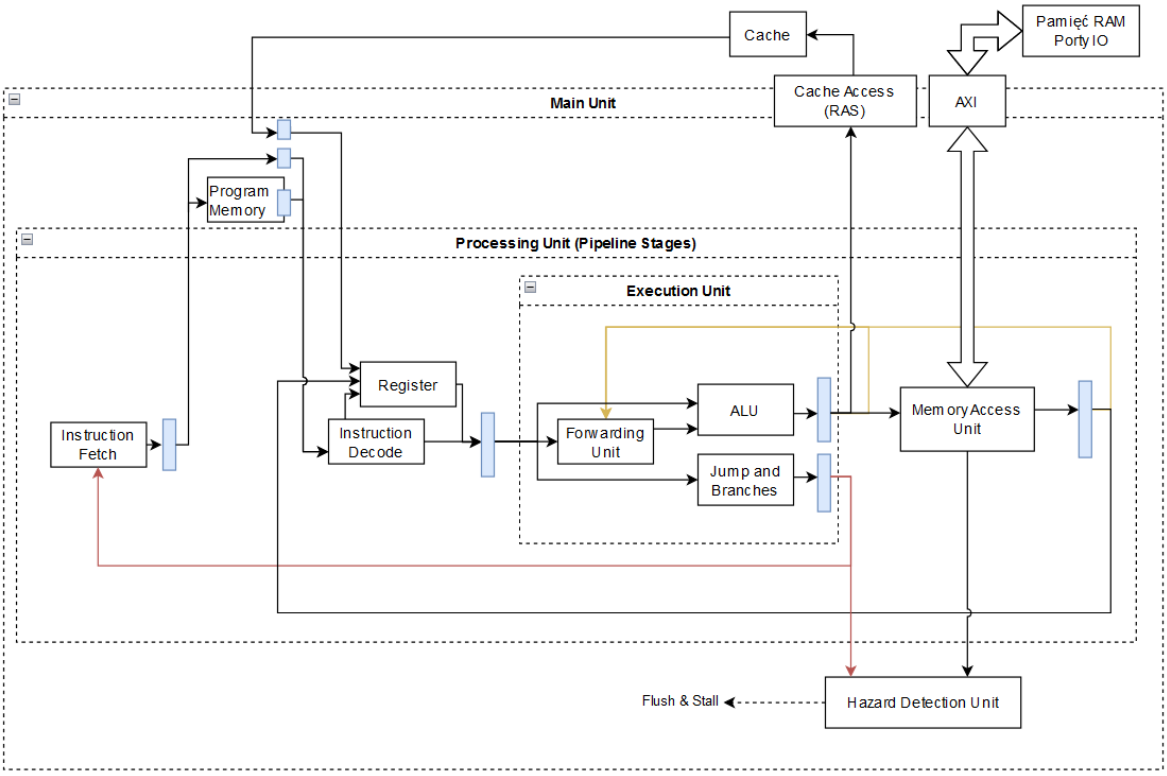
W fazie EX (Execute) wykonywane są faktyczne obliczenia. Jest to też moment, w którym podejmowana jest decyzja o podjęciu skoków. Wynik obliczeń, wraz z ważnymi sygnałami kontrolnymi przekazywany jest do kolejnej fazy, i dodatkowo w przypadku skoku wysyłany jest sygnał do fazy IF o zmianie PC na adres docelowy.

W fazie MEM (Memory) przeprowadzane są operacje na pamięci RAM oraz ewentualnie innych urządzeniach I/O (Input/Output, urządzeniach wejścia/wyjścia). Według dokumentacji faza MEM powinna odpowiadać jedynie za odczytywanie wartości z pamięci, a zapisywanie powinno odbywać się w kolejnej fazie. W rzeczywistości jednak wszystkie operacje mają miejsce w MEM. Jeśli dana instrukcja nie jest operacją na pamięci, faza MEM działa jedynie jako opóźnienie.

W ostatniej fazie, czyli WB (WriteBack) wyniki wszystkich wcześniej przeprowadzonych operacji zapisywane są z powrotem do rejestrów.

3.2. Poszczególne moduły

Aby poprawnie zaimplementować moduł predykcji skoków, potrzebna jest dokładniejsza wiedza o wewnętrznym działaniu mikroprocesora. Na Rys. 3.3 pokazano bardziej rozbudowany schemat mikroprocesora, wyszczególniający relacje pomiędzy konkretnymi zaimplementowanymi modułami, a nie tylko abstrakcyjnymi fazami potoku.



Rys.3.3. *Schemat budowy wewnętrznej mikroprocesora*

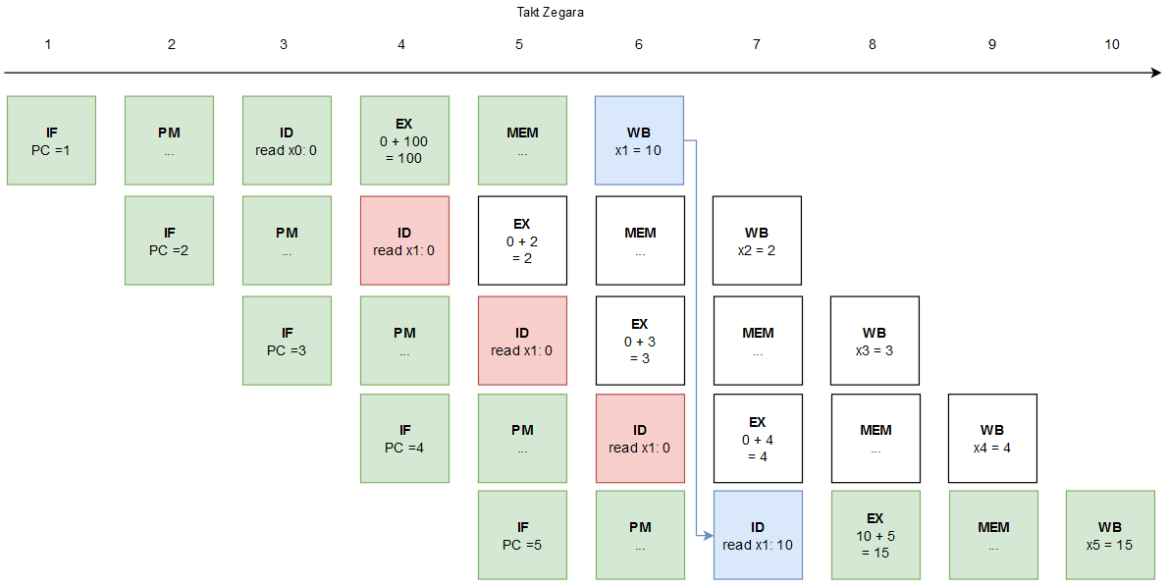
Nazwy wszystkich modułów i podmodułów na diagramie są zgodne z ich nazwami w kodzie źródłowym [14]. Poprzez niebieskie pionowe prostokąty oznaczono przerzutniki, a więc oznaczają punkty rozdzielenia faz potoku. Wszystko w obrębie „Main Unit" należy do wnętrza rdzenia mikroprocesora, a pozostałe elementy (czyli RAM i Cache) są zewnętrznymi modułami, z którymi komunikacja odbywa się poprzez odpowiedni interfejs. Komunikacja z RAM'em odbywa się poprzez interfejs AXI.

3.2.1. Forwarding Unit

Istotnym aspektem, o którym nie wspomniano wcześniej przy omawianiu mikroprocesorów, jest reagowanie na wyniki instrukcji, które jeszcze nie zostały w pełni zrealizowane przez potok. Na Rys. 3.4 przedstawiono przykładowy program w asemblerze dla RISC-V, a na Rys. 3.5. jego wykonanie w poszczególnych fazach potoku.

```
1: ADDI   x1,   x0,   10    ; x1 = 10
2: ADDI   x2,   x1,   2     ; x2 = x1 + 2
3: ADDI   x3,   x1,   3     ; x3 = x1 + 3
4: ADDI   x4,   x1,   4     ; x4 = x1 + 4
5: ADDI   x5,   x1,   5     ; x5 = x1 + 5
```

Rys.3.4. *Kod źródłowy Programu 1*



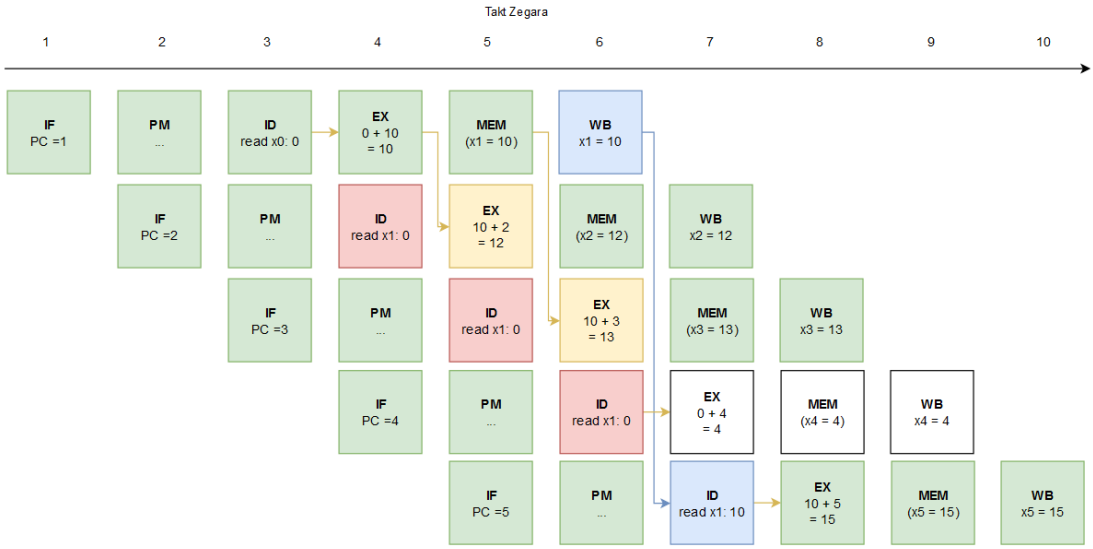
Rys.3.5. *Przebieg programu 1 przez potok*

Program powinien ustawić wartość rejestru x1 na 10, a rejestrów od x2 do x5 na od 12 do 15. Instrukcje od 2 do 5 posiadają jako argument wejściowy rejestr x1. Jednak wartość rejestru x1 nadpisana zostanie dopiero w fazie WB. Instrukcje od 2, 3 i 4 odczytają więc wartość rejestru x1 błędnie jako 0 (lub, co prawdopodobniejsze, jako nieprzewidywalną wartość, zależną od poprzednio wykonywanych instrukcji lub inicjalizacji rejestru przy starcie mikroprocesora). Przeciwdziałać temu problemowi można na 3 sposoby:

- Przy każdej instrukcji wykonywanej w EX, która spowodować ma efekt uboczny, należy wstrzymać pracę mikroprocesora w fazach IF, PM i ID. Spowoduje to, że wartość do zapisania do rejestru docelowego zawsze zdąży spropagować do fazy WB, i zawsze będzie potem poprawnie odczytywana. Jest to jednak rozwiązanie skuteczne, ale kompletnie nieefektywne. Niweluje praktycznie całkowicie ideę istnienia potoku.
- Programista ma pełną władzę nad tym, która instrukcja korzysta z których rejestrów. Może więc w taki sposób stworzyć kod, aby zapewnić odstęp odpowiedniej liczby instrukcji, zależnie od konkretnego mikroprocesora, zanim użyje rejestru docelowego jako źródłowego. Nie zawsze jest jednak to możliwe, co wymusza potencjalne dodanie wielu instrukcji NOP, co znacznie spowolni wykonywanie programu, podobnie jak miałoby to miejsce w pierwszym rozwiązaniu. Dodatkowo, nie byłoby to zgodne ze standardem RISC-V, który zezwala na korzystanie z wyników obliczeń już w kolejnej instrukcji. Mikroprocesor nie byłby kompatybilny więc ze znaczną większością istniejących programów.
- Najsensowniejszym, i zastosowanym, rozwiązaniem jest analizowanie każdej instrukcji poprzez porównanie adresów rejestrów źródłowych instrukcji w fazie EX

oraz adresów rejestrów docelowych instrukcji znajdujących się w kolejnych fazach potoku. Jeśli napotkano zgodny adres, w fazie EX wartość operandów źródłowych powinna być czerpana nie z fazy ID, ale z fazy MEM lub WB. Dokładnie takie zadanie posiada Forwarding Unit.

Sygnały związane z Forwarding Unit na Rys. 3.3. zaznaczono kolorem żółtym. W module porównywane są adresy rejestrów we wszystkich tych fazach potoku, a następnie wartość operandu jest adekwatnie przekierowywana. Przebieg przykładowego programu z uwzględnieniem Forwarding Unit zamieszczono na Rys. 3.6.



Rys.3.6. *Przebieg programu 1 przez potok z uwzględnieniem Forwarding Unit*

Jak widać, dalej nie naprawia to wszystkiego. W otrzymanym projekcie mikroprocesora błąd ten nie został naprawiony. Należało więc naprawić go własnoręcznie. W tym celu, zmieniono działanie modułu Register. Domyślnie, jeśli w tym samym takcie wykonywany był odczyt (z fazy ID) i zapis (z fazy WB) tego samego rejestru, to odczytywana była stara wartość, sprzed zapisu. Po wprowadzonych zmianach, zapis bierze priorytet nad odczytem, co de facto przyśpiesza działanie modułu WB o jeden takt, niwelując widoczny na Rys. 3.6. błąd.

Nie jest to jednak koniec wymaganych do wprowadzenia korekt. Jak widać na Rys. 3.3, moduł Forwarding Unit wpływa wyłącznie na ALU. Instrukcje skoków obsługiwane są po części w osobnym module – Jump And Branches. Sprawia to, że Forwarding Unit nie wpływa na instrukcje JALR, które to posiadają rejestr źródłowy, który jak najbardziej powinien być w miarę możliwości przekierowywany jak reszta. Należało więc dodać dodatkową logikę, odpowiedzialną za przekierowywanie dla instrukcji JALR i powiązać ją z modułem od skoków. Fragment zmian w kodzie przedstawiono na Rys. 3.7.

```
// Forwarding Unit
assign mux_jalr_src_c = (!i_jump) ?                                          (2'b00) :
   !i_was_read &&
   (i_ex_rd_write && (i_ex_rd_addr != 0) && (i_ex_rd_addr == i_rs1_addr)) ?
                                                                             (2'b01) :
   (i_mem_rd_write && (i_mem_rd_addr != 0) &&
   (i_mem_rd_addr == i_rs1_addr)) ?                                          (2'b10) :
                                                                             (2'b00);

// Jump And Branches
wire [DATA_WIDTH - 1:0] rs1_data_c = i_mux_jalr_src == 2'b01 ? i_alu_data :
                                     i_mux_jalr_src == 2'b10 ? i_wr_data :
                                                               i_rs1_data;
```

Rys.3.7. *Fragment poprawek dotyczących Forwarding Unit i instrukcji JALR*

3.2.2. Hazard Detection Unit

Moduł o nazwie Hazard Detection Unit odpowiada za generowanie sygnałów zerowania (flush) i wstrzymywania (stall) poszczególnych faz potoku mikroprocesora. Źródłami tych sygnałów mogą być dwie rzeczy – pamięć zewnętrzna oraz skoki.

Moduł Memory Access Unit, stanowiący praktycznie całość fazy MEM, komunikuje się z pamięcią poprzez interfejs AXI. Pamięć RAM nie musi być w żaden sposób zsynchronizowana z zegarem mikroprocesora, i czas dostępu może wynieść dowolną liczbę cykli. Niezbędne jest więc wstrzymanie pracy mikroprocesora, dopóki dana transakcja się nie zakończy. Magistrala AXI jest interfejsem rodzaju Valid/Ready. Nadawca danych ustawia sygnał Valid w momencie, kiedy dane są gotowe do wysłania, a odbiorca ustawia sygnał Ready w momencie, kiedy jest gotowy od odebrania danych. Transakcja jest zakończona w momencie, kiedy zarówno sygnały Ready i Valid na danym kanale są jednocześnie aktywne. Hazard Detection Unit decyduje o wstrzymywaniu mikroprocesora bezpośrednio na podstawie tych sygnałów. Jeśli transakcja została zainicjowana przez Memory Access Unit (Read Ready lub Write Valid jest aktywny), i pamięć zewnętrzna nie odpowiedziała jeszcze odpowiednim sygnałem Valid lub Ready, praca całego mikroprocesora jest wstrzymywana.

Takie działanie ma jeden mankament. Jeśli wykonywaną instrukcją był odczyt wartości z pamięci do rejestru, a kolejna instrukcja korzysta z danego rejestru jako wartość źródłowa, to dane nie zostaną poprawnie przekierowane do instrukcji. Ponieważ wartość rejestru docelowego nie była znana w momencie zakończenia fazy EX, mikroprocesor musi odczekać jeden dodatkowy cykl, aż dane spropagują z wewnątrz fazy MEM do fazy WB. Fakt ten również nie był nigdzie udokumentowany i wymagał poprawek w kodzie ze strony autora pracy magisterskiej. Jeśli więc adres rejestrów docelowego instrukcji odczytu z pamięci i źródłowego dowolnej kolejnej instrukcji jest taki sam, moduły należące do faz IF, PM, ID oraz EX są wstrzymywane. Faza MEM oraz WB przez jeden cykl pracują jako jedyne. W ten sposób wartość w rejestrze źródłowym będzie widoczna dla modułu Forwarding Unit. W takiej konfiguracji jednak trzeba zauważyć, że faza MEM zostanie

wykonana 2-krotnie – raz normalnie, i raz ze względu że poprzednia faza została wstrzymana. Należy ją więc w tej sytuacji wyzerować. Takie rozwiązanie niestety wprowadza 1 dodatkowy cykl opóźnienia, lecz trudno jest problem ten rozwiązać inaczej, bez poważnej ingerencji w strukturę mikroprocesora. Fragment kodu wyznaczania powiązanych sygnałów wstrzymywania i zerowania przedstawiono na Rys. 3.8.

```
// Forwarding Unit
assign o_source_from_mem = i_was_read && !flush_c &&
                           (i_ex_rd_write && (i_ex_rd_addr != 0) &&
                           (i_ex_rd_addr == i_rs1_addr ||
                            i_ex_rd_addr == i_rs2_addr));

// Hazards Detection Unit
wire wait_for_load_c = i_source_from_mem && rready_r && !rready_c;
reg  wait_for_load_r;
`RTL_REG_ASYNC (clk, nreset, enable, wait_for_load_c, wait_for_load_r, 1) // DFF

wire stall_all = rready_c || wvalid_c;
assign o_stall_if     = stall_all|| wait_for_load_c;
assign o_stall_id     = stall_all|| wait_for_load_c;
assign o_stall_ex     = stall_all|| wait_for_load_c;
assign o_stall_mem    = stall_all;
assign o_stall_rd_reg = stall_all|| wait_for_load_c;
assign o_stall_wr_reg = stall_all;
assign o_flush_mem = wait_for_load_r && !wait_for_load_c;
```

Rys.3.8. *Fragment kodu odpowiedzialnego za oczekiwanie na pamięć zewnętrzną*

Oprócz oczekiwania na pamięć, Hazard Detection Unit gra rolę również przy skokach. Tutaj system jest dużo prostszy niż przy pamięci. Jeśli faza EX podjęła decyzję o wykonaniu skoku, to oprócz podmiany wartości PC w fazie IF, informacja ta wysyłana jest do Hazard Detection Unit, który zeruje fazy PM, ID oraz EX.

3.2.3. Cache (RAS)

Mikroprocesor posiada moduł komunikacji z pamięcią Cache, czyli pamięcią podręczną. Zazwyczaj pamięć taka wykorzystywana jest jako pośrednik między RAM i mikroprocesorem, w celu skrócenia czasu dostępu do często używanych adresów. W przypadku tego projektu jest jednak inaczej. Moduł, który posiada (według autora pracy magisterskiej) mylną nazwę Cache, może być wykorzystywany jedynie jako stos RAS.

Zastosowanie RAS wprowadza do mikroprocesora funkcjonalność niezgodną ze specyfikacją RISC-V. Jak było wspomniane w rozdziale 2, mikroprocesory RISC-V zamiast instrukcji CALL i RET wykorzystują instrukcje JAL i JALR operujące na rejestrach x1 lub x5. W programach często stosuje się zagnieżdżone wywołania funkcji (czyli, że funkcja wywoływana jest z innej funkcji, która wywoływana jest z innej funkcji, itd.). Zapewnienie, aby w rejestrze x1 zawsze znajdował się poprawny adres powrotny przy wywołaniach instrukcji JALR leży po stronie programisty. Najczęściej dokonuje się tego przeznaczając jeden z rejestrów na przechowywanie wartości o adresie szczytu stosu w pamięci zewnętrznej. Na początku każdej funkcji adres ten jest inkrementowany, i

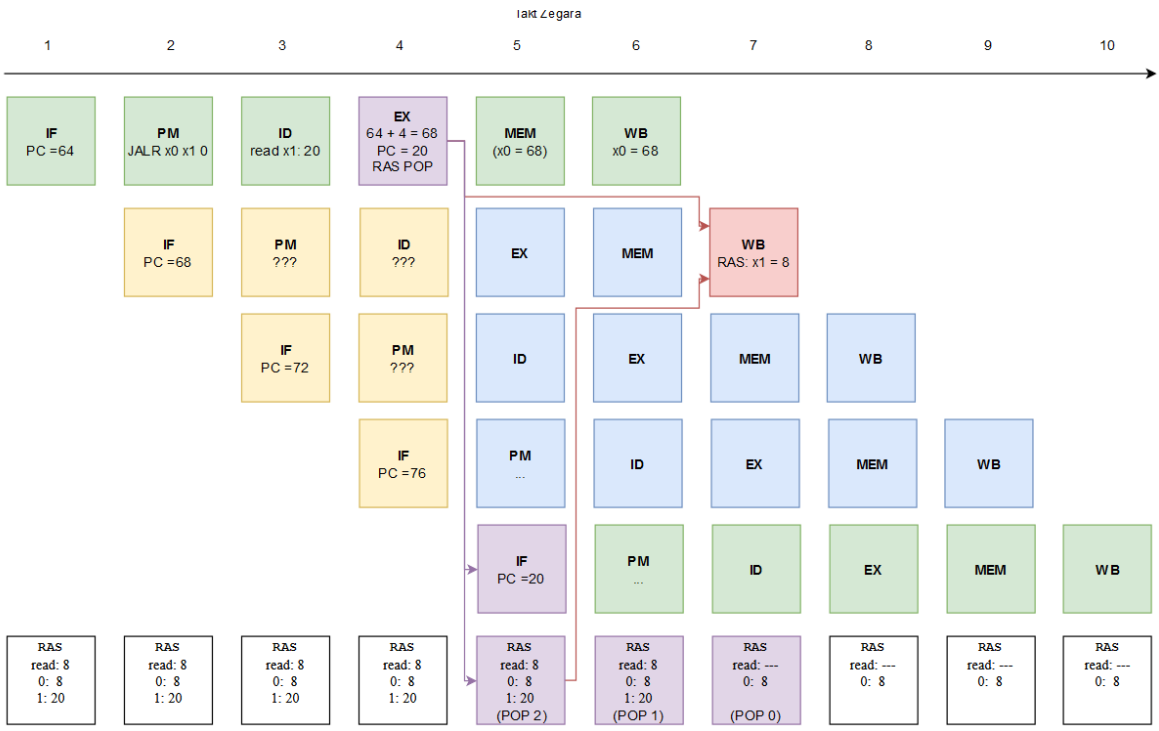
zapisywany jest na niego aktualny adres powrotny. Przed powrotem z wywołanej funkcji (czyli przed instrukcją JALR) adres ze stosu jest wczytywany i wpisany do rejestru x1. Dzięki tej konwencji dowolna funkcja może być w dowolnym momencie wywołana, i adres powrotny zawsze będzie poprawny. Porównanie programów wykorzystującego i niewykorzystującego ową optymalizację przedstawiono na Rys. 3.9. Przyjęto, że wskaźnik na stos przechowywany jest w rejestrze x2, zgodnie ze Standardową Konwencją Wywoływania RISC-V.

```
; Program bez optymalizacji
4:  JAL  x1, +8     ; CALL na adres 12

12: ADDI x2, x2, +4 ; Inkrementracja stosu
16: SD   x2, x1, 0  ; Wpisanie x1 na stos
20: ...
24: JAL  x1, +36    ; CALL na adres 60
28: ...
32: LD   x1, x2, 0  ; Ściągnięcie x1 ze stosu
36: ADDI x2, x2, -4 ; Dekrementacja stosu
40: JALR x0, x1, 0  ; RET

60: ADDI x2, x2, +4 ; Inkrementracja stosu
64: SD   x2, x1, 0  ; Wpisanie x1 na stos
68: ...
72: LD   x1, x2, 0  ; Ściągnięcie x1 ze stosu
76: ADDI x2, x2, -4 ; Dekrementacja stosu
80: JALR x0, x1, 0  ; RET
```

```
; Program z optymalizacją RAS
4:  JAL  x1, +8     ; CALL 12

12: ...
16: JAL  x1, +44    ; CALL 60
20: ...
24: JALR x0, x1, 0  ; RET

60: ...
64: JALR x0, x1, 0  ; RET
```

Rys.3.9. *Porównanie programów bez i z optymalizacją automatycznego RAS*

Wymaga to wszystko dodania do programu dodatkowych instrukcji interakcji z pamięcią. Projektanci mikroprocesora postanowili, że mechanizm ten będzie zarządzany automatycznie, poprzez moduł Cache. Za każdym wywołaniem instrukcji CALL (JAL z rejestrem docelowym x1 lub x5) do stosu RAS w pamięci Cache wkładany jest adres powrotny. Wraz z wywołaniem instrukcji RET (JALR z rejestrem źródłowym x1 lub x5), wartość ze stosu jest ściągana, a **następnie wpisana do rejestru x1**. Jest to optymalizacja działająca niezgodnie ze standardem RISC-V. Instrukcje JALR z rejestrem źródłowym x1 nadpisują rejestr źródłowy. Przykładowe działanie RAS, zobrazowane na potoku, przedstawiono na Rys. 3.10. Przedstawiony został moment wywołania instrukcji JALR pod adresem 64 programu z Rys. 3.9 z optymalizacją RAS.



Rys.3.10. *Przykładowy diagram zachowania RAS przy instrukcji JALR*

Rys. 3.10. możliwie dokładne odzwierciedla zachowania mikroprocesora podczas wykonania instrukcji JALR. W module RAS (czyli w pamięci Cache) przechowywane są 2 wartości – 8 oraz 20. Są to adresy powrotne kolejno wywołanych funkcji (od skoków z adresów 4 oraz 16). W momencie wykonania instrukcji JALR w fazie EX, wykonywany jest skok. Podmieniony zostaje PC w fazie IF na wartość znajdującą się w rejestrze x1, czyli 20. Za pośrednictwem Hazard Detection Unit, fazy EX, ID oraz PM zostaną wyzerowane w kolejnym takcie, co oznaczono kolorem niebieskim. Zerowanie propaguje się wzdłuż potoku. W fazie EX w ALU wyznaczany jest również adres powrotny dla wykonanej instrukcji JALR, jako PC powiększony o 4. Adres ten zapisany zostałby do podanego rejestru docelowego, jednak ponieważ rejestrem docelowym jest x0, nie zostanie nadpisana jego zawsze zerowa wartość. Wszystkie te procesy przebiegłyby niezależnie od zastosowania optymalizacji RAS. Ponieważ jednak ją zastosowano, dzieją się dwie dodatkowe rzeczy.

Po pierwsze, do interfejsu komunikacji z Cache wysyłany jest sygnał o chęci ściągnięcia ze szczytu stosu wartości. Sygnał ten, zanim zacznie działać, jest opóźniony przez kilka warstw przerzutników. Oznaczono to poprzez POP2, POP1 oraz POP0 w module RAS na diagramie. Dopiero podczas POP0 stos się pomniejsza.

Po drugie, poprzez osobną sekwencję przerzutników, opóźniony sygnał o chęci nadpisania rejestru x1 wartością z RAS propaguje od fazy EX w takcie 4 do fazy WB w takcie 7. Warto zauważyć, ze faza ta powinna zostać wyzerowana, ze względu na skok. Nadpisanie wartości rejestru x1 bierze jednak priorytet nad zerowaniem. Wartość, którą

należy nadpisać rejestr, pobierana jest z przedostatniej wartości w RAS. Wartość ta jednak jest również opóźniona poprzez przerzutniki, dlatego źródło odczytu RAS zaznaczono na diagramie dopiero w takcie 5.

Ponieważ ze względu na skok, 3 fazy potoku zostają wyzerowane, kolejny moment, w którym potencjalnie mógłby być odczytany rejestr x1 w celu wykonania kolejnej instrukcji, jest faza ID w takcie 7. Jest to ten sam takt, w którym w fazie WB nadpisywany jest rejestr x1. Wszystko jest więc poprawnie zsynchronizowane. Problem polega jednak na tym, że poprawność działania RAS jest zależna od trzech wyzerowanych faz na potrzebę wykonania skoku. Jeśli zerowanie by nie wystąpiło (np. z powodu dodania do mikroprocesora modułu predykcji skoków), kilka instrukcji mogłoby odczytać starą wartość rejestru x1. Dodatkowo, pierwsza z tych instrukcji mogłaby chcieć nadpisać jakiś rejestr. Operacja ta byłaby wykonana w fazie WB w takcie 7, w tym samym momencie co nadpisanie x1 z RAS. Projekt mikroprocesora nie przewiduje takiej sytuacji, i nadpisanie x1 wzięłoby priorytet. Należało więc zmodyfikować kod źródłowy modułu Register, rozdzielając logikę zależnie od źródła zapisu. Zezwolenie na jednoczesny zapis do dwóch rejestrów komplikuje lekko projekt, lecz jest to jedyne rozwiązanie pozwalające na zachowanie poprawności pracy mikroprocesora po implementacji predyktora skoków.

Trzeba jeszcze wspomnieć o jednym błędzie. RAS reaguje na instrukcje JAL oraz JALR wykorzystujące zarówno rejestr x1, jak i x5. Jednak podczas instrukcji JALR, nadpisywany jest zawsze rejestr x1. Autorzy mikroprocesora najwidoczniej nie byli zgodni co do implementacji wsparcia współprogramów w RAS. Błąd ten nie został naprawiony.

Podsumowując, w przypadku wykonywania standardowego programu na mikroprocesorze, nie przewidującego optymalizacji RAS, dopóki kod programu jest w pełni zgodny ze Standardową Konwencją Wywoływania, oraz nie wykorzystywane są współprogramy, zostanie on w pełni poprawnie wykonany, niezależnie czy w mikroprocesorze zastosowano optymalizację RAS, czy nie. Jeśli jednak nie spełnia któregoś z tych warunków, rejestr x1 może się zachowywać w nieoczekiwany sposób.

3.3. Wybór predyktorów do implementacji

Znając już ogólną budowę wybranego mikroprocesora, po dokonaniu dogłębnego przeglądu literatury oraz przeanalizowaniu istniejących implementacji predyktorów skoków, można dokonać wyboru predyktora do implementacji w pracy magisterskiej.

Ponieważ istnieje wiele różnorodnych rodzajów predyktorów skoków, postanowiono, że zaimplementowane zostaną kilka różnych predyktorów, działających w odmienny

sposób. Pozwoli to w fazie badawczej określić jaki predyktor najlepiej sprawdza się do jakich zadań.

Wykorzystywany mikroprocesor w początkowych fazach potoku nie zna wczytywanej instrukcji. Dopiero w fazie Decode możliwa jest jej analiza. Podobna sytuacja miała miejsce w mikroprocesorze SonicBOOM. Wczytywanie rozpoczynało się w podfazie F0, ale faktyczna instrukcja możliwa do wstępnego dekodowania dostępna była dopiero w podfazie F3. Predykcja dokonywana jest w fazie F0 z użyciem NLP, wyłącznie na podstawie PC. Błąd w np. przewidzianym adresie docelowym skoku, korygowany był dopiero w fazie F3, na podstawie wartości *immediate* wczytanej instrukcji. Z tego powodu, postanowiono stworzyć 2 rodzaje predyktorów. Jeden z nich będzie dokonywał predykcji wyłącznie na podstawie PC, w fazie Fetch, podobnie jak NLP w SonicBOOM. Drugi rodzaj predyktora działać będzie w fazie Decode. Będzie posiadał możliwość korekty jawnie błędnych predykcji. W takim predyktorze możliwe będzie także zaimplementowanie predyktora statycznego, niezależnego w żaden sposób od predyktora z fazy Fetch.

Ponieważ predyktor w fazie Fetch nie ma dostępu do instrukcji, niezbędne jest zaimplementowanie BTB. Postanowiono stworzyć więc BTB wiążący adresy instrukcji z adresami docelowymi, oraz z licznikiem nasycenia. BTB będzie parametryzowalny pod względem liczby możliwych do przechowania rekordów. W literaturze nie spotkano się z licznikami nasycenia o szerokości innej niż 2 bity. Postanowiono więc sparametryzować szerokość liczników w BTB, w celach badawczych.

Innym predyktorem wybranym do implementacji jest taki, gdzie indeksem dostępu byłaby kombinacja PC oraz historii skoków, czyli wykorzystujący BHT. Nie można jednak bazować wyłącznie na BHT, ponieważ służyć to może wyłącznie do przewidywania wykonania skoków warunkowych. Nadal potrzebny jest BTB, aby móc przewidzieć adres docelowy. Predyktor TAGE występujący w SonicBOOM byłby ciekawy w implementacji, jednak jest zbyt skomplikowany na zakres pracy magisterskiej, dlatego pozostano przy rozwiązaniach w NLP. Predyktor składać się będzie więc z BTB oraz BHT. W BTB dodatkowo przechowywana będzie flaga, mówiąca o tym, czy dana instrukcja jest skokiem warunkowym, czy bezwarunkowym. Jeśli przewidywany jest skok warunkowy, o wyniku predykcji decydować będzie odpowiedni rekord w BHT. Jeśli rekord odpowiadający aktualnej historii skoków nie istnieje, predykcja skoku warunkowego nadal może być wykonana na podstawie licznika nasycającego w BTB. Rozmiar BHT oraz przechowywanych w nim liczników również będzie w pełni parametryzowalny.

Postanowiono zaimplementować dodatkowo jeszcze 2 predyktory w ramach testów. Jeden z nich przewidywać będzie w sposób losowy, niezależnie od niczego. Pozwoli to sprawdzić odporność systemu na niespodziewane predykcje. Drugi bazować będzie na predefiniowanych na poziomie syntezy wartościach. Pozwoli to stworzyć predyktor, który

zawsze zachowuje się w ten sam, przewidywalny sposób. Są to predyktory nie mające najmniejszego sensu w zastosowaniu praktycznym, jednak będą bardzo przydatne przy weryfikacji poprawności działania predyktora.

Zarówno dla predyktorów dla fazy Fetch, jak i dla Decode, planowane jest dodanie jeszcze jednej optymalizacji, związanej z instrukcjami RET. W przypadku przewidzenia instrukcji powrotu, adres docelowy może być bezpośrednio zaczerpnięty z rejestru x1, zakładając zgodność ze Standardową Konwencją Wywoływania. Stworzone zostaną więc wersje predyktorów które dodatkowo implementują tę optymalizację.

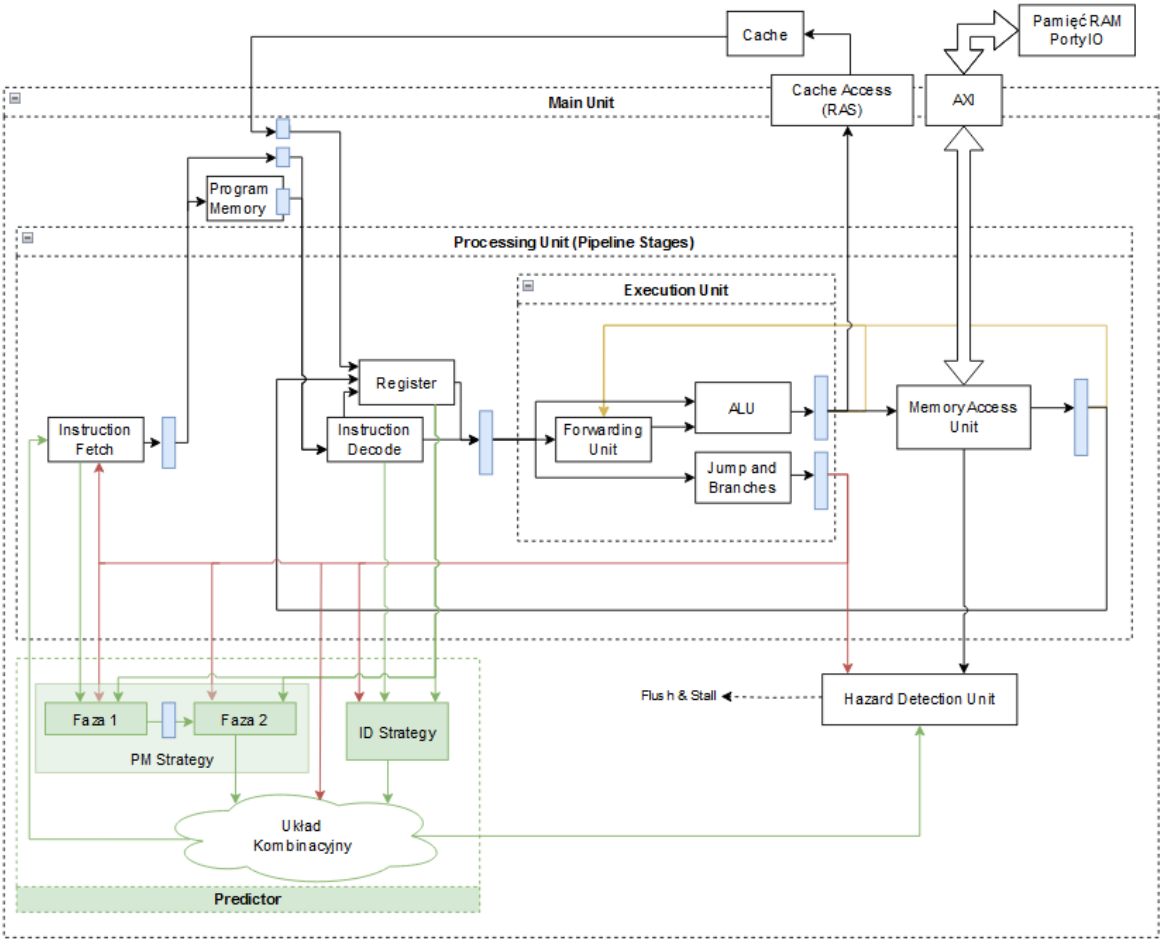
Rozważano również wykorzystanie RAS zaimplementowanego już w mikroprocesorze. Po rozmyśleniach wywnioskowano jednak, że ze względu na duże opóźnienie, zanim RAS zostanie zaktualizowany, wykorzystanie bezpośrednio rejestru x1, bez zastosowania żadnego stosu, da dość dobre wyniki, przy znacznie prostszej implementacji.

Wszystkie predyktory muszą też zapewniać poprawność działania podczas wykonywania spekulatywnego. Ponieważ wszystkie efekty uboczne wykonane zostaną dopiero w fazach MEM lub WB, a decyzja o wykonaniu skoku podejmowana jest wcześniej – w fazie EX, zapewnienie poprawności predykcji musi mieć miejsce nie później niż w fazie EX. Jest to trywialne do wykonania, ponieważ warunek i adres skoku obliczane są właśnie w fazie EX. Wystarczy porównać wykonaną predykcję z wynikiem fazy EX. Wymagane jest jedynie zaimplementowanie mechanizmu odrzucenia wykonania skoków podczas detekcji poprawnej predykcji oraz mechanizmu powrotu do prawidłowego adresu w przypadku błędnej predykcji. Mechanizmy te są niezależne od wybranego algorytmu predyktora skoków.

4. Zaimplementowane predyktory

4.1. Ogólna struktura modułów predyktora

Moduł zaimplementowanego modułu predyktora, wraz z połączeniami z pozostałymi modułami mikroprocesora, przedstawiono na Rys. 4.1 kolorem zielonym.



Rys.4.1. *Schemat mikroprocesora wraz z zaimplementowanym modułem predykcji skoków*

Jak wspomniano w rozdziale 3, postanowiono zaimplementować 2 niezależnie działające algorytmy predyktora – jeden bazujący wyłącznie na PC, a drugi dodatkowo na zdekodowanej instrukcji. Algorytmy te nazwano kolejno PM Strategy oraz ID Strategy. Ich nazwa zależy od fazy potoku mikroprocesora, która aktualnie przetwarza instrukcje, której wynik ma zostać przewidziany. Jest nieintuicyjnym to, że pierwszy algorytm bazuje na fazie Program Memory, a nie Fetch. Jeśli jednak przewidywano by wynik skoku już w fazie Fetch, to w fazie Program Memory wczytana zostałaby już instrukcja z

przewidzianego adresu skoku. Sama instrukcja skoku nigdy nie zostałaby przetworzona przez potok, a więc niemożliwym byłoby wykrycie błędnych predykcji.

Zauważono tutaj też potencjalne ulepszenie procesora. Wszystkie skoki wyznaczane przez fazę Execute, mogłyby bezpośrednio nadpisywać PC, z którego wczytywana jest instrukcja w fazie Program Memory, zamiast PC wewnątrz fazy Fetch. Takie rozwiązanie wprowadzałoby wydłużenie ścieżki krytycznej o jeden 2-wejściowy multiplekser, aby przekierować wejście do Program Memory pomiędzy fazami Fetch i Execute, ale zamiast tego zmniejszałaby liczbę zerowanych faz potoku podczas skoku z 3 do 2.

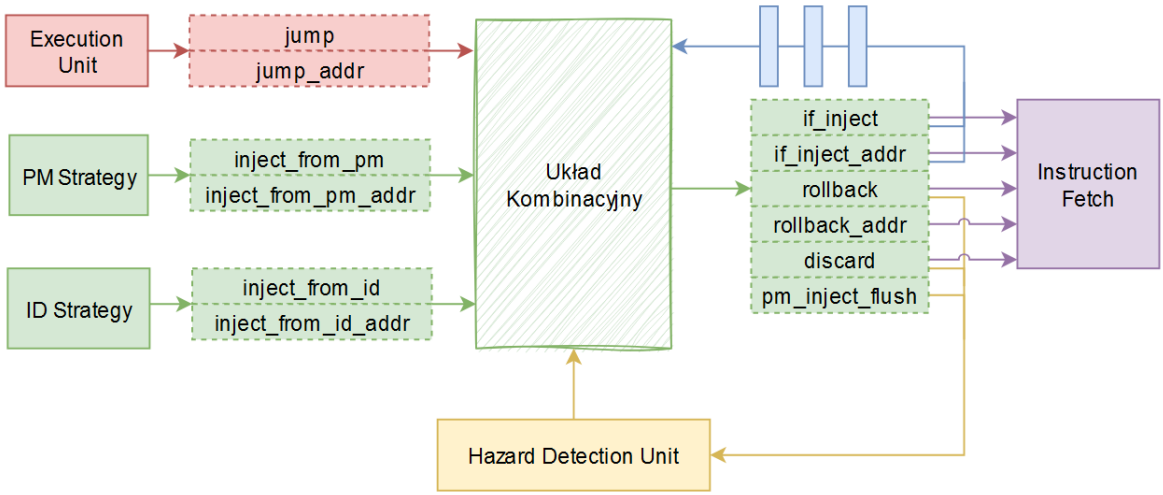
Ponieważ PC wyznaczony w fazie Fetch, zawsze zostanie przekazany do fazy Program Memory, można w bardzo łatwy sposób zaimplementować algorytm PM Strategy jako wielofazowy, z de facto własnym, małym potokiem. W pierwszej fazie PM Strategy na podstawie PC w fazie Fetch można np. odczytać dane z BTB albo BHT, a w drugiej fazie podjąć ostateczną decyzję o wyniku predykcji. Takie rozwiązanie właśnie zastosowano. Podobną strukturę potokową można zastosować wewnątrz ID Strategy, jednak każdy z wybranych do implementacji algorytmów sprowadza się do prostego układu kombinacyjnego, dlatego ID Strategy nie rozbijano na fazy.

Jak widać na Rys. 4.1, PM Strategy bazuje więc wyłącznie na PC wewnątrz fazy Fetch, a ID Strategy na całej zdekodowanej instrukcji wewnątrz fazy Decode. Dodatkowo, ponieważ niektóre wybrane algorytmy predykcji korzystają z aktualnej wartości rejestru x1, wartość ta jest również bezpośrednio odczytywana z modułu Register i przekazywana do predyktorów. Wszystkie predyktory dynamiczne muszą również mieć możliwość aktualizacji swojego stanu na podstawie faktycznie wykonanych skoków przez mikroprocesor. W tym celu każdy moduł predyktora jest połączony z wyjściem z fazy Execute.

Wynikowa predykcja przesyłana jest do fazy Fetch. Na jej podstawie wyznaczany jest kolejny PC, podobnie jak w przypadku zwykłych skoków. Oprócz tego, moduł predyktora automatycznie wykrywa również poprawność przeszłych predykcji. Jeśli predykcja okazała się niepoprawna, należy podmienić PC w fazie Fetch na te sprzed wykonania predykcji oraz wyzerować odpowiednie fazy potoku za pomocą modułu Hazard Detection Unit. Jeśli predykcja była poprawna, to należy anulować skok aktualnie wykonywany w fazie Execute. Ostatnim sygnałem wyjściowym jest wymuszenie zerowania fazy Program Memory (za pośrednictwem Hazard Detection Unit). Jeśli predykcja wykonana została na podstawie ID Strategy, to w fazie Program Memory wczytywana jest aktualnie niepożądana instrukcja. Trzeba w takiej sytuacji tę fazę wyzerować. Sprawia to, że poprawnie przewidziane skoki przez ID Strategy mają koszt 1 pominiętego taktu zegara, ale ID Strategy może osiągnąć znacznie większą dokładność niż PM Strategy, ponieważ opiera predyckję na znacznie większej ilości informacji.

4.2. Wyznaczanie finalnej predykcji

Na Rys. 4.2. wyszczególniono wszystkie sygnały, wejściowe oraz wyjściowe, fragmentu modułu predykcji skoków, odpowiedzialnego za wyznaczenie ostatecznej predykcji, wraz z połączeniami między pozostałymi modułami procesora. Wszystkie strategie predykcji posiadają ujednolicony interfejs. Ponieważ więc format sygnałów wyjściowych strategii jest zawsze taki sam, przedstawiony fragment działa niezależnie od wybranego algorytmu predykcji.



Rys.4.2. *Schemat sygnałów związanych z wyznaczeniem ostatecznej predykcji*

Każda strategia zwraca jedynie 2 informacje – czy przewidziano wykonanie skoku oraz na jaki adres przewidziano wykonanie skoku. Są to sygnały *inject_from_pm* i *inject_form_pm_addr* dla PM Strategy, oraz *inject_from_id* i *inject_form_id_addr* dla ID Strategy. Sygnały te są ze sobą porównywane, po czym wyliczane zostają ostatecznie sygnały *if_inject* oraz *if_inject_addr*. Działają one analogicznie jak skoki, inicjowane z fazy Execute. Jeśli *if_inject* jest włączony, to PC w fazie Fetch zostaje podmienione na *if_inject_addr*, jednak tutaj inicjatorem jest predyktor, a nie faza Execute. Jeśli predykcja wykonana została bazując na ID Strategy, włączony zostaje sygnał wyjściowy *pm_inject_flush*, który informuje Hazard Detection Unit o wyzerowaniu fazy Program Memory. Kod odpowiedzialny za wyznaczenie tych sygnałów przedstawiony jest na Rys. 4.3.

```
wire inject_from_id_matches_pm_c = pm_inject_r &&
                                   (inject_from_id_addr_c == pm_inject_addr_r);
assign pm_inject_flush_c     = inject_from_id_c && !inject_from_id_matches_pm_c;
assign if_inject_c           = inject_from_pm_c || pm_inject_flush_c;
assign if_inject_addr_c      = pm_inject_flush_c ?
                                  inject_from_id_addr_c :
                                  inject_from_pm_addr_c;
```

Rys.4.3. *Kod wyznaczający sygnały if_inject, if_inject_addr oraz pm_inject_flush*

Takie pozorne skomplikowanie tej prostej zależności wynika z tego, żeby nie przywidzieć przypadkiem 2 razy tego samego skoku. Jeśli PM Strategy dokona predykcji, to w następnym takcie zegara ID Strategy będzie dokonywał predykcji dokładnie tej samej instrukcji. ID Strategy posiada co do zasady większą dokładność, dlatego powinien brać priorytet nad predykcją z poprzedniego taktu. Jeśli jednak wynik ID Strategy jest identyczny z PM Strategy z poprzedniego taktu, żadna nowa predykcja nie powinna mieć miejsca. Sygnał *inject_form_id_matches_pm_c* reprezentuje dokładnie to, czy wynik ID Strategy powinien w ogóle być brany pod uwagę. Wcześniej nie opisany sygnał *pm_inject_r* określa, czy w poprzednim takcie zegara sygnał *if_inject* był włączony (innymi słowy, czy aktualnie w fazie Program Memory potoku znajduje się instrukcja wczytywana na skutek wykonania predykcji). Jeśli taka sytuacja miała miejsce, oraz adres docelowy owej predykcji (*pm_injexct_addr_r*) jest taki sam, jak adres aktualnie przewidywany przez moduł ID Strategy, oznacza to, że jego predykcja powinna zostać odrzucona. Sygnał *pm_inject_flush* determinuje więc dodatkowo, która strategia ma priorytet w danej sytuacji.

Oprócz samej predykcji, układ odpowiedzialny jest również za weryfikowanie poprawności predykcji. Dlatego jednym z wejść predyktora jest wyjście z fazy Execute, informujące o wykonaniu skoku (*jump*) oraz docelowym adresie (*jump_addr*). Sygnały te pochodzą z późniejszej fazy potoku, a więc porównać je należy z odpowiednią predykcją wykonaną w przeszłości. Odpowiada za to szereg przerzutników, które opóźniają finalne predykcje z poprzednich taktów zegara. Jeśli predykcja sprzed 3 taktów jest identyczna z aktualnie wykonywanym skokiem, predykcja jest poprawna i włączony zostaje sygnał *discard*, informujący fazę Fetch o odrzuceniu skoku z fazy Execute.

Jeśli błędnie przewidziano skok, który w rzeczywistości się w ogóle nie wykonał, należy powrócić do stanu sprzed wykonania predykcji. Do tego służą sygnały *rollback* oraz *rollback_addr*. Działają podobnie jak *if_inject* i *if_inject_addr*, ponieważ również de facto wymuszają wykonanie skoku. Muszą to być jednak osobne sygnały, ponieważ *rollback* nie jest predykcją, a więc nie powinna być w przyszłości wyznaczana jego zgodność z faktycznymi skokami. Adresem *rollback_addr* jest zawsze wartość PC sprzed predykcji, powiększona o 4.

Dla pełnej poprawności, predyktor otrzymuje również informację, czy faza Execute jest aktualnie wyzerowana. Jeśli tak, to nie powinno być dokonane żadne sprawdzenie poprawności poprzedniej predykcji. Sygnały *discard* oraz *rollback* są wtedy wyłączane, ponieważ wykonana wcześniej predykcja nie wpłynęła w żaden sposób na działanie systemu (ponieważ jej wykonanie zostało anulowane poprzez zerowanie), a więc nie trzeba w żaden sposób reagować na jej poprawność.

Warto zauważyć, że jeśli predykcja błędnie przewidziała wykonany skok, to nie trzeba stosować żadnego dodatkowego mechanizmu. Wystarczy po prostu pozwolić skokowi na

wykonanie się. Błędne predykcje wykonanego skoku nie mają więc żadnego dodatkowego kosztu względem niewykonania żadnej predykcji. Błędne predykcje niewykonanych skoków wymagają zainicjowanie „skoku powrotnego” poprzez *rollback*, co powoduje dodatkową stratę 3 taktów zegara względem alternatywnej sytuacji bez predyktora, gdzie skok w ogóle by nie wystąpił. Podsumowanie zaoszczędzonych lub straconych taktów zegara, zależnie od każdej możliwej konfiguracji predykcji i jej poprawności, przedstawiono w Tab. 4.1.

Tab.4.1. *Porównanie zaoszczędzonych (+) i straconych (-) taktów zegara dzięki zastosowaniu predyktora skoków, zależnie od rodzaju predykcji*

	Skok Wykonany	Skok Niewykonany
Predykcja z PM Strategy Poprawna	+3	0
Predykcja z ID Strategy Poprawna	+2	0
Predykcja z PM Strategy Niepoprawna	0	-3
Predykcja z ID Strategy Niepoprawna	0	-3

Warto zauważyć, że w analizowanym mikroprocesorze implementacja predyktora, który zawsze przewiduje niewykonanie skoku, jest tożsame z brakiem implementacji jakiegokolwiek predyktora. Mikroprocesor domyślnie działa w taki sposób, próbując zapełnić potok instrukcjami tak, jakby żaden skok nie wystąpił. Dopiero w momencie wystąpienia skoku przywraca potok do odpowiedniego stanu, zerując odpowiednie moduły i zmieniając PC w fazie Fetch.

Jedyne co pozostało, to zaimplementowanie w module Instruction Fetch logiki odpowiedzialnej za wyznaczenie kolejnego PC, na podstawie potencjalnych skoków i sygnałów zwracanych przez moduł predyktora. Kod odpowiedzialny za tę operację przedstawiony jest na Rys. 4.5.

```
assign pc_c = (i_rollback_jump)                    ? (i_rollback_jump_addr) :
              (i_jump_branch && !i_discard_jump)   ? (i_pc) :
              (i_if_inject)                        ? (i_if_inject_addr) :
                                                     (pc_r + 'h4);
```

Rys.4.5. *Wyznaczanie kolejnej wartości PC w fazie Fetch*

Sygnał *rollback* bierze więc priorytet nad zwykłymi skokami, a skoki biorą priorytet nad *inject*, pod warunkiem, że nie zostaną odrzucone poprzez sygnał *discard*. Jeśli żadna sytuacja nadpisująca PC nie wystąpiła, PC jest zwyczajnie inkrementowany do adresu kolejnej instrukcji.

4.3. Interfejsy modułów strategii

Wszystkie sygnały wejściowe i wyjściowe modułów strategii predyktora są ujednolicone. Na Rys. 4.6. przedstawiono zaprojektowany interfejs w języku SystemVerilog dla PM Strategy, na Rys. 4.7 dla ID Strategy, a na Rys. 4.8. interfejs przeznaczony do oceny poprawności wykonanych wcześniej predykcji, który może być wspólny dla obu rodzajów strategi. Nie zaimplementowano jednak żadnego algorytmu ID Strategy, który musiałby aktualizować swój stan na podstawie wykonanych skoków, więc w rzeczywistości interfejs ten jest wykorzystywany jedynie przez niektóre algorytmy PM Strategy. Wszystkie interfejsy są parametryzowalne pod względem szerokości bitowej adresów. Oprócz opisanych sygnałów w interfejsie, każdy moduł przyjmuje także jako wejście sygnał zegarowy, sygnał resetu, sygnał aktywujący moduł (*enable)* oraz sygnał wstrzymania pracy (*stall*), podobnie jak praktycznie każdy sekwencyjny układ w całym projekcie procesora.

```
interface riscv_next_strategy_from_pm_intf # (parameter ADDR_WIDTH = 16) (
    input [ADDR_WIDTH-1:0] i_if_pc,
    input [ADDR_WIDTH-1:0] i_pm_pc,
    input i_pm_flush,
    input [ADDR_WIDTH-1:0] i_ra_data,
    output                 o_inject,
    output [ADDR_WIDTH-1:0] o_inject_addr
);
endinterface
```

Rys.4.6. *Interfejs modułu PM Strategy*

Wejściem do interfejsu PM Strategy jest przede wszystkim PC. Sygnał *i_if_pc* to PC z fazy Fetch, przeznaczony do wykorzystania w pierwszej fazie PM Strategy, a *i_pm_pc*, to zawsze *i_if_pc* opóźniony o 1 takt zegara, a więc aktualny PC w fazie Program Memory, przeznaczony do wykorzystania w drugiej fazie PM Strategy. Dodatkowo, z Hazard Detection Unit odczytywana jest informacja o tym, czy aktualnie faza Program Memory jest zerowana (*i_pm_flush*). Niektóre algorytmy wymagają aktualizacji swojego stanu spekulatywnie, już w momencie wykonania samej predykcji. W takim przypadku, w razie wystąpienia zerowania, należy takie zmiany cofnąć. Sygnał *i_ra_data* to aktualna wartość rejestru x1. Sygnały wyjściowe *o_inject* oraz *o_inject_addr* są wynikiem predykcji.

```
interface riscv_next_strategy_from_id_intf # (parameter ADDR_WIDTH = 16) (
    input [ADDR_WIDTH-1:0] i_id_pc,
    input wire next_instr_signals_t i_id_signals,
    input i_id_flush,
    input [ADDR_WIDTH-1:0] i_ra_data,
    output                 o_inject,
    output [ADDR_WIDTH-1:0] o_inject_addr
);
endinterface
```

Rys.4.7. *Interfejs modułu ID Strategy*

W przypadku interfejsu ID Strategy występuje nowy sygnał, o nazwie *i_id_signals*. Jest to struktura zawierająca wszystkie potencjalnie ważne sygnały kontrolne, wynikłe na skutek zdekodowania instrukcji. Zawartość struktury *next_instr_signals_t* przedstawiona została na Rys. 4.9. Pozostałe sygnały są analogiczne jak w przypadku PM Strategy, ale odnoszą się do PC oraz wyzerowania fazy Decode.

```
interface riscv_next_strategy_history_intf # (parameter ADDR_WIDTH = 16) (
    input [ADDR_WIDTH-1:0] i_pc,
    input i_flush,
    input i_jump_branch,
    input [ADDR_WIDTH-1:0] i_jump_addr,
    input wire next_instr_signals_t i_signals
);
endinterface
```

Rys.4.8. *Interfejs aktualizacji danych na podstawie wykonanych skoków*

W ostatnim interfejsie wszystkie sygnały odnoszą się do wyniku działania fazy Execute. Sygnał *i_pc* reprezentuje adres instrukcji skoku, *i_flush* stan wyzerowania danej fazy, *i_jump_branch* decyzję o podjęciu skoku, a *i_jump_addr* adres docelowy skoku. Dodatkowo, przekazywane są wszystkie adekwatne sygnały zdekodowanej instrukcji, podobnie jak miało to miejsce w interfejsie ID Strategy.

```
typedef struct packed {
    logic jal;
    logic jalr;
    logic branch;
    logic rs1_zero;
    logic rs1_ra;
    logic imm_sign;
    logic [32-1 : 0] imm;
} next_instr_signals_t;
```

Rys.4.9. *Struktura next_instr_signals_t*

Sygnałami zawartymi w strukturze *next_instr_signals_t* są kolejno: to czy instrukcja jest typu JAL, typu JALR, typu BR, czy pierwszy rejestr źródłowy jest równy x0, czy jest równy x1, bit znaku wartości *immediate* oraz sama wartość *immediate*.

4.4. Strategia PM – Hardcoded

Pierwszy i najprostszy zaimplementowany algorytm predykcji nazwany został Hardcoded. Predyktor musi zostać skonfigurowany już na etapie syntezy, za pomocą odpowiednich makr. Pełny kod strategii przedstawiono na Rys. 4.10.

```
module riscv_next_strategy_hardcoded # (
    parameter ADDR_WIDTH = 64
) (
    input clk,
    input enable,
    input nreset,
    input i_stall,

    riscv_next_strategy_from_pm_intf intf
);

    assign intf.o_inject       = intf.i_pm_pc == `HARDCODED_FROM;
    assign intf.o_inject_addr  = `HARDCODED_TO;

endmodule
```

Rys.4.10. *Implementacja strategi Hardcoded*

Jak widać, trudno wymyślić mniej skomplikowany algorytm. Predykcja wykonania skoku dokonywana jest zawsze, kiedy adres PC w fazie Program Memory jest równy skonfigurowanemu, a adres docelowy predykcji jest ustawiony na stałe. Moduł ten raczej nie ma sensu w zastosowaniu w praktyce, ale pozwala na „wstrzykiwanie" predykcji w dowolne miejsce w programie, co znacznie ułatwia analizę, czy wszystkie sygnały wyjściowe modułu predyktora są wyznaczane poprawnie, jak i czy mikroprocesor w ogóle działa poprawnie.

4.5. Strategia PM – Random

Podobnie jak Hardcoded, strategia o nazwie Random również służy przede wszystkim do weryfikacji poprawności działania procesora, niezależnie od wyników algorytmu predykcji. Kod modułu strategi zamieszczono na Rys. 4.11.

```
module riscv_next_strategy_random # (
    parameter ADDR_WIDTH = 64,
    parameter OFFSET = 5
) (
    input clk,
    input enable,
    input nreset,
    input i_stall,

    riscv_next_strategy_from_pm_intf intf
);

    wire [31:0] rng_res;
    reg [ADDR_WIDTH-1:0] last_pm_pc;
    always @(posedge clk) begin
        if(!nreset) begin
            last_pm_pc <= 0;
        end else if(enable) begin
            last_pm_pc <= intf.i_pm_pc;
        end
    end
```

```
    assign intf.o_inject      = rng_res[OFFSET-1:0] != 0 &&
                                rng_res[OFFSET-1:0] != 1 && rng_res[31:30] == 0;
    assign intf.o_inject_addr = $signed({rng_res[OFFSET-1:0], 2'b00})
                                + intf.i_pm_pc;
    wire generate_new_rng = enable && (intf.i_pm_pc != last_pm_pc);
    RNG rng (
        .clk        (clk),
        .enable     (generate_new_rng),
        .nreset     (nreset),
        .res        (rng_res)
    );

endmodule

module RNG(
    input clk,
    input enable,
    input nreset,
    output wire [31:0] res
);

localparam [31:0] init_state = 32'd2463534242;
localparam [31:0] multiplier = 32'd3084775641;

reg [31:0] state;
reg [63:0] mul_res;

assign res = mul_res[63:32];

always @(posedge clk) begin
    if(!nreset) begin
        state   = init_state;
        mul_res = 0;
    end else if(enable) begin
        state ^= (state << 13);
        state ^= (state >> 17);
        state ^= (state << 5);
        mul_res = state * multiplier;
        mul_res[63:32] = state;
    end
end
endmodule
```

Rys.4.11. *Implementacja strategi Random*

Jak nazwa wskazuje, predykcja dokonywana jest za każdym razem losowo. Algorytm wykorzystuje generator liczb pseudolosowych, działający poprzez połączenie funkcji XOR, przesunięć bitowych, oraz mnożenia. Konkretna implementacja generatora liczb pseudolosowych nie jest ważna, ponieważ nie potrzeba ani dobrej jakości rozkładu wynikowych liczb, ani nie potrzeba struktury optymalnej do syntezy. Strategia wykorzystywana jest jedynie w symulacji, do testów. Na podstawie wybranych bitów wynikowej liczby losowej, generowany jest adres docelowy predykcji, jak i w ogóle to czy predykcja ma zostać wykonana.

4.6. Strategia PM – BTB

Najprostszą zaimplementowaną strategią, która mogłaby zostać zastosowana w praktyce, jest strategia BTB. Bazuje ona, jak nazwa wskazuje, na buforze BTB. Moduł BTB wiąże adresy instrukcji z licznikami nasycenia oraz dodatkową wartością. Dodatkowa wartość w przypadku tej strategi to adres docelowy skoku, lecz ten sam moduł wykorzystywany jest również w innych strategiach, gdzie jest inaczej. Interfejs BTB (nie strategi, wyłącznie samego modułu buforu) przedstawiono na Rys. 4.12.

```
module BTB_COUNTER #(
    parameter ADDR_WIDTH = 64,
    parameter INDEX_WIDTH = 3,
    parameter VALUE_WDITH = 64,
    parameter COUNTER_WIDTH = 2
)(
    input clk,
    input nreset,
    input enable,

    input i_stall,

    input  [ADDR_WIDTH-1  : 0] i_read_addr,
    output [VALUE_WDITH-1 : 0] o_read_value,
    output                     o_read_jump,
    output                     o_read_valid,

    input  [ADDR_WIDTH-1  : 0] i_write_addr,
    input  [VALUE_WDITH-1 : 0] i_write_value,
    input                      i_write_jump,
    input                      i_write_enable
);
```

Rys.4.12. *Interfejs modułu BTB*

Moduł parametryzowalny jest pod względem szerokości adresów, liczby przechowywanych wpisów (wyznaczanej na podstawie parametru *INDEX_WIDTH*), szerokości przechowywanej wartości, oraz szerokości liczników nasycenia.

Wraz ze zboczem sygnału zegarowego, wartość przypisana adresowi *i_read_addr* jest odczytywana. Sygnał wyjściowy *o_read_valid* określa, czy BTB posiada rekord odpowiadający żądanemu adresowi, a więc czy wyjście z BTB powinno być jakkolwiek interpretowane przez dalszą część algorytmu. Sygnał o*_read_jump* określa, czy według przypisanego licznika nasycenia skok powinien się wykonać. Jest to po prostu najbardziej znaczący bit licznika. Sygnał *o_read_value* to przechowywana w danym rekordzie wartość. W przypadku omawianej strategii, wartością tą jest adres docelowy predykcji.

Oprócz samego odczytywania, są zdefiniowane również sygnały pozwalające na dodawanie rekordów do BTB. Jeśli sygnał *i_write_enable* jest włączony, to odpowiedni rekord odpowiadający adresowi *i_write_addr* jest aktualizowany, poprzez podmianę adresu docelowego na *i_write_value* i dekrementacji lub inkrementacji licznika, zależnie od *i_write_jump*.

Liczba przechowywanych rekordów jest zależna od parametru *INDEX_WIDTH*. Określa on rozmiar w bitach indeksu tablicy, w której przechowywane są wszystkie rekordy. Maksymalna liczba możliwych do przechowania wpisów wynosi więc 2^{INDEX_WIDTH}. Jako indeks wykorzystywane są najmłodsze bity adresu instrukcji. Trzeba zauważyć jednak jeden problem – jeśli dwie różne instrukcje będą posiadać takie same najmniej znaczące bity adresu, to będą się one odnosiły do tego samego rekordu. W celu odróżnienia, z którym konkretnie adresem dany rekord jest powiązany, oprócz adresu docelowego oraz licznika, przechowywane są także pozostałe bity PC. Bity te nazwane zostały kluczem (*key*) rekordu, choć potencjalnie adekwatniejszą nazwą byłby „tag”. Pełny format wpisu w BTB przedstawiono na Rys. 4.13.

```
localparam KEY_WIDTH   = ADDR_WIDTH - INDEX_WIDTH;
typedef struct packed {
    logic                    valid;
    logic [KEY_WIDTH-1:0]    key;
    logic [VALUE_WDITH-1:0]  value;
    logic [COUNTER_WIDTH-1:0] cnt;
} entry_t;
```

Rys.4.13. *Format struktury entry_t, reprezentującej wpis w module BTB*

Podczas inicjalizacji BTB, bit *valid* w każdym rekordzie zostaje wyzerowany.

Fragment kodu odpowiedzialny za wyznaczanie rekordów na podstawie adresu zamieszczono na Rys. 4.14.

```
localparam ENTRIES_COUNT = 2**INDEX_WIDTH;
var  entry_t entries [0:ENTRIES_COUNT-1];

wire [INDEX_WIDTH-1:0] read_index_c  = i_read_addr [INDEX_WIDTH-1:0];
wire [INDEX_WIDTH-1:0] write_index_c = i_write_addr[INDEX_WIDTH-1:0];

wire [KEY_WIDTH-1:0]   read_key_c  = i_read_addr [ADDR_WIDTH-1:INDEX_WIDTH];
wire [KEY_WIDTH-1:0]   write_key_c = i_write_addr[ADDR_WIDTH-1:INDEX_WIDTH];

wire entry_t read_entry_c  = entries[ read_index_c];
wire entry_t write_entry_c = entries[write_index_c];

wire read_key_matches_c =  read_entry_c.valid &&
                           read_entry_c.key   == read_key_c;
wire write_key_matches_c = write_entry_c.valid &&
                           write_entry_c.key   == write_key_c;
```

Rys.4.14. *Kod wyznaczający odpowiednie wpisy BTB na podstawie adresu odczytu oraz zapisu*

Najpierw na podstawie adresu wyznaczane są indeks oraz klucz. Z tablicy przechowującej wszystkie rekordy wybierane są te pod wyznaczonym indeksem. Na końcu, na podstawie bitu *valid* zwróconego rekordu, oraz porównania klucza rekordu z kluczem wyznaczonym na podstawie adresu, określane jest, czy w danym rekordzie przechowywany jest aktualnie wpis odpowiadający zadanemu adresowi instrukcji.

Jeśli sygnał *read_key_matches_c* jest włączony, to włączony zostaje również wyjściowy sygnał *o_read_valid*, ponieważ oznacza on, że pomyślnie odczytano rekord

odnoszący się do danego adresu. W przypadku zapisu, jeśli sygnał *i_write_enable* jest włączony oraz *klucze* są zgodne, to wartość *value* zostaje podmieniona, oraz licznik odpowiednio zinkrementowany lub zdekrementowany. Jeśli jednak klucze nie są zgodne, to klucz również zostaje podmieniony, a licznik jest reinicjalizowany na odpowiednią wartość, zależnie od *i_write_jump*. Przedstawia to kod na Rys. 4.15.

```
localparam COUNTER_MAX    = {COUNTER_WIDTH{1'b1}};
localparam COUNTER_MIN    = {COUNTER_WIDTH{1'b0}};
localparam COUNTER_HALF_1 = {1'b1, {(COUNTER_WIDTH - 1){1'b1}}};
localparam COUNTER_HALF_0 = {1'b0, {(COUNTER_WIDTH - 1){1'b1}}};
wire write_counter_is_max_c = write_entry_c.cnt == COUNTER_MAX;
wire write_counter_is_min_c = write_entry_c.cnt == COUNTER_MIN;

wire [COUNTER_WIDTH-1:0] write_next_counter_c =
 i_write_jump && (write_entry_c.cnt != COUNTER_MAX) ? write_entry_c.cnt + 1'd1 :
!i_write_jump && (write_entry_c.cnt != COUNTER_MIN) ? write_entry_c.cnt - 1'd1 :
                                                     write_entry_c.cnt;
// ...
if(i_write_enable) begin
    if(write_key_matches_c) begin
        entries[write_index_c].cnt   <= write_next_counter_c;
        entries[write_index_c].value <= i_write_value;
    end else begin
        `ifdef SIMULATION
        evicts += write_entry_c.valid;
        `endif
        entries[write_index_c] <= '{
            valid: 1'd1,
            key:   write_key_c,
            value: i_write_value,
            cnt:   i_write_jump ? COUNTER_HALF_1 : COUNTER_HALF_0
        };
    end
end
```

Rys.4.15. Kod podpowiadający na zapis rekordów BTB

Wartości *COUNTER_HALF_1* oraz *COUNTER_HALF_0* określają odpowiednio wartości licznika reprezentujące skok wykonany oraz niewykonany, o minimalnym możliwym poziomie nasycenia.

Podsumowując, BTB jest w stanie przechować 2^{INDEX_WIDTH} rekordów. Jeśli adresy dwóch instrukcji mają takie same mniej znaczące bity, to jedna będzie musiała wyprzeć z BTB drugą podczas zapisu. Im więc większy parametr *INDEX_WIDTH*, tym więcej możliwe jest do zapamiętania instrukcji, jak i tym mniejsza szansa na wystąpienie kolizji. Oczywiście, wraz ze zwiększaniem *INDEX_WIDTH*, zwiększa się powierzchnia, jaką BTB będzie zajmował po syntezie na fizycznym urządzeniu, więc nie można wartości tej ustawić dowolnie dużej.

Znając już zasadę działania samego BTB, można przejść do omówienia samej strategii BTB. Jej implementację przedstawiono na Rys. 4.16.

```
module riscv_next_strategy_btb_counter # (
    parameter ADDR_WIDTH = 64
) (
    input clk,
    input enable,
    input nreset,
    input i_stall,

    riscv_next_strategy_from_pm_intf intf,
    riscv_next_strategy_history_intf hist
);

    localparam BTB_ADDR_WIDTH    = ADDR_WIDTH - 2;
    localparam BTB_INDEX_WIDTH   = `BTB_INDEX_WIDTH;
    localparam BTB_COUNTER_WIDTH = `BTB_COUNTER_WIDTH;
    localparam BTB_VALUE_WIDTH   = BTB_ADDR_WIDTH;

    wire [BTB_ADDR_WIDTH-1 : 0] btb_read_value_c;
    wire                        btb_read_jump_c;
    wire                        btb_read_valid_c;

    wire hist_could_jump = hist.i_signals.jal || hist.i_signals.jalr ||
                           hist.i_signals.branch;
    wire btb_write_enable = hist_could_jump && !hist.i_flush;

    BTB_COUNTER #(
        .ADDR_WIDTH     (BTB_ADDR_WIDTH),
        .INDEX_WIDTH    (BTB_INDEX_WIDTH),
        .VALUE_WDITH    (BTB_VALUE_WIDTH),
        .COUNTER_WIDTH  (BTB_COUNTER_WIDTH)
    ) btb (
        .clk    (clk),
        .nreset (nreset),
        .enable (enable),
        .i_stall (i_stall),

        .i_read_addr    (intf.i_if_pc[BTB_ADDR_WIDTH+1 : 2]),
        .o_read_value   (btb_read_value_c),
        .o_read_jump    (btb_read_jump_c),
        .o_read_valid   (btb_read_valid_c),

        .i_write_addr   (hist.i_pc[BTB_ADDR_WIDTH+1 : 2]),
        .i_write_value  (hist.i_jump_addr[BTB_ADDR_WIDTH+1 : 2]),
        .i_write_jump   (hist.i_jump_branch),
        .i_write_enable (btb_write_enable)
    );

    assign intf.o_inject       = btb_read_valid_c && btb_read_jump_c;
    assign intf.o_inject_addr  = {btb_read_value_c, 2'b00};
```

Rys.4.16. *Implementacja strategi BTB*

W przeciwieństwie do wcześniej zaimplementowanych strategii, strategia BTB musi aktualizować swój stan na podstawie wykonanych skoków. Dlatego wykorzystuje również interfejs *hist*.

Pierwszą ważną optymalizację można zauważyć już w wyznaczaniu wartości parametru *BTB_ADDR_WIDTH*. Ponieważ instrukcje mają rozmiar 4 bajtów (dany mikroprocesor nie wspiera skompresowanych instrukcji) oraz muszą występować na adresach będących wielokrotnością owych 4 bajtów, to 2 najmniej znaczące bity adresów instrukcji są zawsze zerami. Nie trzeba więc ich nigdzie przechowywać, ani stosować jako części indeksu.

Odczytywanie wartości z BTB odbywa się w pierwszej fazie działania strategi, na podstawie PC z fazy Fetch. Ponieważ odczyt trwa jeden cykl zegara, predykcja dokonywana jest w drugiej fazie, kiedy dany PC znajduje się w Program Memory. Jeśli odczytano poprawie wartość (*btb_read_valid_c*) oraz najbardziej znaczący bit licznika wynosi 1 (*btb_read_jump_c*), to przewidywane jest, że skok zostanie wykonany. Adres docelowy predykcji to również wartość zwrócona przez BTB, rozszerzona o 2 zerowe bity, aby odwrócić poprzednią optymalizację.

Zapisywanie wartości do BTB odbywa się za pomocą interfejsu *hist*. Jeżeli instrukcją, która została przetworzona w fazie Execute, był JAL, JALR lub BR, oraz faza ta nie została wyzerowana, sygnał *btb_write_enable* jest włączany i dokonywana jest aktualizacja stanu BTB. Adres instrukcji, adres docelowy skoku (nie licząc najmniej znaczących 2 bitów) oraz to, czy skok się wykonał, przekazywane jest z interfejsu *hist* do BTB. BTB nie jest więc nigdy aktualizowany spekulatywnie.

4.7. Strategia PM – BHT

Strategia BHT jest bardzo podobna do strategi BTB. Zmiana wynika ze sposobu predykcji skoków warunkowych. Wartością przechowywaną w BTB jest nie tylko adres docelowy skoku, ale również 1 bit określający, czy instrukcja pod danym adresem jest skokiem warunkowym. Stworzono moduł BHT, o strukturze w większości identycznej jak moduł BTB, gdzie do wyznaczenia indeksu udział bierze nie tylko adres instrukcji, ale również historia wykonania wcześniej przetwarzanych skoków warunkowych. Interfejs modułu przedstawiony został na Rys. 4.17.

```
module HIST_COUNTER #(
    parameter ADDR_WIDTH = 64,
    parameter HIST_WIDTH = 10,
    parameter COUNTER_WIDTH = 2,

    parameter WRITE_CYCLES_OFFSET = 2
)(
    input clk,
    input nreset,
    input enable,
    input i_stall,

    input                       i_flush_read_hisotry,
    input                       i_shift_read_history,
    input  [ADDR_WIDTH-1  : 0] i_read_addr,
    output                      o_read_jump,
    output                      o_read_valid,

    input  [ADDR_WIDTH-1  : 0] i_write_addr,
    input                       i_write_jump,
    input                       i_write_enable
);
```

Rys.4.17. *Interfejs modułu BHT*

Większość sygnałów interfejsu BHT jest identyczna jak w BTB. Brakuje jednak sygnałów odczytu i zapisu *value*. BHT nie musi bowiem przechowywać żadnej dodatkowej informacji oprócz licznika nasycenia. Widać też 2 nowe sygnały – *i_flush_read_history* oraz *i_shift_read_history*. Służą one do spekulatywnego aktualizowania stanu BHT i zostaną opisane dokładnie w dalszej części pracy. BHT zawiera wewnątrz siebie 2 rejestry przesuwające, o szerokości równej parametrowi *HIST_WIDTH*. Jeden jest historią skoków dla odczytu, a drugi dla zapisu. Wraz z napotkaniem instrukcji skoku warunkowego, gdzie skok miał miejsce, do rejestru wsuwane jest 1, a jeśli nie został wykonany, wsuwane jest 0. Rejestry te reprezentują więc historię skoków warunkowych. Indeks do tablicy liczników wyznaczany jest na podstawie wyniku funkcji XOR odpowiedniego rejestru historii oraz najmniej znaczących bitów adresu instrukcji. Szerokość bitowa indeksu jest równa *HIST_WIDTH*. Kod dotyczący wyznaczania wartości rejestrów oraz indeksu przedstawiono na Rys. 4.18.

```
localparam INDEX_WIDTH = HIST_WIDTH;
localparam EXT_HIST_WIDTH = HIST_WIDTH > WRITE_CYCLES_OFFSET ?
                HIST_WIDTH : WRITE_CYCLES_OFFSET;

reg  [EXT_HIST_WIDTH - 1 : 0] read_history_r;
reg  [HIST_WIDTH - 1 : 0]     write_history_r;
reg  [$clog2(WRITE_CYCLES_OFFSET) - 1 : 0] speculative_offset_r;

wire expected_write_jump_c = {read_history_r, 1'b0} >> speculative_offset_r;
wire was_last_prediction_wrong_c =
     (speculative_offset_r == 0 || expected_write_jump_c != i_write_jump) &&
     i_write_enable;

wire [INDEX_WIDTH-1:0] read_index_c  =
             i_read_addr [INDEX_WIDTH-1:0] ^ read_history_r[HIST_WIDTH-1:0];
wire [INDEX_WIDTH-1:0] write_index_c =
             i_write_addr[INDEX_WIDTH-1:0] ^ write_history_r;

wire [EXT_HIST_WIDTH-1 : 0] shifted_read_history_c  =
             {read_history_r , read_jump_r};
wire [HIST_WIDTH-1 : 0]     shifted_write_history_c =
             {write_history_r, i_write_jump};
wire [HIST_WIDTH-1 : 0]     next_write_history_c =
             i_write_enable ? shifted_write_history_c : write_history_r;
// ..
always @(posedge clk) begin
  // ..
  if(was_last_prediction_wrong_c || i_flush_read_hisotry) begin
      read_history_r       <= next_write_history_c;
      speculative_offset_r <= 0;
  end else begin
      if( (i_shift_read_history) && (!i_write_enable))
          speculative_offset_r <= speculative_offset_r + 1'd1;
      if((!i_shift_read_history) &&  (i_write_enable))
          speculative_offset_r <= speculative_offset_r - 1'd1;
      if(i_shift_read_history)
          read_history_r <= shifted_read_history_c;
  end
  write_history_r <= next_write_history_c;
```

Rys.4.18. *Wyznaczanie historii i indeksu w module BHT*

Rejestr nazwany *write_history_r* reprezentuje historię instrukcji, które przetworzone zostały przez fazę Execute. Z każdym taktem sygnału zegarowego, jeśli zapis jest aktywny

(sygnał *i_write_enable*), przypisywana jest mu wartość *shifted_write_history_c*, czyli wsuwana jest wartość *i_write_jump*, określająca to, czy skok został wykonany. Indeks aktualizowanych rekordów jest wyznaczany na podstawie właśnie *write_history_r*.

Dużo bardziej skomplikowanie sytuacja wygląda w przypadku *read_history_r*. Od niej zależy indeks rekordów odczytywanych. Problem jest z aktualizacją tego rejestru. Jeśli predykcja została wykonana za pomocą BHT, to kolejna instrukcja, która trafi do potoku, będzie wynikać z tej właśnie predykcji. Ponieważ BHT służy do przewidywania jedynie skoków warunkowych, to ta nowa instrukcja powinna nieść z sobą inną historię skoków, niż sprzed wykonania predykcji. Historia musi się więc zmieniać spekulatywnie, od razu przy wykonaniu predykcji. Jeśli później okaże się, że predykcja była niepoprawna, należy *read_history_r* przywrócić do poprawnego stanu, sprzed predykcji.

Szerokość rejestru *read_history_r* musi wynosić co najmniej tyle, ile faz potoku znajduje się między fazami Program Memory oraz Execute włącznie, co w tym przypadku wynosi 2. Wymagana jest możliwość określenia poprawności wykonanej predykcji sprzed co najmniej 2 taktów zegara. Dlatego trzeba zapewnić, że *read_history_r*, która jest de facto historią wykonanych predykcji, a nie skoków, będzie miała rozmiar co najmniej 2 bitów. Do tego służą parametry *WRITE_CYCLES_OFFSET* oraz *EXT_HIST_WIDTH*. Oprócz samej predykcji, trzeba również pamiętać, o ile aktualnie historia odczytu „wyprzedza" historię zapisu. Wartość ta przechowywana jest w liczniku *speculative_offset_r*. Za każdym razem kiedy wykonywana jest instrukcja skoku warunkowego w fazie Execute (*i_write_enable*), odpowiedni bit historii odczytu (o indeksie zależnym od *speculative_offset_r*) jest porównywany z tym, czy faktycznie skok został wykonany (*i_write_jump*). Jeśli wartości są niezgodne, należy cofnąć wszystkie spekulatywnie wprowadzone zmiany. Również, jeśli *speculative_offset_r* jest równy 0, czyli nie oczekujemy wystąpienia skoku warunkowego, a skok warunkowy wystąpi, należy zaktualizować historię odczytu, niezależnie czy dokonujemy aktualnie predykcji. Obie te informacje są reprezentowane przez sygnał *was_last_prediction_wrong_c*. Jest jeszcze jedna sytuacja, gdzie wymagane jest usunięcie spekulatywnych zmian – jeśli wykonany został jakikolwiek skok lub *rollback*. Taka sytuacja oznacza, że wszelkie potencjalne zmiany spekulatywne zostaną wykonane względem instrukcji, które zostały właśnie wyzerowane. Informacja ta przekazana zostać powinna do modułu BHT za pomocą wejścia *i_flush_read_history*. Jeśli którakolwiek z opisanych sytuacji wystąpiła, wewnątrz przedstawionego na Rys. 4.18 bloku *always* wartość historii odczytu zostaje ustawiona na historię zapisu, a licznik spekulatywnych aktualizacji zostaje wyzerowany. Moduł musi też wiedzieć, kiedy dokonywana jest na jego podstawie predykcja. Informacja ta powinna zostać przekazana poprzez wejście *i_shift_read_history*.

Wiedząc już jak działa moduł BHT, można przejść do analizy samej strategii BHT. Na Rys. 4.19. przedstawiono wszystkie zmiany względem strategii BTB.

```
    wire bht_shift_read   = btb_read_value_branch_c && btb_read_valid_c &&
                            bht_read_valid_c ;
    wire bht_write_enable = hist_was_branch && !hist.i_flush;

    HIST_COUNTER #(
        .ADDR_WIDTH     (BHT_ADDR_WIDTH),
        .HIST_WIDTH     (BHT_HIST_WIDTH),
        .COUNTER_WIDTH  (BHT_COUNTER_WIDTH)
    ) bht (
        .clk    (clk),
        .nreset (nreset),
        .enable (enable),
        .i_stall (i_stall),

        .i_flush_read_hisotry (intf.i_pm_flush),
        .i_shift_read_history (bht_shift_read),
        .i_read_addr    (intf.i_if_pc[BHT_ADDR_WIDTH+1 : 2]),
        .o_read_jump    (bht_read_jump_c),
        .o_read_valid   (bht_read_valid_c),

        .i_write_addr   (hist.i_pc[BHT_ADDR_WIDTH+1 : 2]),
        .i_write_jump   (hist.i_jump_branch),
        .i_write_enable (bht_write_enable)
    );

    assign intf.o_inject = btb_read_valid_c && (
        (btb_read_value_branch_c && bht_read_valid_c) ? bht_read_jump_c :
                                                        btb_read_jump_c );
    assign intf.o_inject_addr  = {btb_read_value_addr_c, 2'b00};
```

Rys.4.19. *Fragment implementacji strategii BHT*

Odczyt dokonywany jest w każdym takcie zegara zarówno z modułów BTB oraz BHT. Jeśli BTB zwrócił wartość, w której włączona była flaga określająca, że dana instrukcja jest skokiem warunkowym oraz moduł BHT zwrócił w tym samym takcie jakikolwiek wynik (sygnał *bht_read_valid_c* został włączony, czyli posiadał rekord odpowiadający danemu adresowi) decydowane jest, czy jako wynik predykcji zastosować licznik z BTB czy z BHT. Jeśli skorzystano z wyniku zwróconego przez BHT, włączane jest wejście *i_shift_read_history*, aby spekulatywnie zaktualizować historię skoków warunkowych. Syngał *i_flush_read_history* powiązany jest z sygnałem *intf.i_pm_flush*, który występuje dokładnie wtedy, kiedy wykonany został skok lub *rollback*. Wszystkie inne aspekty działania są analogiczne jak w przypadku strategi BTB.

Strategia BHT pozwala więc na dokładnie taki sam mechanizm przewidywania, jak strategia BTB, ale z udoskonaleniem poprzez dokładniejszą predykcję skoków warunkowych, bazując na przeszłej historii skoków, a nie tylko adresu instrukcji. Jeśli dokonywana jest predykcja skoku warunkowego, ale BHT nie posiada jeszcze rekordu odpowiadającemu danej kombinacji PC i historii, priorytet bierze predykcja na podstawie licznika z BTB, a nie BHT.

4.8. Strategia PM – BHTRET

Ostatnią zaimplementowaną strategią typu PM jest BHTRET. Działa ona identycznie, jak strategia BHT, ale posiada dodatkową optymalizację w postaci dedykowanego modułu do przewidywania instrukcji JALR będących powrotem z funkcji. Zmiany względem strategi BHT przedstawiono na Rys. 4.20.

```
    wire ret_table_write_enable_c = hist_was_ret && !hist.i_flush;

    RET_TABLE #(
        .ADDR_WIDTH     (RET_ADDR_WIDTH),
        .INDEX_WIDTH    (RET_INDEX_WIDTH)
    ) ret_table (
        .clk    (clk),
        .nreset (nreset),
        .enable (enable),
        .i_stall (i_stall),

        .i_read_addr    (intf.i_if_pc[BHT_ADDR_WIDTH+1 : 2]),
        .o_read_valid   (ret_table_read_valid_c),

        .i_write_addr   (hist.i_pc[BHT_ADDR_WIDTH+1 : 2]),
        .i_write_enable (ret_table_write_enable_c)
    );

    assign intf.o_inject = ret_table_read_valid_c || (btb_read_valid_c && (
        (btb_read_value_branch_c && bht_read_valid_c) ? bht_read_jump_c :
                                                        btb_read_jump_c ));
    assign intf.o_inject_addr  = ret_table_read_valid_c ? intf.i_ra_data :
{btb_read_value_addr_c, 2'b00};
```

Rys.4.20. *Fragment implementacji strategii BHTRET*

Moduł RET_TABLE działa praktycznie identycznie jak BTB, więc nie zostanie omówiony osobno. Jedyną różnicą jest to, że nie przechowuje on ani licznika, ani żadnej dodatkowej wartości. Jedyne co przechowuje, to klucze rekordów oraz flagi *valid*. Jeśli podczas odczytu, RET_TABLE zwrócił informację, że dany rekord istnieje i klucz pasuje do adresu (*o_read_valid* jest włączony), predyktor uznaje, że przewidywaną instrukcją jest RET, a więc adres docelowy skoku zostaje zaczerpnięty z rejestru x1 (sygnał *intf.i_ra_data*). Jeśli wystąpił skok na skutek instrukcji RET, zamiast aktualizować stan modułu BTB, aktualizowany jest jedynie RET_TABLE.

Wszystko to pozwala odciążyć moduł BTB, ponieważ nie musi już pamiętać informacji o instrukcjach RET. Zmniejsza to ryzyko kolizji różnych skoków, powiązanych z tym samym indeksem. Moduł BTB nie radzi sobie za dobrze z instrukcjami JALR (jakim jest RET), ponieważ adres docelowy takich skoków często się może zmieniać, a BTB jest w stanie zapamiętać jedynie jeden adres docelowy naraz.

Analogiczną optymalizację można dokonać na dowolnej innej strategii typu PM, jednak zaimplementowano ją jedynie dla strategii BHT.

4.9. Strategia ID – JUMP

Zaimplementowana strategia rodzaju ID, o prostej nazwie JUMP, pozwala na bezbłędne przewidywanie dowolnego skoku bezwarunkowego bezwzględnego. Takimi skokami są oczywiście instrukcje JAL. Jednak dodatkowo, możliwe jest wystąpienie instrukcji JALR, której rejestr źródłowy to x0, o stałej wartości 0. W takim wypadku, skok JALR staje się skokiem bezpośrednim, którego adres docelowy jest równy wartości *immediate*. Kod strategii przedstawiono na Rys. 4.21.

```
module riscv_next_strategy_compute_jump # (
    parameter ADDR_WIDTH = 64,
    parameter INSTR_WIDTH = 64
) (
    input clk,
    input enable,
    input nreset,

    riscv_next_strategy_from_id_intf intf
);

    wire relative_jump_c = intf.i_id_signals.jal;
    wire absolute_jump_c = intf.i_id_signals.jalr && intf.i_id_signals.rs1_zero;

    wire [ADDR_WIDTH-1:0] relative_jump_addr_c =
                                intf.i_id_signals.imm + intf.i_id_pc;
    wire [ADDR_WIDTH-1:0] absolute_jump_addr_c = intf.i_id_signals.imm;

    assign intf.o_inject        =
              (relative_jump_c || absolute_jump_c) &&
              !intf.i_id_flush;
    assign intf.o_inject_addr   =
               relative_jump_c ? relative_jump_addr_c : absolute_jump_addr_c;
endmodule
```

Rys.4.21. *Implementacja strategii JUMP*

Implementacja jest bardzo prosta. Na początku ustalane jest, czy dana instrukcja jest typu JAL lub JALR z rejestrem x0. Jeśli którykolwiek z tych warunków jest spełniony, oraz instrukcja która jest aktualnie przewidywana nie pochodzi przypadkiem z aktualnie zerowanej fazy potoku, predykcja jest wykonywana. Adresem docelowym jest albo wartość *immediate*, albo suma *immediate* oraz PC, zależnie od rodzaju instrukcji.

4.10. Strategia ID – JUMPRET

Dla strategii typu ID również zastosowano optymalizację związaną z instrukcjami RET. Działanie również jest bardzo proste. Jedyną różnicą względem JUMP jest dodatkowe obsłużenie skoków typu JALR z rejestrem źródłowym równym x1. Kod implementacji zamieszczono na Rys. 4.22.

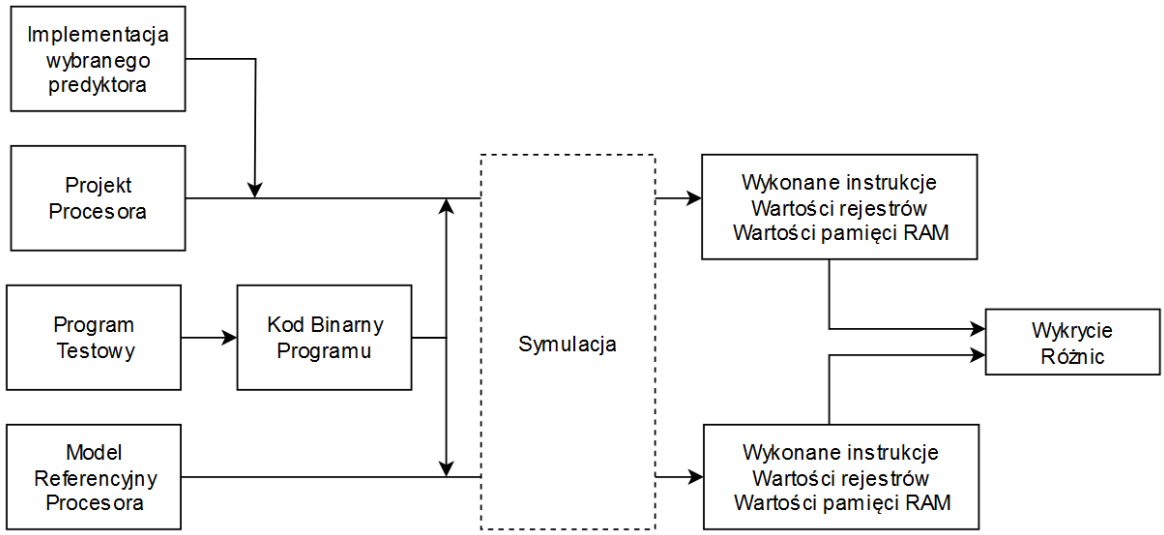
```
module riscv_next_strategy_compute_jump_ret # (
    parameter ADDR_WIDTH = 64,
    parameter INSTR_WIDTH = 64
) (
    input clk,
    input enable,
    input nreset,

    riscv_next_strategy_from_id_intf intf
);

    wire relative_jump_c = intf.i_id_signals.jal;
    wire absolute_jump_c = intf.i_id_signals.jalr && intf.i_id_signals.rs1_zero;
    wire return_jump_c   = intf.i_id_signals.jalr && intf.i_id_signals.rs1_ra;

    wire [ADDR_WIDTH-1:0] relative_jump_addr_c =
                                intf.i_id_signals.imm + intf.i_id_pc;
    wire [ADDR_WIDTH-1:0] absolute_jump_addr_c = intf.i_id_signals.imm;
    wire [ADDR_WIDTH-1:0] return_jump_addr_c   = intf.i_ra_data;

    assign intf.o_inject        =
                !intf.i_id_flush &&
                (relative_jump_c || absolute_jump_c || return_jump_c);
    assign intf.o_inject_addr   = relative_jump_c ? relative_jump_addr_c :
                                  absolute_jump_c ? absolute_jump_addr_c :
                                                    return_jump_addr_c;

endmodule
```

Rys.4.22. *Implementacja strategii JUMPRET*

5. Badania

5.1. Weryfikacja funkcjonalna

Niezbędnym elementem projektowania dowolnego modułu procesora jest zapewnienie jego poprawnego działania. W tym celu stworzono środowisko testowe, pozwalające na automatyczną weryfikację funkcjonalną mikroprocesora, której schemat przebiegu przedstawiono na Rys. 5.1. Jako symulator wybrano darmowego program Intel Questa-Intel FPGA Edition [15].



Rys.5.1. *Schemat przebiegu procesu weryfikacji*

Niezbędnym elementem weryfikacji, jest system ocenienia poprawności wyników działania systemu. Stworzony został więc model referencyjny mikroprocesora. Model ten wprowadza wiele uproszczeń w stosunku do oryginału. Zaimplementowany został stricte w celach symulacji, nie jest syntezowalny. Nie posiada żadnej formy potoku, ani nawet sygnału zegarowego. Na cały stan modelu składają się PC, rejestry, pamięć RAM, oraz stos RAS. Wraz z każdym wykonaniem kroku symulacji, zadana instrukcja RISC-V jest przetwarzana i stan modelu odpowiednio modyfikowany. Nie występuje tutaj żadne rozdzielenie na dekodowanie instrukcji na pojedyncze sygnały kontrolne, powiązane z poszczególnymi podmodułami, ale każdy rodzaj instrukcji przetwarzany jest całkowicie osobno, przez osobny algorytm dedykowany dla niej. Nie występują również żadne czasowe zależności poszczególnych elementów, ani nie ma potrzeby na przekierowywanie

danych na wzór modułu Forwarding Unit. Pozwala to minimalizację potencjalnych błędów w implementacji, i maksymalną zgodność ze specyfikacją RISC-V (nie licząc RAS i specjalnej obsługi JALR). Fragment modelu referencyjnego przedstawiono na Rys. 5.2.

```
task automatic STEP(output string parsed_instr); begin
    // if(!finished && en) begin
    if(!finished) begin
        is_load  = 0;
        is_store = 0;
        new_pc = pc + 4;
        temp_rs1 = REGS[RS1];
        temp_rs2 = REGS[RS2];
        casez (instr)
            LUI    : begin REGS[RD] = IMM_U;                                                                  end
            AUIPC  : begin REGS[RD] = pc  + IMM_U;                                                            end
            JAL    : begin new_pc = pc        + IMM_J; RAS_JUMP(pc, 0,   RD); REGS[RD] = pc + 4;      end
            JALR   : begin new_pc = REGS[RS1] + IMM_I; RAS_JUMP(pc, RS1, RD); REGS[RD] = pc + 4;      end
            BEQ    : begin new_pc = REGS[RS1] == REGS[RS2] ? pc + IMM_B : new_pc;                      end
            BNE    : begin new_pc = REGS[RS1] != REGS[RS2] ? pc + IMM_B : new_pc;                      end
            BLT    : begin new_pc = $signed(REGS[RS1]) <  $signed(REGS[RS2]) ? pc + IMM_B : new_pc;   end
            BGE    : begin new_pc = $signed(REGS[RS1]) >= $signed(REGS[RS2]) ? pc + IMM_B : new_pc;   end
            BLTU   : begin new_pc =          REGS[RS1]  <           REGS[RS2]  ? pc + IMM_B : new_pc;   end
            BGEU   : begin new_pc =          REGS[RS1]  >=          REGS[RS2]  ? pc + IMM_B : new_pc;   end
            LB     : begin REGS[RD] = $signed(LOAD_MEM(LD_ADDR, 1)); is_load=1;                         end
            LH     : begin REGS[RD] = $signed(LOAD_MEM(LD_ADDR, 2)); is_load=1;                         end
            LW     : begin REGS[RD] = $signed(LOAD_MEM(LD_ADDR, 4)); is_load=1;                         end
            LBU    : begin REGS[RD] =         LOAD_MEM(LD_ADDR, 1);  is_load=1;                         end
            LHU    : begin REGS[RD] =         LOAD_MEM(LD_ADDR, 2);  is_load=1;                         end
            LWU    : begin REGS[RD] =         LOAD_MEM(LD_ADDR, 4);  is_load=1;                         end
            LD     : begin REGS[RD] =         LOAD_MEM(LD_ADDR, 8);  is_load=1;                         end
            SB     : begin STORE_MEM(ST_ADDR, 1, REGS[RS2]);         is_store=1;                         end
            SH     : begin STORE_MEM(ST_ADDR, 2, REGS[RS2]);         is_store=1;                         end
            SW     : begin STORE_MEM(ST_ADDR, 4, REGS[RS2]);         is_store=1;                         end
            SD     : begin STORE_MEM(ST_ADDR, 8, REGS[RS2]);         is_store=1;                         end
            ADDI   : begin REGS[RD] = REGS[RS1] +   IMM_I;                                                end
            SLTI   : begin REGS[RD] = REGS[RS1] <   $signed(IMM_I);                                       end
            SLTIU  : begin REGS[RD] = REGS[RS1] <   IMM_I;                                                end
            XORI   : begin REGS[RD] = REGS[RS1] ^   IMM_I;                                                end
            // ...
```

Rys.5.2. *Fragment implementacji modelu referencyjnego*

Samo stworzenie modelu referencyjnego nie pozwoli na pełną weryfikację funkcjonalną. Mikroprocesor musi oczywiście wykonać jakiś program, aby móc zaobserwować efekty jego działania. W tym celu stworzono łącznie 64 programy testowe, z których każdy odpowiada za sprawdzenie konkretnej funkcjonalności mikroprocesora. Różne programy badają między innymi – poprawność działania instrukcji skoków warunkowych, wywoływania funkcji zagnieżdżonych z optymalizacją RAS, wykonywania skoków podczas oczekiwania na pamięć RAM, działania modułu Forwarding Unit, działania modułu Hazards Detection Unit podczas instrukcji interakcji z pamięcią. Wiele testów zostało zaprojektowanych z myślą o predyktorze ze strategią Hardcoded. Możliwość wymuszenia konkretnej predykcji w konkretnym miejscu pozwoliła sprawdzić poprawność zaimplementowanego modułu predyktora w dowolnym kontekście. Programy zostały stworzone w asemblerze RISC-V. Przykładowy program przedstawiono na Rys. 5.3.

```
BEQ     zero,   zero,   L1  ; Skok

ADDI    x31,    x31,    0x10   ; nieosiągalne
ADDI    x31,    x31,    0x20   ; nieosiągalne
ADDI    x31,    x31,    0x40   ; nieosiągalne
ADDI    x31,    x31,    0x80   ; nieosiągalne

L0:
BLT     zero,   zero,   L2  ; Brak Skoku; predykcja hardcoded 20 -> 44
```

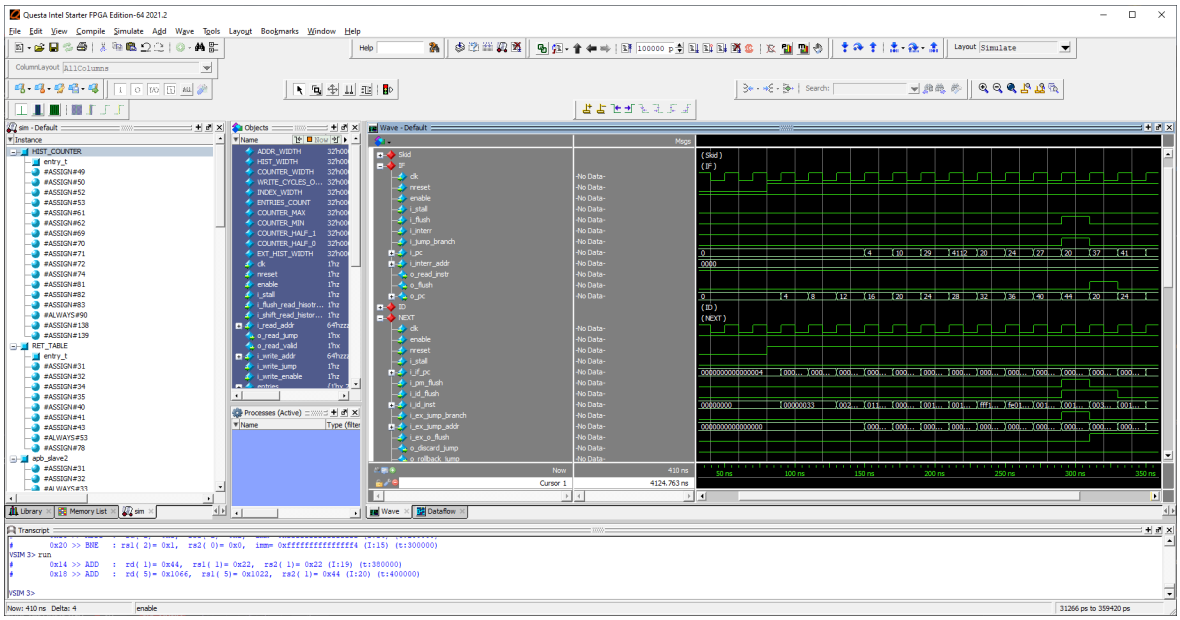
```
ADDI    x1,     x1,    0x010
XOR     zero,    zero,    zero  ; HALT
L1:
ADDI    x1,     x1,    0x01
BEQ     zero,   zero,   L0  ; Skok
ADDI    x31,    x31,    2   ; nieosiągalne

L2: ; addr = 44
ADDI    x31,    x31,    4   ; nieosiągalne
ADDI    x31,    x31,    8   ; nieosiągalne
XOR     zero,    zero,    zero  ; HALT
```

Rys.5.3. *Program testowy sprawdzający poprawność powrotu do prawidłowego stanu po wykonaniu niepoprawnej predykcji.*

W celach łatwiejszego testowania programów, dodano do środowiska symulacyjnego 2 dodatkowe założenia. Po pierwsze, każde nadpisanie wartości rejestru x31 traktowane jest jako wykonanie nieprawidłowej instrukcji. Pozwala to łatwo zorientować się, w którym miejscu mikroprocesor błędnie wykonał skok. Po drugie, przyjęto specjalną interpretację instrukcji XOR ze wszystkimi argumentami równymi rejestrowi x0. Mikroprocesor sam w sobie nie posiada żadnego mechanizmu zatrzymania swojego działania. Domyślnie więc, symulacja działałaby w nieskończoność. Postanowiono wybrać jakąś instrukcję, nie zwracającą efektów ubocznych, która jeśli podczas symulacji przetworzona zostanie przez procesor, zwróci sygnał zakończenia symulacji.

Symulacja przeprowadzana jest jednocześnie dla modelu referencyjnego oraz modelu badanego, z zaimplementowanym wybranym predyktorem. Wynikami symulacji są lista instrukcji wykonanych przez każdy model, oraz końcowe wartości rejestrów oraz pamięci RAM. Wszystkie różnice raportowane są w terminalu symulatora. Przygotowano skrypt, uruchamiający po kolei wszystkie testy dla wybranych kombinacji predyktorów.

Weryfikacja pozwala na określenie, czy błąd wystąpił, i w którym momencie, lecz nie pozwala na wykrycie jego przyczyny. Wymaga to poznania wartości wewnętrznych sygnałów mikroprocesora i ich zmian w czasie. Jeśli zauważono, że dany test przeszedł niepoprawnie, symulacja może zostać uruchomiona w trybie *gui,* jak pokazano na Rys. 5.4. Pozwala to na wyświetlenie przebiegu wszystkich ważnych sygnałów na interfejsie graficznym, oraz wyznaczenie wszystkich źródeł dowolnego sygnału, co znacznie ułatwia ustalenie powodu niepoprawnego działania systemu.



Rys.5.4. *Przykładowy widok przebiegu symulacji w trybie gui*

Uznano jednak, że choć Questa w trybie *gui* zwraca praktycznie wszystkie potrzebne dane, to nie są one wystarczająco czytelne. Szczególnie trudno jest analizować zależności między różnymi fazami potoku. W tym celu, stworzono alternatywny system analizy przebiegu sygnałów wewnątrz rdzenia. Wynikiem symulacji jest plik CSV (Comma Seperated Values), w którym zapisane są wartości wszystkich wybranych uprzednio sygnałów, w każdym takcie sygnału zegarowego. Plik CSV jest następnie przetwarzany przez stworzony skrypt, generujący czytelniejszą wizualizację poszczególnych modułów procesora. Wizualizacja odbywa się za pomocą języka HTML. Przykład wizualizacji przedstawiono na Rys. 5.5.

Rys.5.5. *Przykładowa wizualizacja za pomocą HTML*

Wizualizacja w HTML pozwala na łatwiejsze zgrupowanie zależnych sygnałów względem potoku, jak i bardziej zaawansowane formatowanie ich wartości. Przykładowo,

w komórkach odpowiadających fazie Program Memory można w czytelny sposób zaobserwować wczytywaną instrukcję, a nie jedynie wartości poszczególnych bitów. Dodatkowo możliwy jest podgląd aktualnego stanu rejestrów, pamięci RAM oraz RAS.

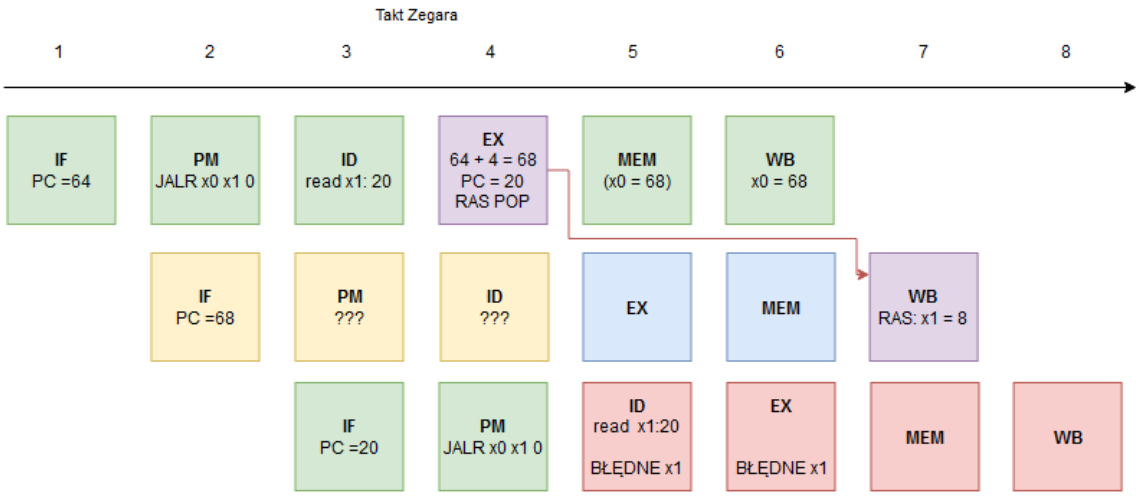
Podczas testów wykryto wiele błędów samego rdzenia mikroprocesora, bez wprowadzania modyfikacji związanych z predyktorem. Przykładowo, warunek wykonania skoku warunkowego BGE był błędny, moduł Forwarding Unit błędnie działał w relacji z oczekiwaniem na pamięć RAM. Z pomocą wszystkich zastosowanych mechanizmów wizualizacji wewnętrznego stanu procesora doprowadzono mikroprocesor do pełnego stanu używalności.

Niestety, podczas weryfikacji predyktorów wykryto jeden poważny błąd. Dotyczy on przewidywania skoków JALR będących powrotami z funkcji. Na Rys. 5.6. przedstawiony został przykładowy program wykorzystujący zagnieżdżone funkcje.

```
    ADDI   x0,    x0,    0 ; NOOP
    JAL    x1,    L1
    ADDI   a2,    a1,    1
    XOR    x0,    x0,    x0 ; HALT

L1:
    JAL    x1,    L2
    JALR   x0,    x1,    0 ; RET

L2:
    JAL    x1,    L3
    JALR   x0,    x1,    0 ; RET

L3:
    JAL    x1,    L4
    JALR   x0,    x1,    0 ; RET

L4:
    JAL    x1,    L5
    JALR   x0,    x1,    0 ; RET

L5:
    ADDI   a1,    a1,    5
    JALR   x0,    x1,    0 ; RET
```

Rys.5.6. *Przykładowy program zawierający zagnieżdżone funkcje*

W przypadku braku predyktora, wykonanie programu przebiega poprawnie. Jeśli jednak zastosowany zostanie predyktor, np. ze strategią JUMPRET, to wykonywane są błędne instrukcje. Na Rys. 5.7. przedstawiono fragment przebiegu działania instrukcji JALR, analogicznie jak na Rys. 3.10 (w rozdziale 3).



Rys.5.7. *Przebieg wykonania instrukcji JALR przy zastosowaniu strategii JUMPRET*

Aktualizacja wartości rejestru x1 za pomocą RAS jest opóźniona o 3 takty zegara względem wykonania skoku. Wcześniej nie sprawiało to żadnych problemów, ponieważ 3 fazy potoku musiały zostać na skutek skoku wyzerowane. Jeśli jednak poprawnie przewidziano skok, jak na Rys. 5.7, kolejna instrukcja może zacząć wykonywać się wcześniej. Jeśli jako rejestr źródłowy wykorzystuje x1 (jak np. instrukcja JALR), odczytana zostanie błędna wartość. Naprawa tego błędu wymagałaby znacznej ingerencji w strukturę procesora, lub dodanie modułu dodatkowo opóźniającego pracę procesora w przypadku, kiedy wykryto taką zależność. Programy kompatybilne ze standardem RISC-V nie przewidują optymalizacji związanej z RAS, a więc błąd ten na nie nie wpływa. Chcąc skorzystać z RAS, programista musi stworzyć program dedykowany pod analizowany procesor. Można więc wymagać od niego stosowania się do konkretnych ograniczeń. Jeśli bowiem programista zapewni odstęp przynajmniej 3 instrukcji pomiędzy kolejnymi powrotami z funkcji, opisany problem nie wystąpi – RAS zdąży poprawnie zaktualizować rejestr x1. Z tego powodu postanowiono nie przeprowadzać naprawy tego błędu.

5.2. Badanie efektywności predyktorów

Podczas symulacji, generowany jest plik CSV, który zawiera wszystkie ważne sygnały związane ze skokami oraz predyktorem. Składają się na nie sygnały wyjściowe fazy Execute i sygnały interfejsu predyktora. Dodatkowo, zapisywane są też wartości przydatnych liczników, które występują jedynie w symulacji. Liczniki te, nazwane Evict Counter, reprezentują to, ile razy wewnątrz modułów BTB, BHT oraz RAS_TABLE wystąpiło nadpisanie istniejącego rekordu nowym. Pozwolą one ułatwić określenie, jak rozmiar tych buforów wpływa na efektywność predykcji skoków.

W celach przeprowadzenia oceny efektywności predyktorów, niezbędny jest program, który zostanie wykonany podczas symulacji. Postanowiono stworzyć wariację popularnego prostego programu, nazywanego FizzBuzz. Nazwa wywodzi się od prostej zabawy dziecięcej, polegającej na liczeniu kolejnych liczb, ale zamiast wielokrotności liczby 3, należy wykrzyknąć „Fizz”, a zamiast wielokrotności 5, wykrzyknąć „Buzz”. Jeśli liczba jest jednocześnie wielokrotnością 3 i 5, należy powiedzieć „FizzBuzz”. Kod programu stworzono w języku C, i przedstawiono go na Rys. 5.8 .

```
#define MEM_D(addr) (*((volatile long*)(addr)))

static long do_fizz(long i) {
    static long counter = 0;
    counter += 1;
    return i + counter;
}

static long do_buzz(long i) {
    static long counter = 0;
    counter += 1;
    return i * counter;
}

ENTRY() {
    __asm__("nop\nli sp, 1000");
    const long limit = MEM_D(8);
    long mem_offset = 1024;
    int i_mod_3 = 1;
    int i_mod_5 = 1;
    for(long i = 1; i <= limit; i++) {
        if(i_mod_3 == 0 && i_mod_5 == 0) {
            MEM_D(mem_offset) = do_fizz(i);
        } else if (i_mod_3 == 0) {
            MEM_D(mem_offset) = do_buzz(i);
        } else if (i_mod_5 == 0) {
            MEM_D(mem_offset) = do_fizz(i) ^ do_buzz(i);
        } else {
            MEM_D(mem_offset) = i;
        }
        mem_offset += 8;
        i_mod_3 = i_mod_3 == 2 ? 0 : i_mod_3 + 1;
        i_mod_5 = i_mod_5 == 4 ? 0 : i_mod_5 + 1;
    }
    __asm__("xor x0,x0,x0");
}
```

Rys.5.8. *Kod programu FizzBuzz*

Program wykorzystuje specjalne makra i wstawki asemblerowe, które umożliwiają uruchomienie go na analizowanym mikroprocesorze. Mikroprocesor nie posiada wsparcia funkcjonalności niezbędnych do uruchomienia pełnoprawnego systemu operacyjnego, dlatego wymaga, między innymi, ręcznej inicjalizacji niektórych rejestrów. Stworzenie i kompilacja pliku w języku C pozwala jednak lepiej odwzorować faktyczny rodzaj programu, jaki w praktyce może być wykonywany na procesorze, niż jeśli miałoby się program stworzyć w samym asemblerze.

Głównym elementem programu jest pętla *for*. Liczba iteracji, jakie pętla ma wykonać, wczytywana jest z pamięci RAM przy starcie programu. Jest to w każdym

przeprowadzanym teście wartość 100. W każdej iteracji, na podstawie wartości licznika, podejmowana jest decyzja, jaką funkcję należy wykonać. Procesor nie posiada możliwości wypisywania wartości na ekranie, więc nie można bezpośrednio zasymulować gry FizzBuzz. Zamiast tego, program zapisuje odpowiednio przetworzone wartości na kolejne adresy pamięci RAM. W programie zdefiniowano 2 funkcje: *do_fizz* oraz *do_buzz*. Ta druga wykorzystuje instrukcję mnożenia, która zostanie przez kompilator zaimplementowana jako jeszcze jedna funkcja, która wykonywana będzie rekurencyjnie. Jeśli aktualna wartość iteratora jest wielokrotnością 3 lub 5, wykonywane są odpowiednie funkcje, i ich wynik zapisywany jest w pamięci RAM. Jeśli licznik nie jest wielokrotnością, do pamięci RAM po prostu zapisywany jest sam licznik. Taki program pozwala na przetestowanie wpływu predyktorów skoków na zwykłe konstrukcje *if...else*, na pętle oraz na wywoływania funkcji.

Do wyznaczenia efektywności predykcji mierzone są 2 główne wartości. Pierwszą jest liczba instrukcji wyzerowanych podczas przetwarzania przez potok w stosunku do łącznej liczby wczytanych instrukcji. Wartość ta oznaczona jest nazwą FLUSHES. Drugą jest stosunek niepoprawnie przewidzianych instrukcji skoku, w stosunku do wszystkich przetworzonych instrukcji skoku, Wartość ta podzielona jest ze względu na rodzaj skoku (JAL, JALR i BR), i oznaczona jest poprzez MISSES. Podczas badań, jedynie predykcje PM Strategy są uznawane jako poprawne. Jeśli to ID Strategy dokona predykcji, nadal tracony jest co najmniej jeden takt zegara. Łączną wartość MISSES rozdzielono więc na sumę dwóch składników - pierwszy reprezentuje skoki błędnie przewidziane przez dowolną strategię, a drugi poprawnie przewidziane przez ID Strategy. Wartość FLUSHES stanowi bezpośredni wyznacznik efektywności mikroprocesora – jako de facto stosunek czasu spędzonego bezczynnie do łącznego czasu wykonania programu (bez uwzględnienia czekania na pamięć RAM). Wskaźniki MISSES pozwalają za to precyzyjniej określić konkretne sytuacje, w których dany predyktor się sprawdza.

5.2.1. ID Strategy

Jako pierwsze zmierzony zostanie wpływ zastosowania ID Strategy. Porównane zostanie wykonanie programu FizzBuzz bez żadnego predyktora, z zaimplementowaną strategią JUMP oraz ze strategią JUMPRET. Wyniki symulacji przedstawiono w Tab. 5.1.

Tab.5.1. *Efektywność ID Strategy*

ID Strategy	FLUSHES	MISSES JAL	MISSES JALR	MISSES BR
Brak	40.3% (3240/8049)	100%+0% (297+0/297)	100%+0% (102+0/102)	63.3%+0% (681+0/1075)
JUMP	35.5% (2646/7455)	0%+100% (0+297/297)	100%+0% (102+0/102)	63.3%+0% (681+0/1075)
JUMPRET	33.7% (2442/7251)	0%+100% (0+297/297)	0%+100% (0+102/102)	63.3%+0% (681+0/1075)

Jak widać, program bez zastosowania żadnego predyktora potrzebował do wykonania się 8049 taktów zegara. 3240 z tych taktów zostało „zmarnowanych", ze względu na zerowanie faz procesora przy skokach. Żaden skok JAL ani JALR nie został poprawnie przewidziany. Widać jednak, że tylko 63% skoków typu BR zostało błędnie przewidzianych. Żeby to wyjaśnić, należy sobie przypomnieć, że w przypadku analizowanego mikroprocesora, brak predyktora jest tożsamy z przewidywaniem zawsze skoku niewykonanego. Oznacza to, że 36% skoków warunkowych było niewykonanych, co zostało poprawnie przewidziane.

Patrząc na kolumnę MISSES JAL dla strategi JUMP, widać, że wszystkie wykonane skoki JAL zostały poprawnie przewidziane w ID Strategy. Ponieważ każdy skok przewidziany przez ID Strategy daje zysk 2 taktów zegara względem niepoprawnej predykcji. Ponieważ liczba przetworzonych instrukcji skoku wynosi 297, łączna liczba FLUSHES powinna zmaleć z poziomu 3240 o dwukrotność tej wartości, czyli o 594, czyli do 2646, co jest zgodne z obserwacjami.

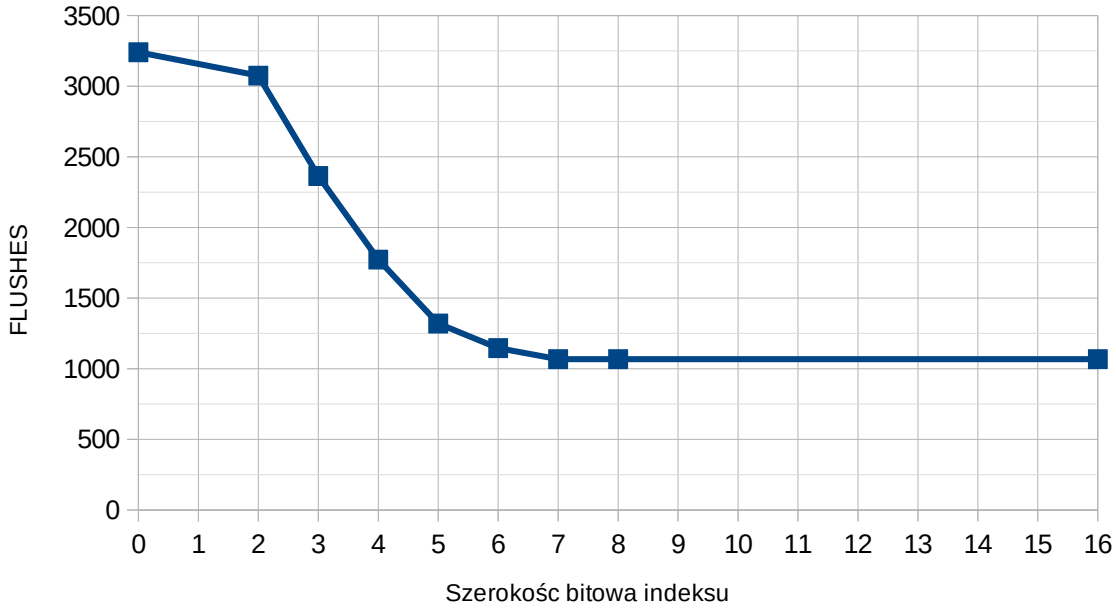
Zastosowanie strategi JUMPRET dalej polepsza sytuację, poprawnie przewidując wszystkie instrukcje JALR, (czyli powroty z funkcji). Trzeba jednak zauważyć, że żadna strategia nie ma żadnego wpływu na efektywność przewidywania skoków BR. Skoki warunkowe stanowią przeważającą większość wszystkich przetwarzanych skoków, dlatego niezbędne jest zastosowanie bardziej zaawansowanego mechanizmu predykcji.

Wszystkie kolejne testy przeprowadzone zostaną z wyłączoną ID Strategy. Pozwoli to lepiej zaobserwować działanie samych algorytmów PM Strategy.

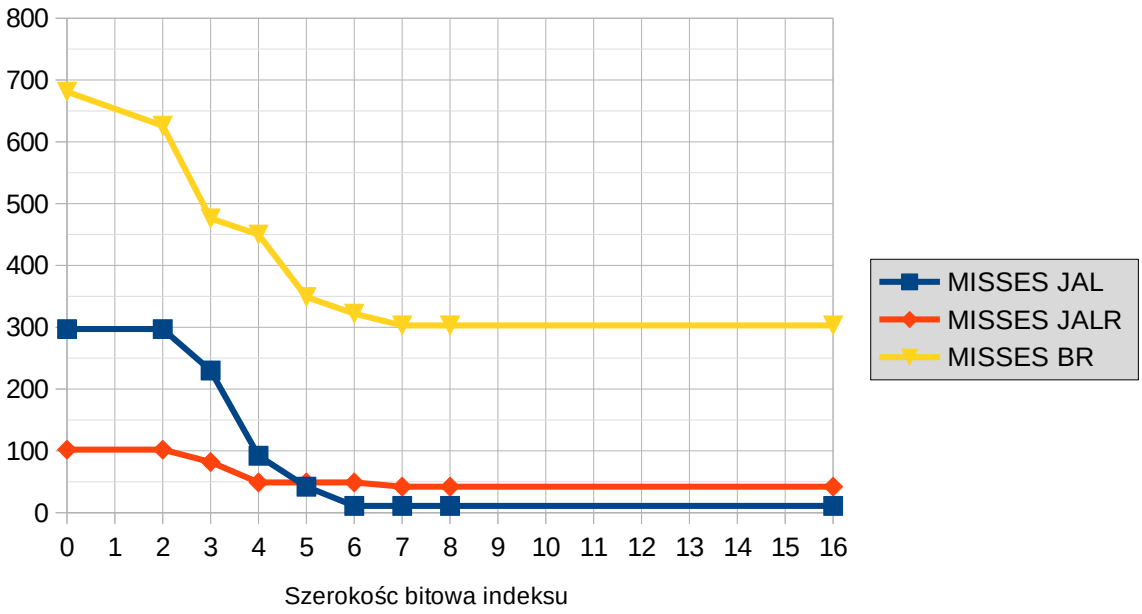
5.2.2. PM Strategy – BTB

Strategia BTB jest parametryzowalna pod 2 względami – szerokości bitowej indeksu, oraz szerokości bitowej licznika nasycenia. Zbadano wpływ każdego z tych parametrów na jakość predykcji. Na wykresie Rys. 5.9. przedstawiono wskaźnik FLUSHES zależnie od rozmiaru indeksu, a na Rys. 5.10. poszczególne wskaźniki MISSES. Dodatkowo, zarejestrowano wartość Evict Counter'a, który przedstawiono jako wskaźnik jakości o

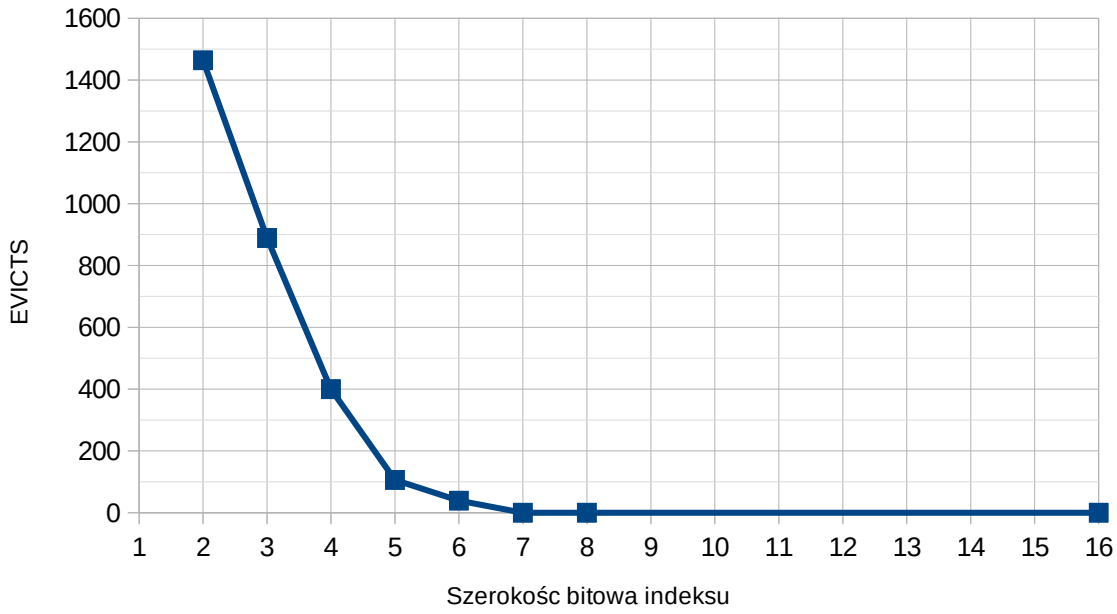
nazwie EVICTS, na Rys. 5.11. Dla punktów na wykresach dla szerokości bitowej indeksu równej zero żaden predyktor nie jest włączony. Jako rozmiar liczników nasycenia przyjęto 2 bity.



Rys.5.9. *Zależność FLUSHES od rozmiaru indeksu BTB*



Rys.5.10 *Zależność MISSES od rozmiaru indeksu BTB*



Rys.5.11. *Zależność EVICTS od rozmiaru indeksu BTB*

Jak widać, na podstawie wskaźnika FLUSHES, zwiększanie rozmiaru indeksu BTB polepsza jakość predykcji. Najszybszy spadek można zaobserwować dla wielkości mniejszych niż 5. Dla rozmiarów 5, 6 oraz 7 różnica jest znikoma, po czym dla każdej zbadanej większej szerokości indeksu wartość ta się nie zmienia. Bezpośrednim powodem kształtu tej tendencji są nadpisania istniejących rekordów wewnątrz modułu BTB. Patrząc na wskaźnik EVICTS można zaobserwować podobny kształt, lecz znacznie gładszy. Każde zwiększenie rozmiaru indeksu o 1 powinno spowodować zwiększenie rozmiaru całego BTB dwukrotnie. Szansa na kolizję adresów powinna więc maleć około dwukrotnie, aż do osiągnięcia 0 kolizji. Kształt wykresu wskaźnika EVICTS jest zgodny z oczekiwaniami.

Wskaźnik MISSES JAL ma monotoniczny wykres, analogicznie do wskaźnika EVICTS. Trzeba zauważyć, że nigdy nie osiąga on 0. Każdy skok w programie musi się wykonać (i zostać nieprzewidzianym) chociaż raz, aby móc w ogóle otrzymać swój rekord w BTB. Ponieważ dla rozmiarów indeksu 8 (i więcej) wartość EVICTS wynosi 0, żaden skok JAL nie został nadpisany przez inny. Wskaźnik MISSES JAL może więc minimalnie osiągnąć wartość równą unikalnym instrukcją skoku JAL w wykonanym programie, co też się stało. Wartość ta wynosi 11.

Podobnie jest z instrukcjami JALR. W programie występują jednak jedynie 3 funkcje – *do_fuzz*, *do_buzz* oraz automatycznie wygenerowana funkcja mnożenia. Dlaczego więc wartość MISSES JALR nie spada do wartości 3, a jedynie do 42? Funkcje mogą bowiem być wywoływane z różnych miejsc. Np. funkcja *do_fuzz* jest wywoływana zarówno w przypadku licznika podzielnego przez 3, jak i podzielnego przez 15. Adres powrotny dla każdego tego przypadku jest różny. Za każdym razem, kiedy funkcja zostanie więc

wywołana z innego miejsca w programie, adres powrotny się zmienia, więc rekord w BTB musi zostać nadpisany.

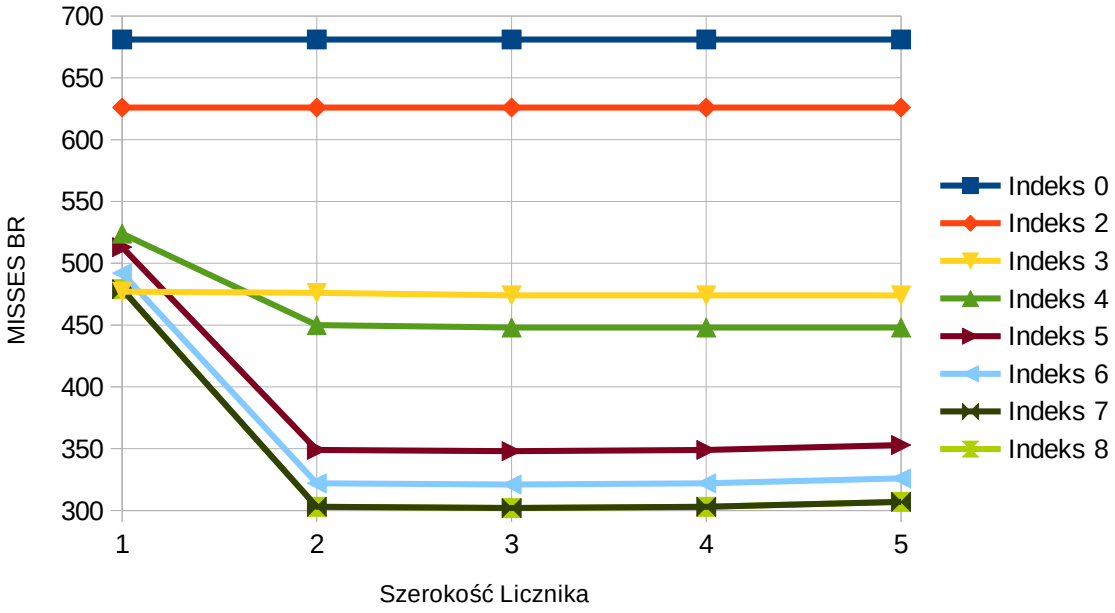
W odróżnieniu od samej ID Strategy, skoki warunkowe w PM Strategy mogą być przewidywane. Jak widać, nawet dla szerokości indeksów równych 2, wskaźnik MISSES BR jest mniejszy, niż przy braku predyktora. Wskaźniki MISSES JAL oraz MISSES JALR wymagają co najmniej 3 bitów indeksu by wykazać poprawę. Cecha ta wynika najprawdopodobniej z tego, że w programie została wygenerowana mała pętla (np. wewnątrz funkcji mnożącej), przez co skoki warunkowe mogą zdążyć się „nauczyć", bez szansy na nadpisanie ich rekordów w międzyczasie innym skokiem.

Zestawienie konkretnych wartości wszystkich wskaźników zamieszczono na Tab. 5.2.

Tab.5.2 *Zależność efektywności predyktora od rozmiaru indeksu BTB*

Rozmiar Indeksu	FLUSHES	MISSES JAL	MISSES JALR	MISSES BR	EVICTS
Brak BTB	40.3% (3240/8049)	100%+0% (297+0/297)	100%+0% (102+0/102)	63.3%+0% (681+0/1075)	0
2	39% (3075/7884)	100%+0% (297+0/297)	100%+0% (102+0/102)	58.2%+0% (626+0/1075)	1464+0+0
3	33% (2364/7173)	77.4%+0% (230+0/297)	80.4%+0% (82+0/102)	44.3%+0% (476+0/1075)	889+0+0
4	26.9% (1773/6582)	31%+0% (92+0/297)	48%+0% (49+0/102)	41.9%+0% (450+0/1075)	400+0+0
5	21.5% (1320/6129)	14.1%+0% (42+0/297)	48%+0% (49+0/102)	32.5%+0% (349+0/1075)	106+0+0
6	19.2% (1146/5955)	3.7%+0% (11+0/297)	48%+0% (49+0/102)	30%+0% (322+0/1075)	39+0+0
7	18.2% (1068/5877)	3.7%+0% (11+0/297)	41.2%+0% (42+0/102)	28.2%+0% (303+0/1075)	0+0+0
8	18.2% (1068/5877)	3.7%+0% (11+0/297)	41.2%+0% (42+0/102)	28.2%+0% (303+0/1075)	0+0+0
16	18.2% (1068/5877)	3.7%+0% (11+0/297)	41.2%+0% (42+0/102)	28.2%+0% (303+0/1075)	0+0+0

Przeprowadzono dodatkowo badanie wpływu rozmiaru licznika nasycenia. Jedyne wskaźniki, na jaki zmiana licznika wpłynęła, to FLUSHES i MISSES BR. Jest to oczekiwane zachowanie, ponieważ licznik powinien wpływać jedynie na skoki warunkowe. Postanowiono więc porównać jedynie wskaźniki MISSES BR, co przedstawiono na wykresie Rys. 5.12, oraz w tabeli Tab. 5.3.



Rys.5.12. *Zależność MISSES BR od rozmiaru indeksu oraz licznika*

Tab.5.3. *Zależność MISSES BR od rozmiaru indeksu oraz licznika*

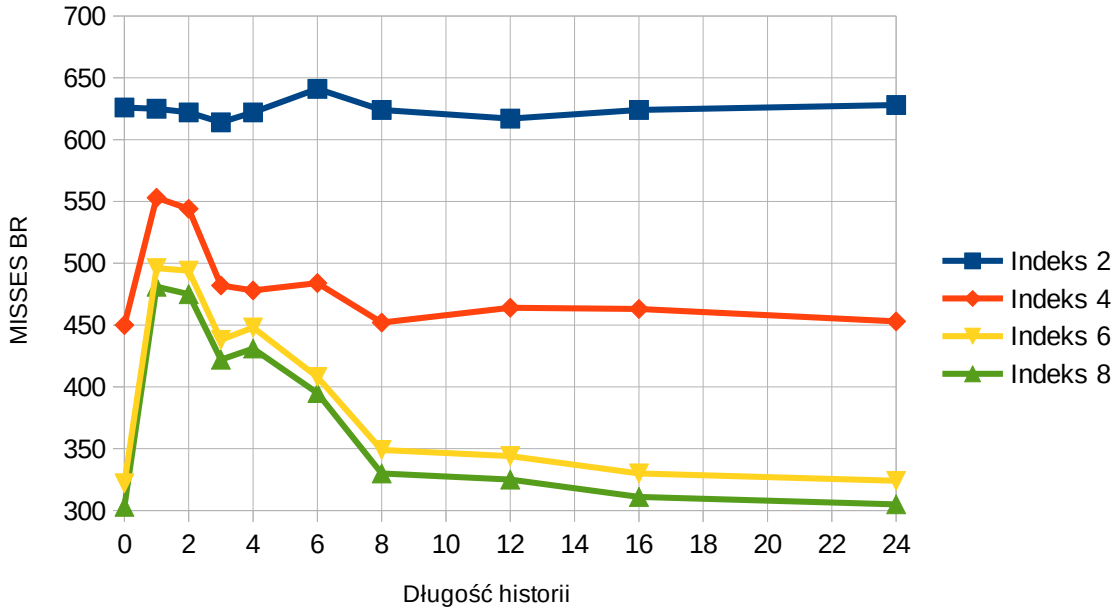
Rozmiar Licznika	**Indeks 0**	**Indeks 2**	**Indeks 3**	**Indeks 4**	**Indeks 5**	**Indeks 6**	**Indeks 7**	**Indeks 8**
1	681	626	477	524	513	492	479	479
2	681	626	476	450	349	322	303	303
3	681	626	474	448	348	321	302	302
4	681	626	474	448	349	322	303	303
5	681	626	474	448	353	326	307	307

Jedyny zauważalny wpływ szerokości liczników nasycenia na efektywność predykcji obserwujemy pomiędzy rozmiarami 1 i 2. Liczniki o rozmiarze 2 lub większym dają drastycznie lepsze efekty predykcji niż o rozmiarze 1, o ile rozmiar indeksu wynosił co najmniej 4 bity. Jeśli indeks był mniejszy, rozmiar licznika nie ma żadnego wpływu na jakość predykcji. Mała szerokość indeksu powoduje wysokie prawdopodobieństwo wystąpienia kolizji indeksów. Najwidoczniej, w badanym programie nie występuje ścieżka, której wynik predykcji mógłby się zmienić w czasie szybszym, niż po prostu wyrzucenie adekwatnego rekordu z BTB. Co ciekawe, w przypadku 1-bitowego licznika wskaźnik MISSES BR jest najmniejszy dla indeksu o rozmiarze 3 bitów. Nie można jednak liczyć na tą zależność w przypadku ogólnym. Prawdopodobnie jest to specyficzna cecha konkretnego testowanego programu. Zwiększanie rozmiaru licznika powyżej 2 bitów nie daje żadnych zauważalnych efektów. Wręcz lekko pogarsza jakość predykcji.

Oznacza to, że nie ma żadnego racjonalnego powodu, na implementację liczników o szerokości innej niż 2. Wszystkie przyszłe przeprowadzane testy przyjmą więc rozmiar stosowanych liczników nasycenia równy 2.

5.2.3. PM Strategy – BHT

Jakość predykcji powinna móc zostać dalej polepszona poprzez zastosowanie strategii BHT. Strategia ta działa identycznie jak BTB, za wyjątkiem skoków warunkowych. Ponownie więc jedyny ważny wskaźnik poddany badaniu to MISSES BR. Zaobserwowaną zależność MISSES BR od szerokości indeksu BTB oraz długości historii BHT przedstawiono na Rys. 5.13. oraz na Tab. 5.4. Wszystkie liczniki nasycenia są 2-bitowe.



Rys.5.13. *Zależność MISSES BR od długości historii BHT i szerokości indeksu BTB*

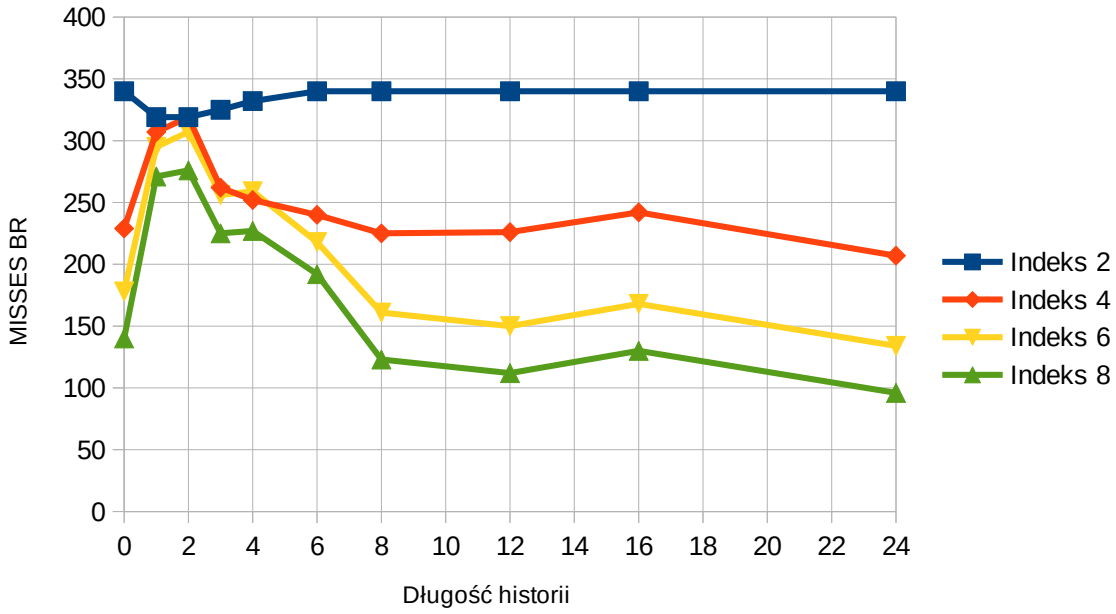
Tab.5.4. *Zależność MISSES BR od długości historii BHT i szerokości indeksu BTB*

	0	**1**	**2**	**3**	**4**	**6**	**8**	**12**	**16**	**24**
Indeks 2	626	625	622	614	622	641	624	617	624	628
Indeks 4	450	553	544	482	478	484	452	464	463	453
Indeks 6	322	496	494	438	448	408	349	344	330	324
Indeks 8	303	481	475	422	431	395	330	325	311	305

Wyniki okazały się być bardzo zaskakujące. Okazało się, że zastosowanie BHT jedynie pogorszyło jakoś predykcji, w stosunku do strategi BTB. Próbowano przetestować jeszcze

dłuższe historie, jednak symulator nie zezwalał na takie duże struktury danych (powyżej 2^{24} rekordów). Najwidoczniej, wybrany do testów program w ogóle nie współpracuje z BHT. Posiada skoki bardzo mało zależne od poprzednich, Głównym założeniem BHT jest wykrycie fragmentów programu, które powtarzają się często w taki sam sposób. Najwidoczniej program FizzBuzz w połączeniu z mnożeniem generuje wyjątkowo mało zależne od siebie ścieżki przebiegu, przez co BHT jedynie pogarsza rezultat.

Postanowiono zmodyfikować lekko program testowy. Funkcja *do_buzz* zamiast wykonywać mnożenie, będzie teraz wykonywać odejmowanie. Powtórzono eksperyment dla nowego programu, rezultaty zamieszczając na Rys. 5.14 oraz Tab. 5.5.



Rys.5.14. *Zależność MISSES BR od długości historii BHT i szerokości indeksu BTB, dla zmodyfikowanego FizzBuzz*

Tab.5.5. *Zależność MISSES BR od długości historii BHT i szerokości indeksu BTB, dla zmodyfikowanego FizzBuzz*

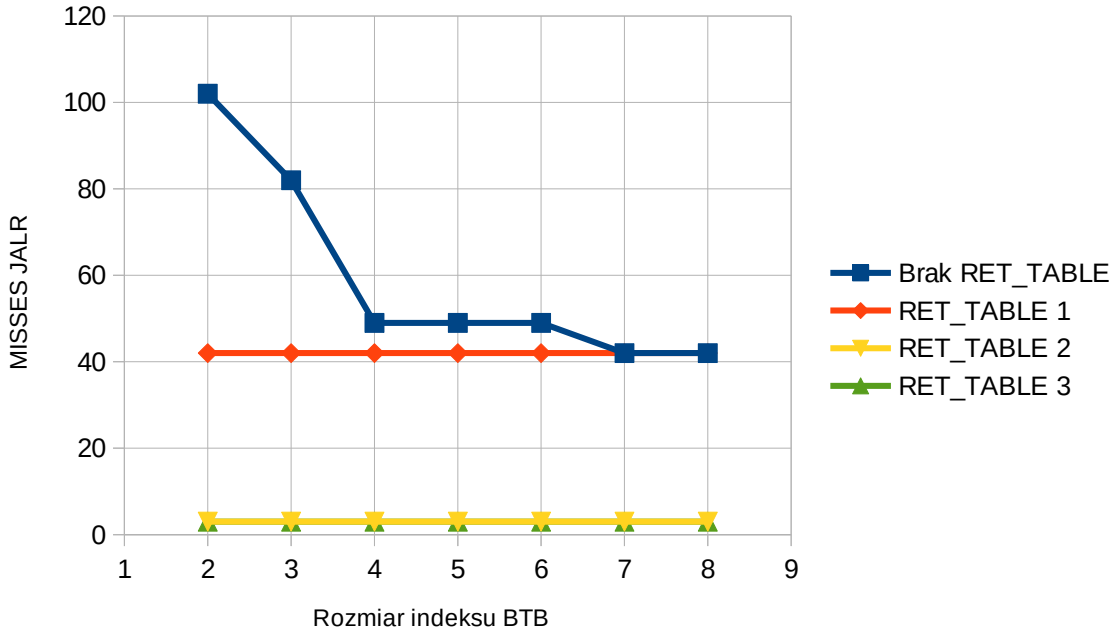
	0	**1**	**2**	**3**	**4**	**6**	**8**	**12**	**16**	**24**
Indeks 2	340	319	319	325	332	340	340	340	340	340
Indeks 4	229	307	319	262	252	240	225	226	242	207
Indeks 6	178	295	307	256	259	218	161	150	168	134
Indeks 8	140	271	276	225	227	192	123	112	130	96

Jak widać, dla zmodyfikowanego programu wyniki są dużo ciekawsze. Krótkie historie powodują pogorszenie efektywności, lecz przy historiach o długości co najmniej 8, wartość MISSES BR jest już lepsza, niż dla przypadku bez BHT. Zysk jest jednak wciąż znacznie mniejszy, niż się spodziewano.

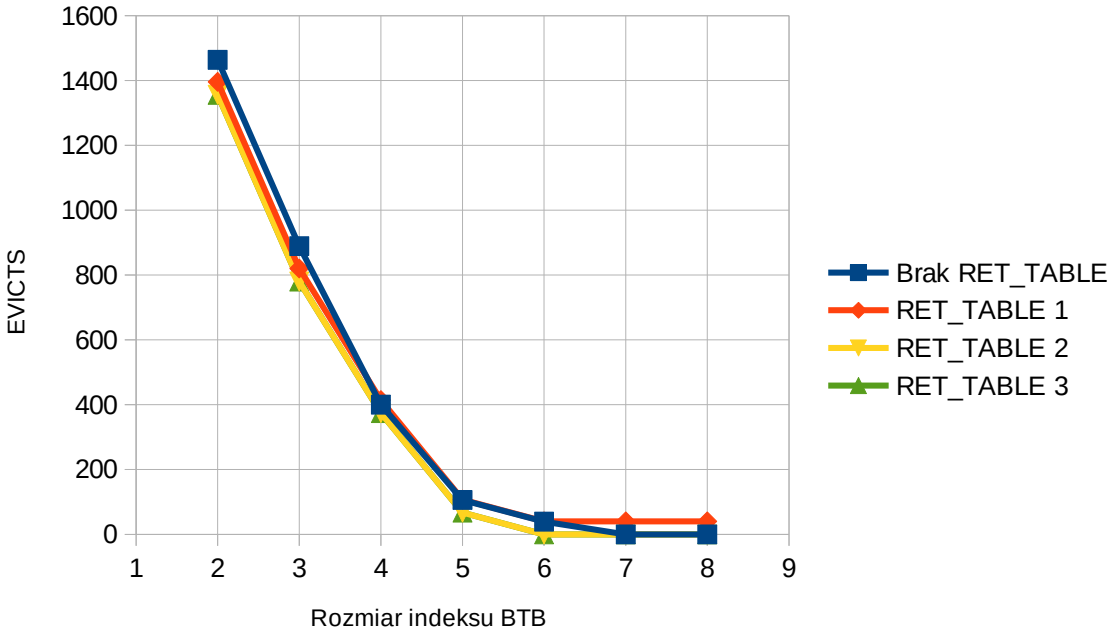
Warto zauważyć jednak, że wykresy nie są monotoniczne. Oznacza to, że niektóre długości historii powodują gorszą jakość predykcji, niż minimalnie mniejsze lub większe. Prawdopodobnie dlatego w mikroprocesorze SonicBOOM zastosowany został predyktor TAGE, który pozwala na wykonywanie predykcji na podstawie wielu historii, o różnych długościach. Takie działanie pozwoliłoby ominąć tego rodzaju pojedyncze piki niewydajności, niezależnie od wykonywanego programu.

5.2.4. PM Strategy – BHTRET

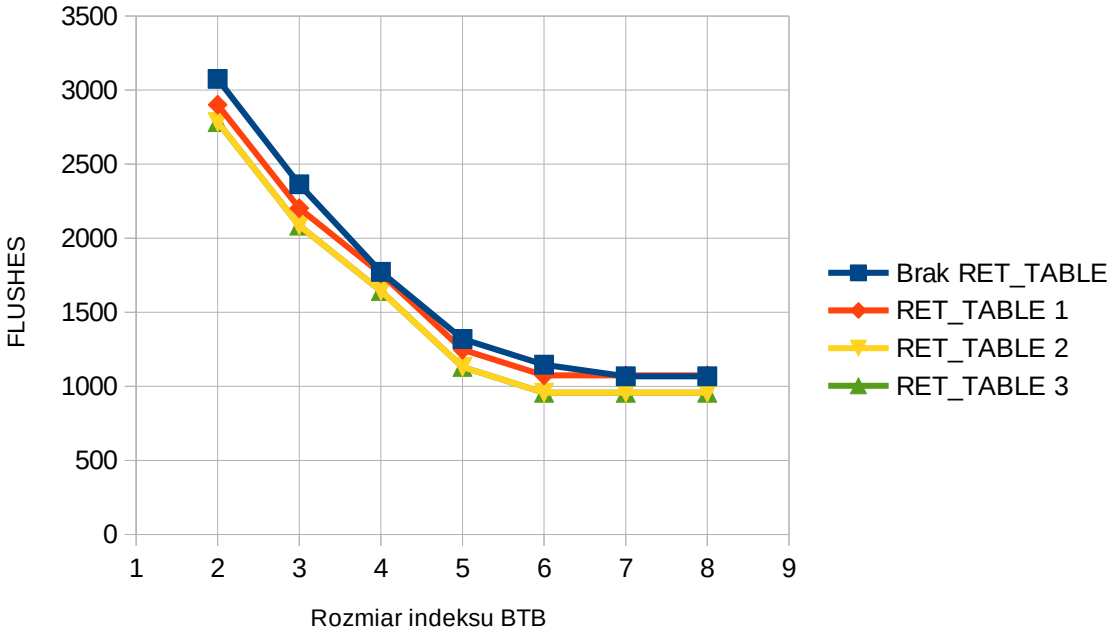
Do zbadania została jeszcze strategia BHTRET. Testy przeprowadzono dla niezmodyfikowanego programu, dla długości historii równej 24. Strategia BHTRET powinna wpłynąć głównie na MISSES JALR, oraz na EVICTS. Wyniki symulacji przedstawiono na Rys. 5.15, Rys. 5.16, Rys. 5.17.



Rys.5.15. *Zależność MISSES JALR od rozmiaru indeksu BTB oraz RET_TABLE*



Rys.5.16. *Zależność EVICTS od rozmiaru indeksu BTB oraz RET_TABLE*



Rys.5.17. *Zależność FLUSHES od rozmiaru indeksu BTB oraz RET_TABLE*

Jak widać, zastosowanie BHTRET zawsze polepsza jakość predykcji, w stosunku do zwykłego BTB. Testowany program posiadał jedynie 3 unikalne instrukcje JALR. RET_TABLE o indeksie o długości 1 bita może pomieścić maksymalnie 2 rekordy. Nieuniknionym jest więc wystąpienie nadpisań rekordów. Jak widać, wartość EVICTS jest w każdym mierzonym punkcie o 40 większa, niż w przypadku braku RET_TABLE. Zwiększenie rozmiaru indeksu do 2 bitów daje już szansę na uniknięcie jakichkolwiek kolizji. Tak też się dzieje, dzięki czemu wartość MISSES JALR ma minimalną możliwą

wartość 3, niezależnie od długości indeksu BTB. Ponieważ już dla 2-bitowego indeksu RET_TABLE nie występują kolizje, dalsze jego zwiększanie nie przyniesie żadnych zmian rezultatów dla badanego programu testowego.

6. Podsumowanie

Praca magisterska miała na celu zaprojektowanie, implementację i zbadanie efektywności modułu predykcji skoków dla mikroprocesora RISC opartego o specyfikację RISC-V. Autor otrzymał projekt mikroprocesora, który powstał jako efekt innych prac magisterskich. Otrzymany projekt okazał się zawierać błędy i w wielu aspektach był niezgodny z dostarczoną dokumentacją. Pierwszym zadaniem było więc wykrycie i naprawa obecnych w nim błędów. W celu przetestowania zaimplementowanych modułów został stworzony uproszczony, między innymi o przetwarzanie potokowe, model referencyjny mikroprocesora oraz zostało opracowane rozbudowane środowisko testowe. Istotnym, bardzo przydatnym aspektem było dodanie mechanizmu pozwalającego na podgląd sygnałów wybranych modułów mikroprocesora. Stworzono również wiele programów testowych. Środowisko pozwalało na wykonanie dowolnego testu i uruchomienie symulacji, dla dowolnego programu testowego, przy dowolnej konfiguracji projektu mikroprocesora, przy użyciu jednej komendy, bez potrzeby na modyfikację jakiegokolwiek pliku źródłowego i ręcznego uruchamiania kolejnych pomniejszych skryptów. Pozwala to na stwierdzenie poprawności i poziomu efektywności wprowadzonych zmian w projekcie mikroprocesora – zarówno tych usuwających wcześniejsze błędy, jak i tych kluczowych, związanych z zaimplementowaniem modułu predykcji skoków.

Postanowiono zaimplementować kilka różnych, konfigurowalnych wersji predyktora skoków. Każda z nich charakteryzować się może innymi wadami i zaletami, które ujawniają się w różnych sytuacjach. Zaimplementowano 2 główne rodzaje strategii predyktora – PM Strategy (mniej dokładna, ale szybciej reagująca) oraz ID Strategy (dokładniejsza, ale wolniej reagująca). Przeprowadzono rozbudowane porównanie wariantów modułu predykcji na podstawie eksperymentów symulacyjnych. Spośród wszystkich 5 testowanych algorytmów predykcji, większość z nich dała efekty zgodne z oczekiwaniami. Wykazano optymalną szerokość bitową liczników nasycenia, która wyniosła 2 bity. Najciekawsze wyniki badań dała analiza strategii BHT. Okazało się, że wprowadzenie predykcji zależnej od historii skoków warunkowych daje znacznie gorszą efektywność predyktora, niż wykorzystanie jedynie adresu skoków jako indeksu. Jedynie w specyficznych, uproszczonych przypadkach, przy zastosowaniu odpowiednio długiej historii skoków, można zaobserwować polepszenie jakości predykcji przez zastosowanie BHT. Potwierdzono również poprawność stworzonego systemu predykcji instrukcji JALR, będących powrotami z funkcji. Wyznaczanie adresu docelowego jedynie na podstawie rejestru adresu powrotu, bez zastosowania stosu RAS, pozwoliło na w pełni poprawną

predykcję tego typu skoków, bez potrzeby implementacji dodatkowej struktury stosu, jak miało to miejsce w już istniejących procesorach..

Projekt predyktora wykonano w sposób modularny, co oznacza, że w razie konieczności zmiany sposobu działania algorytmu predykcji potrzebna będzie minimalna modyfikacja projektu, jedynie w części odpowiedzialnej stricte za sam algorytm. Ponadto, projekt wykonano w taki sposób, że nie tylko nie jest wymagana ingerencja w architekturę portów modułu, ale nawet nie ma konieczności uzyskania wiedzy o szczegółowym działaniu pozostałych aspektów integracji modułu z mikroprocesorem.

Cel pracy został osiągnięty, a zakres w pełni zrealizowany. Zaprojektowano, zaimplementowano oraz zbadano efektywność predyktora skoków, wspierającego wiele różnych algorytmów predykcji. Przeprowadzono weryfikację funkcjonalną zarówno samego rdzenia procesora, jak i modułu predykcji skoków, dla różnych konfiguracji. Wszystkie warianty predyktora posiadają wspólny interfejs, co zapewnia planowaną modularność implementacji.

Jeśli chodzi o dalsze prace rozwoju projektu, ciekawym do zbadania byłby wpływ zaimplementowanych predyktorów skoków na parametry zsyntezowanego, fizycznego układu. Na parametry te składałyby się przede wszystkim zajętość powierzchni układu scalonego, maksymalna częstotliwość sygnału zegarowego, jak i zużycie mocy. Pozwoliłoby to przeprowadzić dokładniejszą analizę jakości różnych strategii predyktora, porównując efektywność ich predykcji do zasobów przez nie zużywanych.

Samo badanie efektywności predyktorów mogłoby również zostać bardziej rozbudowane. Wszystkie badania przeprowadzono dla tylko jednego programu testowego (za wyjątkiem jego małej modyfikacji podczas testów strategii BHT). Analiza efektywności predykcji dla innych programów byłoby prawdopodobnie bardziej miarodajna, niż tylko dla jednego. Przede wszystkim należałoby dokładniej zbadać strategię BHT, by precyzyjniej określić sytuacje, w których daje niepożądane efekty.

Innym ulepszeniem, które można w przyszłości zastosować, jest zmiana struktury tablicy rekordów wewnątrz modułów BTB, BHT oraz RET_TABLE, tak aby zmniejszyć szanse na wystąpienie kolizji adresów, bez potrzeby zwiększania rozmiaru buforu. Przykładowo, można by skorzystać z rozwiązania zastosowanego w mikroprocesorze SonicBOOM, gdzie rekordy przechowywane są w skompresowanej postaci.

Oczywiście, zawsze możliwe jest też stworzenie całkowicie nowych strategii predykcji, jak np. predyktor TAGE wykorzystywany w mikroprocesorze SonicBOOM.

Bibliografia

[1] Andrew Waterman, Krste Asanović, SiFive Inc., *The RISC-V Instruction Set Manual*, 2019.

[2] Fabian Oleś, Oskar Stajer, Paweł Lempa, Bartłomiej Jękot, Błażej Korzuch, *System mikroprocesorowy z 64. bitowym rdzeniem RISC-V.* Sprawozdanie projektowe PBL, 2023.

[3] Błażej Korzuch, *Mechanizm wymiany danych pomiędzy jednostką centralną mikroprocesora RISC-V a urządzeniami peryferyjnymi*, Praca magisterska, 2023.

[4] M. Michael, J. E. Moreira, D. Shiloach and R. W. Wisniewski, "Scale-up x Scale-out: A Case Study using Nutch/Lucene," *2007 IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, 2007, pp. 1-8, doi: 10.1109/IPDPS.2007.370631.

[5] John L. Hennessy David A. Patterson. *Computer Organization and Design, The Hardware/Software Interface: RISC-V Edition*. Morgan Kaufmann Publishers, 2017.

[6] Agner Fog, *The microarchitecture of Intel, AMD, and VIA CPUs An optimization guide for assembly programmers and compiler makers*, 2023

[7] I. Healy, P. Giordano and W. Elmannai, "Branch Prediction in CPU Pipelining," *IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, New York, NY, USA, 2023, pp. 0364-0368, doi: 10.1109/UEMCON59035.2023.10316163.

[8] Rinu Joseph, *A Survey of Deep Learning Techniques for Dynamic Branch Prediction*, 2021

[9] https://github.com/lowRISC/ibex (dostęp: 11.06.2024)

[10] https://github.com/openhwgroup/cva6 (dostęp: 11.06.2024)

[11] Zhao, Jerry and Abraham Gonzalez. *Sonic BOOM: The 3rd Generation Berkeley Out-of-Order Machine*, 2020.

[12] https://docs.boom-core.org/en/latest/sections/intro-overview/boom-pipeline.html (dostęp: 11.06.2024)

[13] K. Matsui, M. Ashraful Islam and K. Kise, "An Efficient Implementation of a TAGE Branch Predictor for Soft Processors on FPGA," *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, Singapore, 2019, pp. 108-115, doi: 10.1109/MCSoC.2019.00023.

[14] https://github.com/wildpika/polsl_riscv_mgr (dostęp: 11.06.2024)

[15] Intel, *Questa Intel FPGA Edition Simulation User Guide Updated for Intel® Quartus® Prime Design Suite: 23.1*, 2023

Spis skrótów i symboli

RISC	Procesor o zredukowanej liście rozkazów (ang. Reduced Instruction Set Computing)
RAM	Pamięć operacyjna, pamięć o dostępie swobodnym (ang. Random Access Memory)
ROM	Pamięć tylko do odczytu (ang. Read Only Memory)
BHT	Tablica historii skoków warunkowych (ang. Branch History table)
BTB	Bufor adresów docelowych skoków (ang. Branch Target Buffer)
RAS	Stos adresów powrotu (ang. Return Address Stack)
ISA	Zbiór instrukcji procesora, model programowy procesora (ang. Instruction Set Architecture)
FPGA	*Bezpoś*rednio programowalna macierz bramek (*ang. Field Programmable Logic Device*)
ASIC	Specjalizowany układ scalony (ang. Application Specific Integrated Circuit)
PC	Licznik programu, adres aktualnej instrukcji (ang. Program Counter)
CSR	Rejestr kontrolno – statusowy (ang. Control Status Register)
PMP	Układ fizycznej ochrony pamięci (ang. Physical Memory Protection)
FIFO	Kolejka typu „pierwsze weszło, pierwsze wyszło” (ang. First-In, First-Out)
LIFO	Stos typu „ostatnie weszło, pierwsze wyszło” (ang. Last-In, First-Out)
NLP	Predyktor kolejnego adresu (ang. Next Line Predictor)
BPD	Predyktor wspierający (ang. Backing Predictor)
BIM	Tablica 2-bitowych liczników nasycenia (ang. Bimodal Table)
TAGE	Predyktor typu tagowanego-geometrycznego (ang. Tagged Geometric)
ALU	Jednostka arytmetyczno-logiczna (ang. Arithmetic Logic Unit)
CSV	Format danych oddzielonych przecinkami (ang. Comma Seperated Values)

Lista dodatkowych plików, uzupełniających tekst pracy

W systemie do pracy dołączono dodatkowe pliki zawierające:

- kod źródłowy wszystkich zaimplementowanych predyktorów
- kod źródłowy wszystkich wykorzystanych programów testowych
- kod źródłowy skryptów wykorzystanych podczas badań