Implementacja out-of-order rozszerzenia wektorowego RISC-V w procesorze Coreblocks

(Out-of-order implementation of RISC-V vector extension in Coreblocks processor)

Kuba Nowak

Praca magisterska

Promotor: dr Marek Materzok

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

19 maja 2024

Streszczenie

Szybkie i wydajne energetycznie operacje na dużej ilości danych stają się aktualnie podstawą współczesnej informatyki. Jednocześnie dąży się do jej coraz większej otwartości, w związku z czym już nie tylko programy, ale także sprzęt są udostępniane na otwartej licencji. Wpisując się w te trendy, niniejsza praca implementuje rozszerzenie wektorowe do otwartej architektury RISC-V. Implementacja ta jest w mikroarchitekturze *out-of-order* i rozszerza istniejący procesor Coreblocks o otwartym kodzie źródłowym. Przeanalizowano i zoptymalizowano powstały rdzeń, a następnie porównano go do czołowych istniejących implementacji. Zweryfikowano tym samym możliwość zastosowania biblioteki Transactron – leżącej u podstaw Coreblocksa – do tworzenia bardziej skomplikowanych układów.

Fast and energy-efficient operations on large data are crucial for the today's computing. At the same time, efforts are being made to make computer science more open. Nowdays, not only software, but also hardware, is shared under open source licences. Following these trends, this thesis implements a vector extension of the RISC-V architecture. The implementation uses an out-of-order microarchitecture and extends the Coreblocks open source processor. As part of this work, the performance of the created core has been analysed and optimised. After that it have been compared with state of the art processor. Additionaly, the Transactron library, which forms the basis of Coreblocks, has been tested for the use in more complex project.

Spis treści

Rozdział 1.

Wstęp

Instrukcje wektorowe, wprowadzone jeszcze w CRAY-1 [Russell 1978], przeżywają aktualnie swój renesans, ze względu na osiągany z ich pomocą wzrost wydajności. Sprzętowe wsparcie działań wektorowych pozwala zaimplementować mnożenie macierzy szybciej oraz bardziej efektywnie energetycznie, co nabiera szczególnego znaczenia w kontekście współczesnych koncepcji *AI on the edge*, w których to urządzenia końcowe – zazwyczaj dysponujące małą mocą obliczeniową i działające na baterię – będą wykonywały obliczenia potrzebne by zaaplikować sieci neuronowe.

W ostatnich latach dużym wzrostem popularności cieszy się architektura RISC-V, nad którą prace rozpoczęły się w 2010 roku [RISC-V International 2023] i z roku na rok przyłącza się do nich coraz więcej osób i organizacji. RISC-V jako projekt współczesny – i mający początki na Uniwersytecie w Berkeley – podszedł krytycznie do starszych architektur, podczas tworzenia których podjęto różnego złe decyzje projektowe. Jego celem jest zaprojektowanie nowej architektury od podstaw, w sposób modularny, pozbawiony obciążeń wynikających ze wstecznej kompatybilności[1]. Podstawowy zbiór instrukcji „I" składa się z kilkunastu instrukcji sterujących, arytmetyczno-logicznych i operujących na pamięci. Większa część instrukcji jest natomiast zdefiniowana w rozszerzeniach, przykładowo: „M" – instrukcje mnożenia i dzielenia liczb całkowitych, „F" – instrukcje operujące na liczbach zmiennopozycyjnych 32-bitowych, „V" – instrukcje wektorowe.

Celem niniejszej pracy jest dodanie wsparcia dla rozszerzenia „V" w procesorze Coreblocks – zoptymalizowanego dla FPGA[2] (ang. field-programmable gate array) – przy założeniu wykonania instrukcji wektorowych *out-of-order*. Głównym wkładem twórczym jest implementacja tego rozszerzenia z użyciem biblioteki Transactron oraz następna jego optymalizacja. Badane są także ograniczenia zastosowań Transactrona w skomplikowanych projektach, bazując na przykładzie stworzonego rdzenia wektorowego.

[1] Dobrym przykładem takich obciążeń jest *delay slot*, który został wprowadzony w MIPS-ach potokowych, a musi być wspierany także w MIPS-ach *out-of-order* [Shen and Lipasti 2002].

[2] W przeciwieństwie do wielu współczesnych, alternatywnych implementacji, które celują w ASIC.

1.1. Współczesne mikroarchitektury procesorów

Dzisiejsze procesory ogólnego przeznaczenia można podzielić ze względu na zastosowaną w nich mikroarchitekturę na:

- proste procesory *in-order*,
- potokowe procesory *in-order*,
- procesory *out-of-order*.

Prosta mikroarchitektura *in-order* jest tania w implementacji i była najwcześniej używaną mikroarchitekturą procesorów. Taki procesor może jednocześnie przetwarzać tylko jedną instrukcję, a ich przetwarzanie odbywa się w porządku programu. Wykonywana instrukcja musi się skończyć i zostać wdrożona do stanu architektonicznego zanim rozpocznie się obsługa następnej. W wielu przypadkach jednak wiemy jaka będzie następna instrukcja do wykonania. Co więcej, jej dane wejściowe mogą być niezależne od aktualnie wykonywanego polecenia procesora, a w takiej sytuacji oczekiwanie na jego zakończenie jest marnotrawieniem czasu. Obie instrukcje mogą się albowiem wykonywać równolegle. Obserwacja ta stoi u podstaw rozszerzenia prostych procesorów *in-order* o przetwarzanie potokowe, w którym jednocześnie przetwarzanych jest wiele instrukcji w porządku programu.

W procesorach potokowych *in-order* przetwarzanie instrukcji podzielone jest na etapy, które ułożone jeden za drugim tworzą potok. W każdym etapie może się znajdować na raz tylko jedna instrukcja, która – o ile następny etap jest wolny lub właśnie zwalniany – przepływa do następnego etapu wraz z końcem cyklu zegarowego. Gdy nie można jej przekazać dalej, to procesor wykonuje *stall*, czyli zatrzymuje instrukcję w aktualnym etapie i czeka aż następny się zwolni. Dzięki temu w procesorze może być na raz przetwarzane tyle instrukcji ile jest etapów potoku, więc przepustowość rdzenia jest większa niż w przypadku prostych procesorów *in-order*.

Procesory potokowe *in-order* przetwarzają instrukcje zgodnie z porządkiem w programie, w związku z tym by były efektywne i wykonywały program z maksymalną przepustowością, to kolejne instrukcje muszą być od siebie niezależne. Nieefektywności powstają, gdy istnieją zależności na danych, co zaprezentowano w kodzie 1.1, gdzie polecenie `add`[3], czeka na wykonanie `lw`[4] i tym samym blokuje wykonanie `addi`[5], które nie ma żadnych zależności, więc potencjalnie mogło by zostać już wykonane.

Nieoptymalność ta wynika z faktu, że procesory potokowe *in-order* wykonują instrukcje w porządku programu. Naturalnym rozszerzeniem są więc procesory *out-of-order*, które pozwalają wykonać instrukcje poza porządkiem programu tak długo, jak zachowany jest przepływ danych między nimi. W kodzie 1.1 są dwa niezależne

[3]Instrukcja dodająca dwie liczby całkowite z rejestrów skalarnych.
[4]Instrukcja *load word* pobierająca z pamięci 32-bitowe słowo.
[5]Instrukcja dodająca do rejestru całkowitoliczbowego stałą.

```
1 lw x2, 0(x1)
2 add x3, x2, x2
3 addi x4, x0, 44
```

Kod źródłowy 1.1: Kod w asemblerze RISC-V, który wykona się nieoptymalnie na procesorze potokowym.

strumienie danych. Pierwszy zawiera instrukcje 1 i 2, a drugi – instrukcję 3. W związku z tym w procesorze *out-of-order* rzeczywistym porządkiem wykonania instrukcji będzie 1, 3, 2. Ukryje to częściowo opóźnienie instrukcji `lw` poprzez wykonanie instrukcji `addi`.

Oczywiście procesor nie może sam z siebie zmienić kolejności wykonania instrukcji, bowiem będzie to niezgodne z gwarancjami zawartymi w architekturze. W związku z tym w każdym momencie, w którym programista będzie chciał zweryfikować rzeczywisty stan programu (weryfikacja taka odbywa się poprzez przerwanie lub wyjątek), procesor musi być w stanie zaprezentować spójny stan architektoniczny. Utrzymywanie takiego stanu przy jednoczesnym wykonywaniu instrukcji poza kolejnością jest kosztowne, toteż choć mikroarchitektura *out-of-order* jest najbardziej efektywna z przedstawionych, to jest także najbardziej skomplikowana i wymaga najwięcej energii.

Wyżej przedstawione mikroarchitektury są stosowane w procesorach ogólnego przeznaczenia. Niemniej jednak można także wyróżnić specjalistyczne mikroarchitektury przeznaczone dla akceleratorów, takie jak tablice systolityczne czy rdzenie wektorowe.

1.2. Historia instrukcji wektorowych

Instrukcje wektorowe w procesorach można zdefiniować na dwa sposoby. Pierwszy wywodzi się z komputera CRAY-1, gdzie wektor jest ciągiem liczb pewnej długości nie większej niż maksymalna dla danego procesora. Zapewnia to elastyczność użycia, bowiem raz wektor może być pełen i z użyciem instrukcji procesora operuje się na wszystkich elementach na raz, a innym razem może być zapełniony tylko w części i wtedy te same instrukcje operują na mniejszej liczbie elementów, a tym samym trwają krócej. Dodatkową funkcjonalnością w CRAY-1 jest możliwość potokowania [Russell 1978]. Jeśli dwie instrukcje wektorowe następują jedna po drugiej i druga używa wyniku pierwszej, to mogą one się wykonywać równolegle i druga instrukcja zamiast odczytywać swój operand z pliku rejestrów otrzymuje go bezpośrednio z jednostki funkcyjnej. Jednakże, wraz z rozwojem procesorów *out-of-order* w latach 90-tych ubiegłego wieku, procesory z wyspecjalizowanymi jednostkami zaczęły być wypierane przez procesory *out-of-order* ogólnego przeznaczenia ze względu na prostotę implementacji programów, przy ówcześnie porównywalnej wydajności.

Wraz ze zdobywaniem dominacji na rynku procesorów przez architekturę x86 zaczęła się też zmieniać definicja instrukcji wektorowych. Architektura x86 udostępnia instrukcje wektorowe w postaci rozszerzeń SSE (ang. Streaming SIMD Extension) i AVX (ang. Advanced Vector Extension), które pomimo tego że nawiązują do przetwarzania wektorowego, to działają na całkowicie innej zasadzie niż te znane z CRAY-1. Instrukcje w x86 operują zawsze na fragmentach ustalonej długości; w zależności od zaimplementowanego zestawu rozszerzeń są to fragmenty 128-, 256- lub 512-bitowe. Do operowania na fragmentach różnej długości są zdefiniowane różne instrukcje. Co więcej, nie ma możliwości przeprowadzenia operacji tylko na części fragmentu; przykładowo jeśli potrzebne jest tylko pierwsze 384 bitów z fragmentu 512-bitowego, to operacja zostanie i tak wykonana na wszystkich 512-bitach, natomiast ostatnie 128-bitów zostanie zignorowane podczas zapisywania do rejestru.

Architektura instrukcji wektorowych przedstawiona w x86 jest prostsza w implementacji fizycznej w procesorze, jednakże niesie ze sobą narzut wydajnościowy w porównaniu do tej wprowadzonej w CRAY-1. W celu zilustrowania tego narzutu można rozważyć przykład, w którym dodawane są dwie tablice sześćdziesięciu czterech 64-bitowych liczb całkowitych. Zakładając, że tablice są w pamięci RAM (ang. Random Access Memory), to w przypadku CRAY-1, wystarczą do tego cztery instrukcje: załaduj z pamięci RAM dane do rejestrów wektorowych (2 instrukcje po jednej dla każdej tablicy wejściowej), dodaj rejestry, zapisz rejestr wynikowy do pamięci RAM. W przypadku x86 trzeba wykonać 4 analogiczne instrukcje dla każdego fragmentu tablic. Zakładając, że mamy najnowszy procesor wyposażony w AVX2, to fragmenty są 512-bitowe i tym samym jedna tablica o 4096 bitach składa się z 8 fragmentów. Aby więc wykonać tę samą operację na x86, trzeba przetworzyć 8 razy więcej instrukcji, co przekłada się na większy koszt energetyczny związany z ich pobraniem z cache/pamięci oraz dekodowaniem. Co więcej, operowanie na 8 zestawach instrukcji utrudnia prefetching danych z pamięci, co może powodować niepotrzebne opóźnienia.

W RISC-V zdecydowano się dodać w pierwszej kolejności[6] instrukcje wektorowe znane z CRAY-1. Rozszerzenie to dostało oznaczenie „V" (czasami można się spotkać także ze skrótem RVV rozwijającym się do RISC-V V). Aktualnie specyfikacja jest zamrożona w wersji 1.1 i oczekuje na ratyfikację przez komitet standaryzacyjny RISC-V.

W celu uniknięcia niejednoznaczności w nazewnictwie, w ramach tej pracy poprzez instrukcje/rozszerzenie wektorowe należy rozumieć architekturę wywodzącą się z CRAY-1 i definiowaną przez rozszerzenie „V" RISC-V; natomiast w sytuacji gdy będzie potrzeba by odnieść się do instrukcji SSE/AVX z x86, będzie stosowane określenie „instrukcje SIMD" (ang. Single Instruction Multiple Data).

[6] Aktualnie trwają prace nad rozszerzeniem „P" dodającym instrukcje SIMD.

1.3. Najważniejsze elementy rozszerzenia „V”

RVV jest aktualnie największym rozszerzeniem zdefiniowanym do RISC-V, specyfikuje między innymi:

- 32 architektoniczne rejestry wektorowe (oznaczane jako `v0-v31`);
- ponad 250 nowych instrukcji;
- stałe zdefiniowane na etapie implementacji: `ELEN`, `VLEN`;
- 7 CSR-ów[7], w tym: `vstart`, `vl`, `vtype`.

Każdy architektoniczny rejestr wektorowy ma długość zdefiniowaną na etapie implementacji równą `VLEN` bitów i dzieli się na elementy o długości `ELEN` bitów, gdzie `ELEN` może przyjąć zgodnie z aktualną specyfikacją jedną z czterech wartości: {8,16,32,64}. Tym samym `ELEN` określa maksymalną długość jednego elementu[8], który może przetworzyć dana implementacja (minimalną długością jest zawsze 8 i każda implementacja musi być w stanie przetworzyć elementy o długości mniejszej niż `ELEN`).

Każdy z rejestrów architektonicznych zawiera pewną liczbę elementów ustalonej szerokości (`EEW` – Effective Element Width), szerokość ta musi być nie większa niż `ELEN` oraz należeć do tego samego zbioru co `ELEN`. Tym samym w zależności od tego, jakie jest `EEW`, różna jest maksymalna liczba elementów w rejestrze i waha się od `VLEN/64` do `VLEN/8`. Co więcej, w rejestrze może być mniej elementów niż wynosi jego maksymalna pojemność. W związku z tym, w celu określenia liczby elementów do przetworzenia, przechowuje się skuteczną długość rejestru w CSR-rze `vl`. `EEW` jest natomiast określany na podstawie aktualnie przetwarzanej instrukcji oraz rejestru CSR `vtype`. Instrukcje operujące na danych rozpoczynają pracę od odczytania `vl` oraz `vtype` w celu określenia szerokości i liczby elementów. Następnie przetwarzają odpowiednio długi fragment początkowy rejestrów wektorowych będących operandami i zapisują wynik do docelowego rejestru wektorowego.

Oprócz kontrolowania liczby przetwarzanych elementów można też sterować obliczaniem pojedynczych elementów za pomocą maski bitowej. Prawie każda z 250 wprowadzanych instrukcji występuje w dwóch wersjach. W jednej z nich każdy element jest przetwarzany, natomiast w drugiej wersji rejestr `v0` jest traktowany jako maska bitowa i wynik jest zapisywany wtw. gdy bit dla danego elementu jest zapalony.

[7] ang. Control and Status Register – rejestr sterowania i statusu.

[8] W dalszej części pracy `ELEN` będzie stosowane zamiennie na określenie stałej oraz rejestru o długości zadanej przez tą stałą.

Polityka dla elementów, które mają maskę równą 0 oraz dla elementów leżących poza `vl` (tzw. elementy ogonowe), jest określana przez CSR `vtype`. Pomijane elementy mogą być zachowywane w niezmienionej postaci (*undisturbed*) albo ignorowane. W przypadku gdy są ignorowane, specyfikacja wymaga, by albo były skopiowane w niezmienionej postaci, albo miały wszystkie bity zapalone na 1. Te dwie polityki pozwalają optymalizować wykonanie instrukcji, bowiem dzięki temu procesory z przenazywaniem rejestrów wektorowych mogą pominąć kopiowanie niemodyfikowanych elementów i zawsze zwracać dla nich same jedynki.

Dodatkową optymalizacją umożliwianą przez rozszerzenie „V" jest możliwość łączenia rejestrów architektonicznych w grupy. Służy do tego parametr `LMUL` – przechowywany w `vtype` – który specyfikuje mnożnik długości rejestrów wektorowych. Dopuszczalne wartości to: $\{1/8, 1/4, 1/2, 1, 2, 4, 8\}$. Ustawienie `LMUL` wpływa na długość rejestrów architektonicznych, które mają wtedy `VLEN * LMUL` bitów. W przypadku wartości ułamkowych `LMUL` i 1, liczba rejestrów wektorowych pozostaje bez zmian. Natomiast w przypadku wartości `LMUL > 1`, liczba rejestrów wektorowych jest równa `32 / LMUL` i z oryginalnych 32 rejestrów tylko co `LMUL`-ty jest poprawny, tzn. dla `LMUL=8` poprawnymi rejestrami są wyłącznie {`v0`, `v8`, `v16`, `v24`}. Dzięki temu z użyciem jednej instrukcji wektorowej można przetwarzać 8 razy więcej danych niż wynikałoby to z ograniczeń fizycznych rejestrów wektorowych (`VLEN`). Warto w tym miejscu zauważyć, że dla `VLEN=4096` długość jednego rejestru z `LMUL=8` jest równa 4KB, czyli rozmiarowi jednej strony pamięci RAM.

```
Example VLEN=128b, with SEW/LMUL=16

Byte      F E D C B A 9 8 7 6 5 4 3 2 1 0
vn        - - - - - - - - 7 6 5 4 3 2 1 0  SEW=8b, LMUL=1/2

vn          7   6   5   4   3   2   1   0  SEW=16b, LMUL=1

v2*n            3       2       1       0  SEW=32b, LMUL=2
v2*n+1          7       6       5       4

v4*n                    1               0  SEW=64b, LMUL=4
v4*n+1                  3               2
v4*n+2                  5               4
v4*n+3                  7               6
```

Rysunek 1.1: Przykładowe rozmieszczenie kolejnych elementów w fizycznych rejestrach wektorowych przy założeniu stałej liczby elementów w grupie rejestrów (stały stosunek `SEW/LMUL=16`). W tym przykładzie dla `SEW` $\in \{8, 16\}$ cała grupa mieści się w jednym rejestrze fizycznym. Dla `SEW = 32` zajmuje dwa rejestry fizyczne i zaczyna się w rejestrze o parzystym identyfikatorze. A dla `SEW = 64` zajmuje cztery rejestry i zaczyna się w rejestrze fizycznym o identyfikatorze podzielnym przez 4.

Źródło: RISC-V International (Sept. 2021). *RISC-V "V" Vector Extension.* URL: `https://github.com/riscv/riscv-v-spec/releases/tag/v1.0`

Rozdział 2.

Opis implementacji

2.1. Funkcjonalności procesora Coreblocks

Jako procesor stanowiący podstawę do implementacji rozszerzenia „V" wybrano Coreblocksa[1] tworzonego przez osoby powiązane z Instytutem Informatyki Uniwersytetu Wrocławskiego. Jest to modularny procesor RISC-V w mikroarchitekturze *out-of-order*, którego głównymi celami są prostota implementacji i rozszerzalność.

Coreblocks jest tworzony w Pythonie z użyciem biblioteki do opisu sprzętu Amaranth[2]. Tym samym Python jest traktowany jako metajęzyk, generujący AST[3] z klas Amarantha, które z kolei opisują w jaki sposób ma zostać z syntezowany układ (istnieje możliwość wygenerowania Veriloga bądź RTLIL). Nad biblioteką Amaranth nadbudowana jest biblioteka Transactron będąca częścią projektu Coreblocks, a stanowiąca wysokopoziomową warstwę abstrakcji pozwalającą w generyczny sposób łączyć moduły Amarantha.

Aktualnie Coreblocks wspiera w pełni `rv32icbm`, jednakże jest to relatywnie nowy projekt i duża część optymalizacji nie jest jeszcze zaimplementowana – będzie to rzutowało w dalszej części pracy na wyniki prezentowanego rozszerzenia wektorowego. W szczególności:

- instrukcje skoku wykonują *stall* potoku aż do ich zatwierdzenia do stanu architektonicznego;
- operacje na magistrali pamięci mają niską wydajność (1 operacja na 4 cykle);
- jednostka LOAD/STORE wspiera wykonanie tylko jednej instrukcji na raz (poprzednia musi być zatwierdzona do stanu architektonicznego, zanim rozpocznie się przetwarzanie następnej);

[1] `https://github.com/kuznia-rdzeni/coreblocks`
[2] `https://github.com/amaranth-lang/amaranth`
[3] ang. Abstract Syntax Tree.

- brakuje predyktora skoków;
- nie ma prefetchingu instrukcji ani danych;
- przerwania są w trakcie implementacji.

2.2. Funkcjonalności zaimplementowane i pominięte

Jak wspomniano we wstępie, rozszerzenie wektorowe jest największym rozszerzeniem RISC-V, w związku z tym – ze względu na ograniczone zasoby czasowe – zdecydowano się zaimplementować w ramach tej pracy tylko wybrane funkcjonalności RVV. Położono natomiast nacisk na to by moduł wektorowy działał w pełni *out-of-order*. Zaimplementowano więc następujące części RVV:

- operacje arytmetyczno-logiczne;
- LOAD/STORE typu *unit-stride*;
- wsparcie dla `ELEN = 32`;
- wsparcie dla operacji na elementach o szerokości mniejszej niż `ELEN`.

Natomiast z istotnych elementów pominięto:

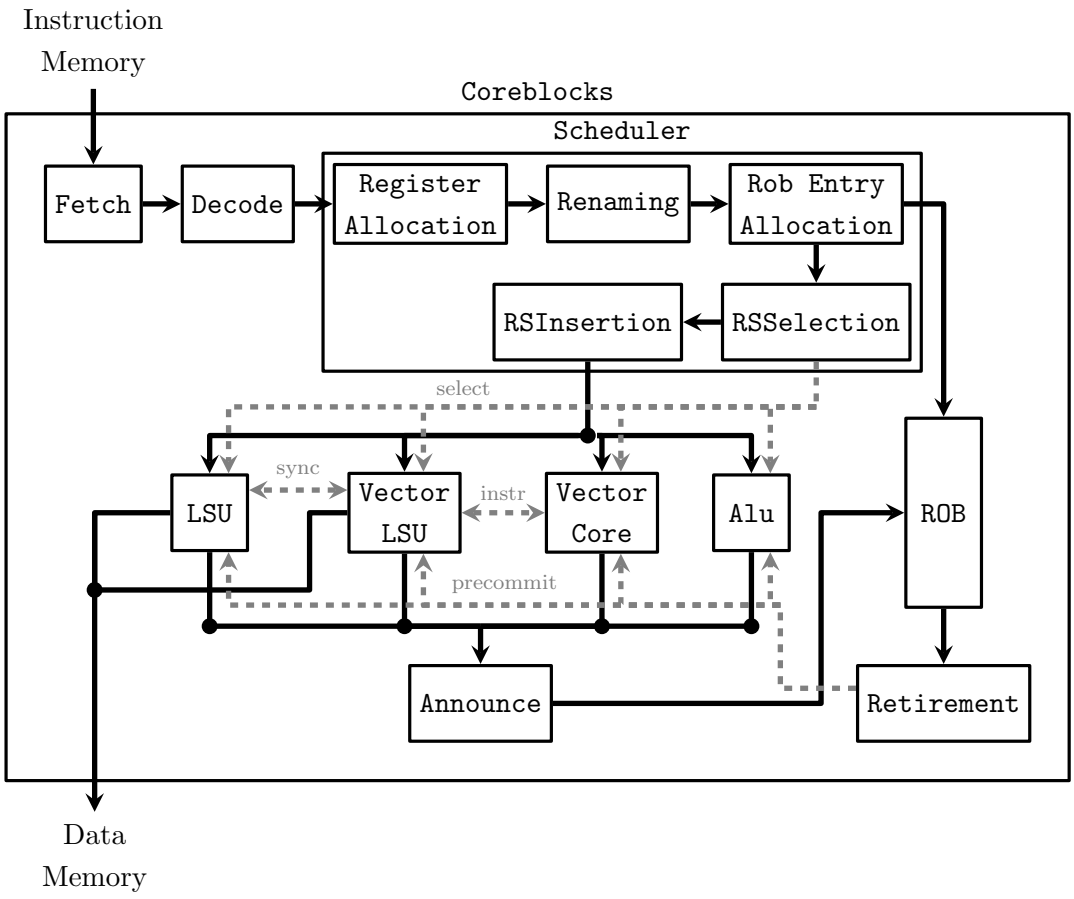
- wsparcie dla operacji zmieniających szerokość elementów;
- wsparcie dla `LMUL != 1`;
- obsługę maskowania elementów;
- instrukcje redukcji.

2.3. Mikroarchitektura wysokopoziomowa

Wysokopoziomowa mikroarchitektura została przedstawiona na rysunku 2.1. Z perspektywy rdzenia skalarnego, w celu umożliwienia obsługi instrukcji wektorowych dodano:

- `VectorCore` zawierający większość przetwarzania instrukcji wektorowych, w tym:
  - RS[4] skalarny dla skalarnych operandów instrukcji wektorowych;
  - RS wektorowy;

[4] ang. Reservation Station.



Rysunek 2.1: Schemat ilustrujący Coreblocksa z dodanym rozszerzeniem wektorowym wraz z sygnałami sterującymi istotnymi z punktu widzenia tego rozszerzenia.

  - obsługę CSR-ów wektorowych;
  - wykonanie operacji arytmetyczno-logicznych;
  - rozgłaszanie zakończenia instrukcji wektorowych;
  - obsługę retirementu instrukcji wektorowych i ich wdrażanie do stanu architektonicznego *in-order*;
- `VectorLSU` odpowiadający za wykonanie instrukcji wektorowych operujących na pamięci;
- wsparcie dekodowania instrukcji wektorowych w dekoderze instrukcji;
- typ rejestrów fizycznych.

Należy podkreślić, że dzięki zamknięciu większości funkcjonalności w `VectorCore` z powyższych zmian, jedynie typ rejestrów fizycznych jest zmianą stałą – niezależną od tego czy rozszerzenie wektorowe jest włączone czy też nie. By zminimalizować jego narzut, został on zaprojektowany tak, by w przyszłości uwspólnić go z rozszerzeniami zmiennopozycyjnymi. Dzięki temu wpływ nieużywanego rozszerzenia „V" na rdzeń skalarny jest minimalny.

Tworząc moduł wektorowy zdecydowano się na użycie klasycznej mikroarchitektury, w której plik rejestrów jest podzielony na fragmenty i do każdego fragmentu przypisana jest jednostka funkcyjna z niego odczytująca i zapisująca. Fragment pliku rejestrów, jednostka funkcyjna oraz powiązana z nimi logika tworzą taśmę[5].

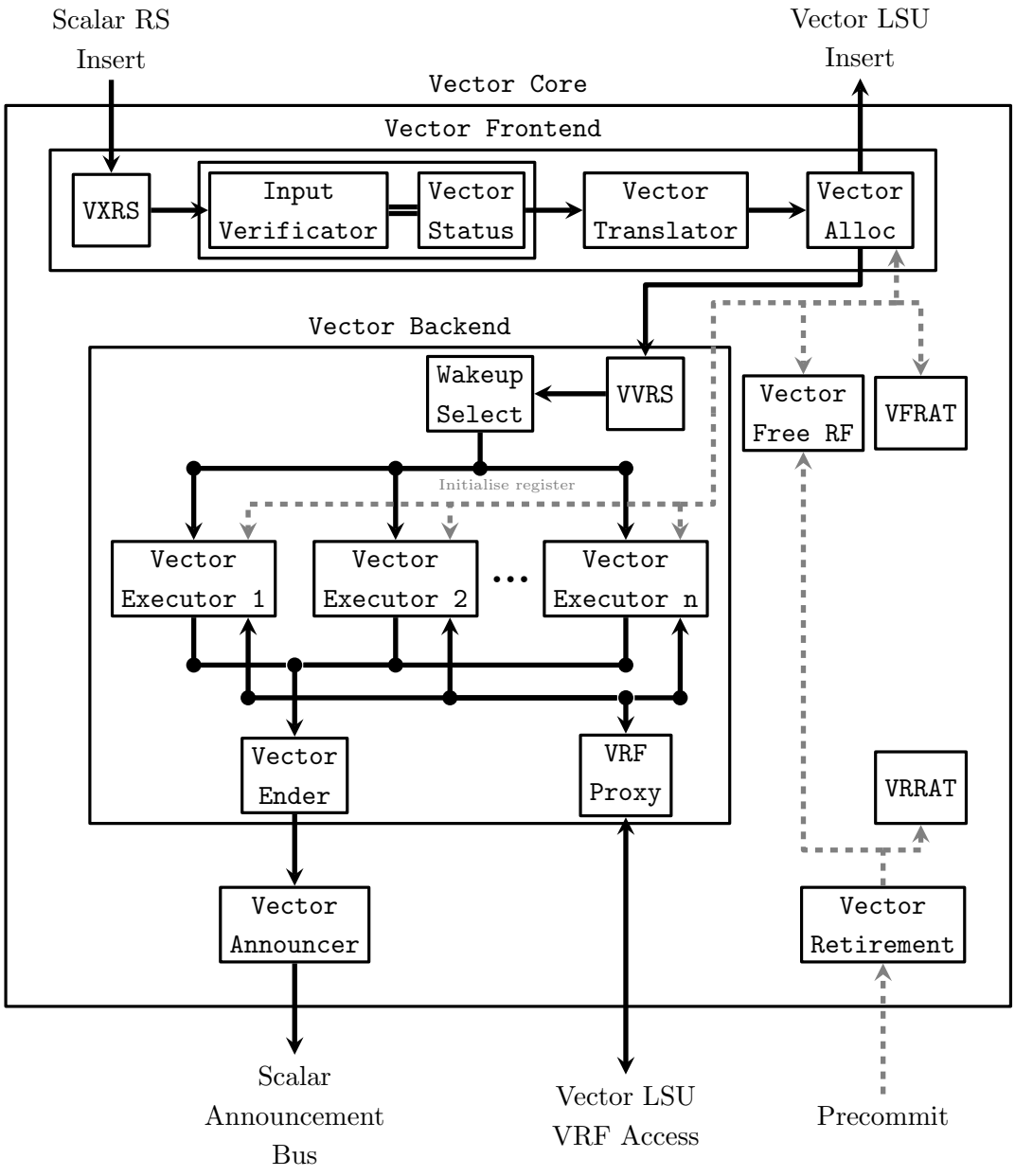
Elementy są ułożone kolejno w rejestrze wektorowym, czyli najpierw zapełniany jest pierwszy fragment, następnie drugi i tak aż do końca. W przypadku gdy operacje wektorowe odbywają się na wektorach mniejszych niż `VLEN` powoduje to nieoptymalne zużycie zasobów, bowiem ostatnie fragmenty nie będą w ogóle zajęte, a wykorzystanie początkowych będzie pełne. Jednakże taka konstrukcja znacząco ułatwia implementację, gdyż nie trzeba tworzyć logiki translacji adresów między adresami architektonicznymi, a adresami mikroarchitektonicznymi.

2.4. Przepływ instrukcji arytmetyczno-logicznych

2.4.1. Frontend wektorowy

Stworzony rdzeń wektorowy składa się z kilku części przedstawionych na rysunku 2.2. Przychodzące instrukcje w pierwszej kolejności trafiają do `VectorFrontend`, gdzie w zmodyfikowanym RS-ie zachowującym kolejność instrukcji czekają na operandy skalarne. Wymóg zachowania kolejności instrukcji wchodzących

[5]W literaturze angielskiej stosuje się określenie *lane*, które nie ma utartego odpowiednika w języku polskim. Ze względu na trudności z jego spolszczeniem i następną odmianą przez przypadki, zdecydowano się – w ramach tej pracy – tłumaczyć je jako „taśma".



Rysunek 2.2: Schemat zaimplementowanego w Coreblocksie rdzenia wektorowego.

jest potrzebny, ze względu na zależności między instrukcjami na stanie globalnym. W szczególności zazwyczaj przed każdą instrukcją wektorową operującą na danych występuje instrukcja `vset{i}vl{i}`, konfigurująca wektorowe `vl` oraz `vtype`, tak by następująca po niej operacja została wykonana na poprawnych danych. Należy przy tym zauważyć, że kolejność instrukcji ważna jest tylko we frontendzie jednostki wektorowej i jest ona usuwana poprzez tagowanie, stanem CSR-ów wektorowych, wszystkich instrukcji wektorowych operujących na danych. Dzięki czemu, na etapie wykonania, może zajść reordering między instrukcjami wektorowymi.

`VectorFrontend` po odczytaniu instrukcji z gotowymi operandami skalarnymi z `VXRS` przekazuje je do jednostek `VectorInputVerificator` i `VectorStatusUnit`. Pierwsza z nich sprawdza, czy instrukcje wchodząca jest poprawną instrukcją (wartość `vstart` jest wspierana, a `vtype` ma zgaszony bit `vill`), jeśli tak nie jest, to zgłasza wyjątek. Instrukcje zarezerwowane – zgodnie ze specyfikacją RISC-V – mają niezdefiniowane zachowanie. By nie komplikować nadmiernie logiki weryfikacji, przyjęto, że nie będą one odrzucane.

Zweryfikowana instrukcja jest następnie przekazywana do `VectorStatusUnit`, gdzie jest albo tagowana aktualną konfiguracją CSR-ów wektorowych, albo je modyfikuje (w przypadku instrukcji `vset{i}vl{i}`). Zarówno weryfikacja instrukcji jak i jej przetworzenie w `VectorStatusUnit` zawsze odbywają się w tym samym cyklu, bowiem by zweryfikować instrukcję potrzebna jest aktualna wartość CSR-a `vtype`.

Następnie na otagowanej instrukcji wykonywane są translacje, mające ją przystosować do wewnętrznego formatu danych rdzenia wektorowego. Rejestr docelowy jest kopiowany i zapisywany jako trzeci rejestr źródłowy (dla potrzeb operacji które operują na trzech rejestrach źródłowych na raz). Natomiast wartość potencjalnego operandu pochodzącego z rejestru skalarnego jest zapisywana jako stała i tym samym zwężana jest szyna łącząca kolejne elementy potoku[6].

Ostatnim etapem przetwarzania instrukcji w `VectorFrontend` jest zaalokowanie nowego rejestru wektorowego na wektor wyjściowy, jego następna inicjalizacja oraz wykonanie przenazywania wektorowych rejestrów źródłowych. Tak przygotowana instrukcja jest kierowana albo do `VectorLSU` albo do `VectorBackend` wykonującego operacje arytmetyczno-logiczne.

2.4.2. Backend wektorowy

Instrukcje wchodzące do `VectorBackend` są wstawiane do `VVRS`, gdzie oczekują aż wszystkie operandy wektorowe będą gotowe. Stan dla każdego rejestru wektorowego (to czy jest jeszcze w trakcie obliczania, czy też nie), jest przechowywany w

[6]Transformacja ta wynika to z obserwacji, że żadna instrukcja wektorowa, poza `vset{i}vl{i}` (a te zostały przetworzone na wcześniejszym etapie), nie używa jednocześnie stałej i operandu z rejestru skalarnego.

scoreboardzie, z którego odczytywany jest status podczas wstawiania instrukcji do VVRS, natomiast po wstawieniu VVRS nasłuchuje na szynie rozgłoszeniowej rdzenia wektorowego. Instrukcje do wykonania są wyciągane z VVRS w sposób *out-of-order*.

Jak wspomniano VectorBackend przetwarza operacje arytmetyczno-logiczne, a podczas projektowania modułu wektorowego zdecydowano się na klasyczne podejście, w którym operacje tego typu wykonują się na taśmach z podzielonym plikiem rejestrów. W związku z tym w backendzie wektorowym znajduje się określona na etapie kompilacji liczba VectorExecutorów, z których każdy reprezentuje jedną taśmę i zawiera część pliku rejestrów (schemat VectorExecutora został przedstawiony na rysunku 2.3). VectorExecutor otrzymuje gotowe instrukcje do wykonania z VVRS, a następnie równolegle:

- Określa liczbę elementów długości ELEN do pobrania z odpowiadającego mu fragmentu VRF[7] na podstawie EEW oraz vl.
- Wyznacza maskę poprawnych elementów z ostatniego ELEN (w przypadku gdy EEW $<$ ELEN).
- Określa rejestry używane przez aktualnie przetwarzaną instrukcję.
- Inicjalizuje stan VectorElemsDownloader, VectorExecutionDataSplitter oraz VectorElemsUploader.

Powyższe dane są przekazywane do VectorElemsDownloadera, który pobiera elementy z odpowiednich rejestrów VRF. Pobieranie odbywa się w taki sposób, by zminimalizować liczbę zablokowanych portów w VRF, to znaczy jeśli tylko dwa operandy źródłowe są potrzebne (z maksymalnie 4), to tylko te dwa rejestry zostaną zablokowane przez VectorElemDownloader. Dodatkowo VectorElemsDownloader wykrywa sytuacje, w których dwa operandy są takie same i generuje tylko jedno żądanie odczytu zamiast dwóch. Pobrane krotki z argumentami są przekazywane do VectorExecutionDataSplitter, gdzie są dzielone i uzupełniane danymi stałymi dla aktualnie wykonywanej instrukcji wektorowej:

- Argumenty operacji arytmetyczno-logicznej są przekazywane do VectorAlu wraz z exec_fn opisującym jaka operacja ma zostać wykonana.
- Jeśli operacja używa maski, to v0 jest przekazywane do VectorMaskExtractor, gdzie wybierane są bity stanowiące maskę dla aktualnie wykonywanej operacji (maska ta ma od 1 do 8 bitów w zależności od ELEN i EEW).
- Stara wartość rejestru wektorowego jest przekazywana do VectorUploader, tak by można było zapewnić *mask undisturbed policy*.

[7] ang. Vector Register File.

`VectorAlu` na podstawie `exec_fn` wybiera operację do wykonania i następnie wykonuje ją w sposób SIMD, czyli równolegle na wszystkich elementach zdefiniowanych przez `EEW` znajdujących się w `ELEN`. Tym samym dla przykładowego `ELEN=32` równolegle można przeprowadzić:

- jedną operację na liczbach 32 bitowych,
- dwie operacje na liczbach 16 bitowych,
- cztery operacje na liczbach 8 bitowych.

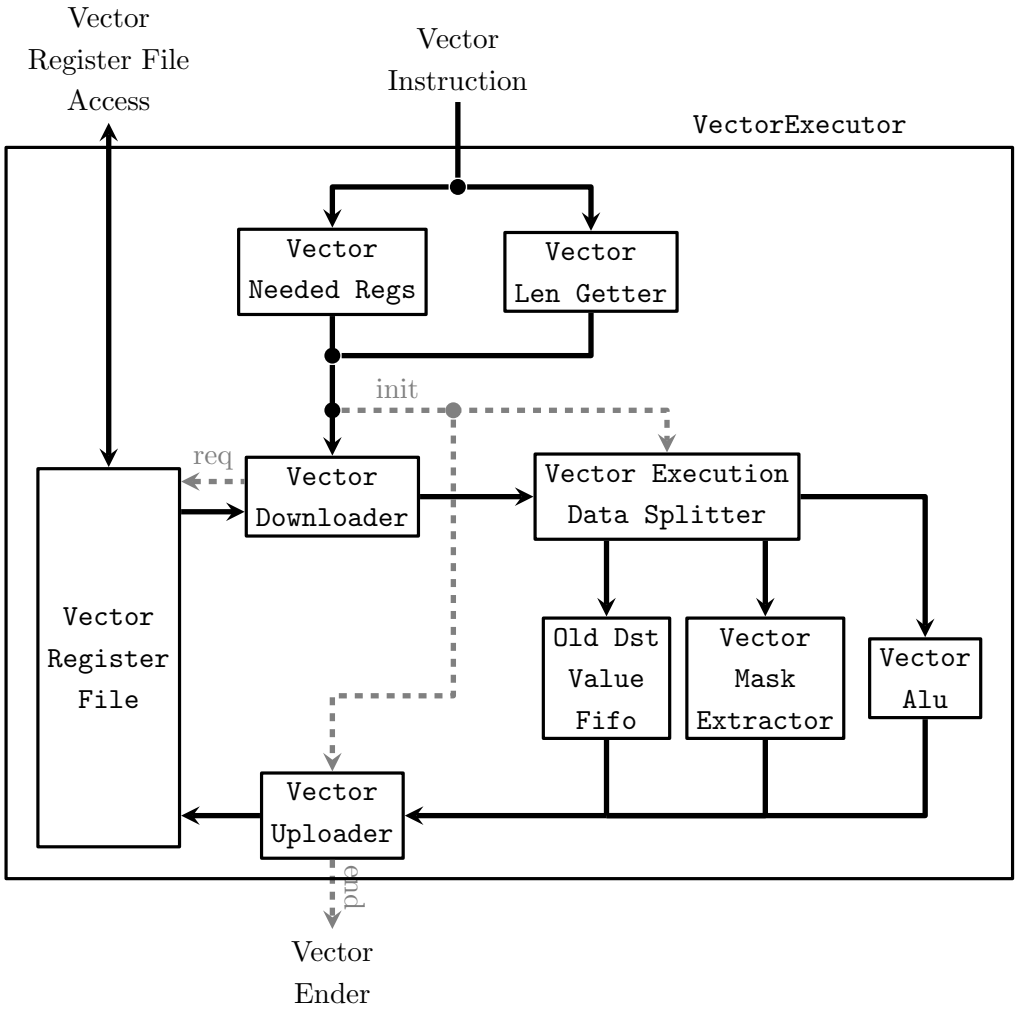
Wyniki z `VectorAlu` są przekazywane do `VectorUploader`, który pobiera maskę z `VectorMaskExtractor` aplikuje ją na wyliczony wynik i uzupełnia nieaktywne elementy starymi wartościami z rejestru docelowego. Tak przygotowana wartość jest następnie zapisywana w VRF. Po tym jak wszystkie elementy z fragmentu VRF odpowiadającego danej taśmie zostaną przetworzone i zapisane w VRF, `VectorElemsUploader` wysyła sygnał o zakończeniu działania do `VectorExecutionEnder`. Blok ten czeka aż wszystkie taśmy zakończą pracę, po czym rozgłasza do `VVRS`-a i scoreboarda, że docelowy rejestr wektorowy jest już gotowy do użytku przez następne instrukcje. Jednocześnie wysyła wiadomość o zakończeniu wykonania do `VectorAnnouncer` – który przesyła ją do rdzenia skalarnego – a także informuje `VectorRetirement`, jaki był `rob_id` oraz docelowy fizyczny rejestr wektorowy zakończonej instrukcji.

Rdzeń skalarny Coreblocksa zatwierdza instrukcje do stanu architektonicznego w sposób *in-order* i każda taka czynność jest poprzedzona rozgłoszeniem aktualnie zatwierdzanego `rob_id` z użyciem metody `precommit`. Na metodzie tej nasłuchuje `VectorRetirement`, który porównuje aktualnie zatwierdzany `rob_id` z listą zakończonych instrukcji wektorowych. Jeśli `rob_id` odpowiada takowej instrukcji, to jest ona usuwana z listy, aktualizowany jest `VRRAT`, a stary rejestr fizyczny z `VRRAT`-a jest dealokowany i udostępniany do powtórnego użycia.

2.5. VectorLSU

Za obsługę instrukcji operujących na pamięci odpowiada `VectorLSU`, który podobnie jak `VectorCore` implementuje interfejs bloku funkcyjnego. W aktualnej implementacji `VectorLSU` wspiera przetwarzanie tylko jednej instrukcji jednocześnie tak samo, jak LSU[8] skalarne zaimplementowane w Coreblocksie. Dzięki takiej decyzji projektowej możliwe jest stworzenie bardzo małego LSU, bowiem nie musi mieć ono struktur służących wdrażaniu efektów ubocznych (czyt. operacji na pamięci) w sposób *in-order*.

[8] ang. Load Store Unit.



Rysunek 2.3: Schemat jednostki `VectorExecutor` będącej częścią modułu wektorowego Coreblocksa.

Podczas implementacji zdecydowano się, by `VectorLSU` było osobnym blokiem funkcyjnym względem skalarnego LSU. Decyzja ta wynikała z chęci zwiększenia modularności kodu i zapobiegnięcia jego duplikacji, która wystąpiłaby gdyby `VectorLSU` wspierało skalarne operacje na pamięci. Jednakże postanowienie to pociągnęło za sobą konieczność zsynchronizowania obu jednostek operujących na pamięci, tak by zachować porządek odczytów i zapisów do pamięci.

Może się bowiem zdarzyć, że najpierw wykonujemy skalarny zapis, a następnie z tego samego adresu chcemy wykonać wektorowy odczyt (lub na odwrót, czyli wektorowy zapis i skalarny odczyt). Odczyty są wykonywane od razu po otrzymaniu instrukcji, natomiast zapisy muszą czekać, aż będzie można je bezpiecznie wdrożyć do stanu architektonicznego. W związku z tym istnieje ryzyko, że odczyt wektorowy wykonywany przez jeden blok funkcyjny pobierze starą wartość z pamięci i nie uwzględni wyniku zapisu skalarnego wykonywanego przez drugi blok funkcyjny.

Połączono więc oba bloki funkcyjne przy pomocy sygnału zajętości zapalanego przez dowolną z tych jednostek w trakcie przetwarzania instrukcji. Sygnał ten blokuje przyjmowanie instrukcji do wykonania przez drugie LSU, które jest i pozostaje bezczynne. Dzięki temu na raz może być przetwarzana tylko jedna instrukcja operująca na pamięci. Tym samym jeśli jest to zapis, to następujący po nim odczyt może się zacząć dopiero po zakończeniu przetwarzania tego zapisu przez LSU, czyli po jego wdrożeniu do pamięci.

2.6. Przepływ instrukcji operujących na pamięci

Wektorowe instrukcje operujące na pamięci są przekazywane przez scheduler skalarny do `VectorLSU` w celu wykonania. Z kolei `VectorLSU` równolegle – w tym samym cyklu co zlecenie – przekazuje tę instrukcję do `VectorFrontend` oraz blokuje wykonanie instrukcji w skalarnym LSU. Następnie instrukcja przebywa całą drogę w `VectorFrontend`zie opisaną w sekcji 2.4.1., aż do momentu w którym jest z powrotem kierowana do `VectorLSU`. W przypadku instrukcji „load” wykonanie zaczyna się od razu po odbiorze przetworzonej instrukcji przez `VectorLSU` (w związku z założeniem, że odczyt z pamięci RAM nie ma efektów ubocznych) i przychodzące dane są wysyłane do odpowiedniej taśmy w celu zapisania ich w pliku rejestrów. Wysłanie to odbywa się poprzez połączenia bezpośrednie (bez żadnej sieci routingów) między `VectorLSU` a `VectorExecutor`.

Obsługa instrukcji „store” jest trochę bardziej skomplikowana, bowiem najpierw należy odczekać na sygnał, że można wykonywać efekty uboczne. Jest on rozgłaszany przez skalarny retirement w postaci metody `precommit`. Gdy istnieje pewność, że dana instrukcja się wykona i jej efekt ma być wdrożony do stanu architektonicznego, wysyłany jest *request* z prośbą o odczytanie danego elementu z rejestru. Następnie `VectorLSU` oczekuje na odpowiedź, która ze względu na konflikty na rejestrach,

może mieć różne opóźnienie. Dopiero po otrzymaniu odpowiedzi zlecany jest zapis do pamięci.

`VectorLSU` korzysta z tej samej implementacji dostępu do magistrali co skalarne LSU, jest więc obarczone tym samym problemem wydajnościowym co opisany w sekcji 2.1., czyli maksymalna przepustowość magistrali wynosi tylko jedną operację na 4 cykle.

2.7. Typy rejestrów

Główną zmianą w skalarnej części Coreblocksa jest dodanie wsparcia dla typów rejestrów. Oryginalna wersja tego procesora wspiera bowiem wyłącznie rejestry `x0-31`, czyli rejestry całkowitoliczbowe. Natomiast dwa rozszerzenia RISC-V – omawiane tutaj rozszerzenie wektorowe `V`, oraz rozszerzenie dodające wsparcie liczb zmiennopozycyjnych `F` – wprowadzają nowe zestawy rejestrów. W związku z tym po dodaniu tych rozszerzeń każdy identyfikator rejestru może w zależności od kontekstu wynikającego z instrukcji, wskazywać na rejestr w jednym z trzech zestawów. Pojawia się więc potrzeba przekazania tego kontekstu z dekodera instrukcji do kolejnych etapów przetwarzania.

W tym celu dodano w wewnętrznym potoku dodatkowe pole enumeracyjne do każdego identyfikatora rejestru. Pole to może przyjąć jedną z trzech wartości: `X`, `F`, `V`. Domyślną jest `X`, które mapuje się na 0. Następnie zmodyfikowano Coreblocksa by rejestry typu `V` były ignorowane przez rdzeń skalarny i przekazywane bez zmian do modułu wektorowego. W tym celu:

- Zmodyfikowano scheduler tak by:
  - Nie alokował rejestru, jeśli docelowy rejestr jest typu `V`.
  - Ignorował rejestry wektorowe podczas przenazywania.
  - Nie próbował odczytywać wartości rejestrów wektorowych z pliku rejestrów skalarnych.
- Dodano pomijanie rozgłaszania gotowości rejestrów wektorowych na szynie skalarnej.
- Zaktualizowano retirement skalarny dodając warunek pozwalający na aktualizację RRAT[9] tylko gdy zatwierdzana jest instrukcja ze skalarnym rejestrem docelowym.

[9] ang. Retirement Register Alias Table.

Wszystkie powyższe zmiany zostały zrobione w sposób uniwersalny. Dzięki temu w sytuacji jeśli przyszła implementacja rozszerzenia `F` w Coreblocksie rozdzieli frontend skalarny od frontendu zmiennopozycyjnego (analogicznie jak to zostało zrobione w niniejszej pracy w przypadku rozszerzenia wektorowego), to nie będzie potrzeby powtórnego modyfikowania rdzenia skalarnego.

2.8. Inne decyzje projektowe

Poza decyzjami implementacyjnymi opisanymi w poprzednich sekcjach, a mających znaczący wpływ na działanie rozszerzenia wektorowego, podjęto także szereg mniej znaczących decyzji.

2.8.1. Implementacja pliku rejestrów

Jak wcześniej opisano, plik rejestrów został podzielony na fragmenty i każdy fragment został przypisany do jednej taśmy. Implementacja jednego fragmentu składa się natomiast ze zbioru banków pamięci typu 1R1W (jeden port do odczytu i jeden port do zapisu). Banków we fragmencie jest tyle ile rejestrów fizycznych i każdy bank jest przypisany na stałe do jednego rejestru fizycznego. Dzięki temu każdy z 5 portów udostępnianych przez VRF (4 porty do odczytu i 1 port do zapisu) może działać niezależnie od pozostałych tak długo, jak odczytuje inny rejestr fizyczny. Każdy rejestr przechowuje maskę ilustrującą, które bajty mają poprawne wartości, a które nie. Podczas odczytu bajty niepoprawne są wypełniane jedynkami. Liczba wektorowych rejestrów fizycznych jest konfigurowalna, z założeniem, że musi ich być więcej niż logicznych.

2.8.2. Zarządzanie rejestrami wektorowymi

Dodanie pliku rejestrów wektorowych pociągnęło za sobą konieczność podjęcia decyzji, gdzie tymi rejestrami zarządzać. We frontendzie skalarnym, czy też w `VectorCore`? Zdecydowano, że sterowanie tymi rejestrami będzie odbywać się wewnątrz jednostki funkcyjnej.

Uzasadnieniem dla takiej decyzji jest to, że w celu określenia liczby rejestrów wektorowych, które potrzebujemy zaalokować, oraz tego, jakie rejestry trzeba poddać przenazywaniu, potrzebna jest znajomość `LMUL` dla danej instrukcji. Pole to jest ustawiane przez poprzedzające instrukcji `vset{i}vl{i}`, natomiast wartość ustawiana przez `vset{i}vl{i}` zależy od zawartości rejestrów skalarnych. Istnieje więc zależność między wynikiem poprzedniej instrukcji, a sposobem alokowania rejestrów dla aktualnej.

Administrowanie rejestrami wektorowymi w `VectorCore` ma więc dwie zalety nad administrowaniem w frontendzie skalarnym. Po pierwsze, jesteśmy w stanie wykonać *forwarding* wyników `vset{i}vl{i}` do aktualnej instrukcji (odbywa się on w `VectorStatusUnit`). Po drugie, gdy wykonanie poprzedniej instrukcji `vset{i}vl{i}` jeszcze się nie zakończyło i przetwarzanie aktualnej musi wykonać *stall*, to nie blokujemy frontendu skalarnego. Instrukcja ta została już bowiem skierowana do `VectorCore`, gdzie może oczekiwać na `LMUL`, a w tym czasie rdzeń skalarny może dekodować i przetwarzać inne instrukcje.

2.8.3. Struktura listy wolnych rejestrów

Ważnym aspektem listy wolnych rejestrów wektorowych jest to, że trzeba wspierać potencjalnie aż 8 alokacji równolegle (w przypadku gdy `LMUL=8`). W związku z tym implementacja listy wolnych rejestrów używana przez skalarną część Coreblocksa, nie nadawała się do ponownego wykorzystania w części wektorowej. Bazuje ona bowiem na kolejce FIFO (ang. First-In First-Out) jednoportowym, co ogranicza przepustowość do nie więcej niż jednej alokacji w cyklu. Potencjalnie można by było uogólnić to rozwiązanie i użyć kolejki FIFO z ośmioma portami, by alokować na raz do 8 rejestrów, jednakże w celu uniknięcia zakleszczeń zarówno wstawianie do tej kolejki jak i wyciąganie z niego musiałoby być idealnie zbalansowane. Oznacza to, że dla każdej pary portów różnica w liczbie wykonanych zapisów (i analogicznie odczytów) musiałaby być co najwyżej 1. Osiągnięcie takiej własności jest trudne ze względu na skomplikowany układ kombinacyjny, który się nie skaluje na liczbę portów większą niż 4.

W związku z tym zdecydowano się użyć listy wolnych rejestrów takiej jak ta zaimplementowana w Xiangshan [Xu et al. 2022], gdzie trzymany jest jeden rejestr, w którym każdy bit symbolizuje, czy rejestr fizyczny jest wolny, czy też nie. Wybór wolnego rejestru w takim przypadku sprowadza się do prostego (choć z relatywnie długą ścieżką krytyczną) dekodera priorytetowego. Takie rozwiązanie można uogólnić na alokowanie większej liczby portów dodając kolejne dekodery, które będą wybierały kolejne wolne rejestry.

2.8.4. Użycie CAM w `VectorRetirement`

Podczas projektowania należało zdecydować w jaki sposób przechowywać informacje potrzebne do zatwierdzenia instrukcji wektorowej do stanu architektonicznego. Rozważano: rozszerzenie ROB-a (ang. Reorder Buffer) skalarnego o możliwość zapisywania translacji z logicznego rejestru wektorowego na fizyczny rejestr wektorowy; zduplikowanie ROB-a i stworzenie osobnego dla instrukcji wektorowych (analogicznie do Vitriuvus+ autorstwa Minervini et al. 2023); oraz użycie CAM (ang. Content Adressable Memory) by mapować `rob_id` z ROB-a skalarnego na dane potrzebne `VectorRetirement`.

Pierwsza opcja byłaby najbardziej oszczędna ze względu na zasoby, jednakże wymagałaby rozszerzenia interfejsu ROB o dodatkowy port, co w przyszłości mogło by utrudnić migrację ROB-a z rejestrów do pamięci. Druga opcja jest nieoptymalna pod względem zasobów, bowiem istniałyby dwie struktury ROB i w obu przechowywana by była instrukcja wektorowa: w ROB-ie skalarnym po to, by zachować kolejność, a w ROB-ie wektorowym by mapować ją na dane. Ostatecznie zdecydowano się na użycie CAM, w którym trzymane są tylko dane zakończonych, a jeszcze nie zatwierdzonych instrukcji.

Rozdział 3.

Optymalizacja

Jak wspomniano we wstępie, rdzeń wektorowy – tak jak i cały procesor Coreblocks – został zaprojektowany z myślą o FPGA. W związku z tym gotową i przetestowaną implementację zsyntezowano z użyciem dwóch toolchainów:

- Yosys + Nextpnr – toolchain *open source*, umożliwiający syntezę na układy FPGA Lattice i ECP5.
- Intel Quartus – toolchain *proprietary* dla układów FPGA wyprodukowanych przez Intela. Jego podstawowa wersja dla małych FPGA jest dostępna bez opłat licencyjnych.

Pierwsze wyniki nie były zadowalające, bowiem z użyciem Yosys-a i Nextpnr, Coreblocks z dodanym modułem wektorowym zawierającym jedną taśmę osiągał 36 MHz częstotliwości zegara oraz zużywał większość zasobów dostępnych na ECP5 (patrz tabela 3.1). Dla porównania konfiguracja Coreblocksa zawierająca wszystkie do tej pory zaimplementowane rozszerzenia poza rozszerzeniem `V`, czyli `rv32ibmc_zcsr` osiąga około 48 MHz, a wersja `rv32i` około 55 MHz. W związku z tym, że różnica była znacząca i nie było możliwości zsyntezowania na docelową FPGA dwóch taśm, zaczęto optymalizować implementację.

	1 taśma	2 taśmy	Dostępne zasoby
Flip-Flop	18634 (22%)	22432 (26%)	83640
Logika kombinacyjna	64605 (77%)	87553 (104%)	83640
RAM	1191 (11%)	1343 (12%)	10455
FMax	36,27 MHz	Brak	N/A

Tabela 3.1: Porównanie wyników syntezy zasobów w niezoptymalizowanej wersji Coreblocksa z modułem wektorowym. Wyniki uzyskano z użyciem Yosys+Nextpnr. W przypadku wersji z dwoma taśmami nie było możliwym wyznaczenie maksymalnej częstotliwości ze względu na większe zużycie zasobów FPGA niż dostępne na ECP5.

3.1. Optymalizacje całego rdzenia

Poszukiwania możliwych optymalizacji zaczęto od analizy ścieżki krytycznej wyznaczonej przez Nextpnr. Okazało się, że nieintencjonalnie pojawiła się zależność kombinacyjna między rozgłoszeniem wyniku przez `VectorLSU`, a wstawianiem nowej instrukcji arytmetyczno-logicznej do `VVRS`. Wyniknęła ona z ustawienia relacji kolejności na portach do zapisu w scoreboardzie przechowującym informację, czy rejestr wektorowy zawiera już poprawny wynik, czy jeszcze nie. Ścieżka krytyczna miała postać następującą:

- `VectorLSU` przeprowadza ostatnią operację na swojej instrukcji i rozgłasza zakończenie pracy.
- Rozgłoszenie o zakończeniu ustawia w scoreboardzie bit informujący, że odpowiadający mu rejestr `v_n1` zawiera poprawne dane.
- Wstawienie do `VVRS` czeka na zakończenie rozgłoszenia (ze względu na relację kolejności na portach w scoreboardzie). Zaznacza następnie, że rejestr docelowy `v_n2` dla aktualnie wstawianej instrukcji nie zawiera jeszcze poprawnych danych.
- Odbywa się zapisanie instrukcji arytmetyczno-logicznej do `VVRS`.

Powyższa ścieżka zależności jest niepotrzebna, bowiem implementacja rdzenia gwarantuje, że jeśli jakiś rejestr wektorowy został już zaalokowany na wynik instrukcji, to tak długo jak jest w użyciu (oczekuje na wdrożeniu do stanu architektonicznego lub jest częścią stanu architektonicznego) nie zostanie on zaalokowany ponownie. W powyższej ścieżce krytycznej zachodzi więc `v_n1 != v_n2`, można więc było usunąć niepotrzebną relację kolejności na portach scoreboarda i tym samym skrócić tę ścieżkę. Zmiana ta spowodowała wzrost maksymalnej częstotliwości do 39 MHz.

Następnym krokiem było zoptymalizowanie ścieżek krytycznych w `BasicFifo`. Struktura ta – zazwyczaj o długości 2 – jest bowiem używana jako bufor między elementami mającymi się wykonywać w kolejnych cyklach. Innymi słowy służy do przecinania ścieżek kombinacyjnych, jest więc na początku i na końcu w zasadzie każdej ścieżki. W oryginalnej implementacji używano pamięci z kombinacyjnym odczytem, a część logiki znajdowała się pod niepotrzebnymi multiplekserami. Wynikało to z tego, że `BasicFifo` było pisane z użyciem starszej wersji biblioteki Transactron, która nie miała wsparcia dla domen `top_comb` i `av_comb`.[1] Zaktualizowano więc implementację, tak by możliwie jak najwięcej sygnałów było ustawianych w `top_comb`.

[1] W bibliotece Amaranth implementowane są dwie standardowe domeny zegarowe. W domenie `comb` znajdują się operacje kombinacyjne (dziejące się w ramach jednego cyklu zegarowego). Natomiast w domenie `sync` umieszczane są operacje synchroniczne, czyli dziejące się na zadanym zboczu sygnału zegarowego. Domeny `top_comb` i `av_comb` są domenami specyficznymi dla Transactrona, w których umieszczane są operacje kombinacyjne: w domenie `top_comb` uruchamiane są zawsze i ignorują wszelkie warunki aktywacji; natomiast te w domenie `av_comb` ignorują wewnętrzne warunki

Zauważono także, że w przypadku FIFO zawsze wiemy jaki element będzie odczytywany w następnym cyklu. Jeśli w aktualnym cyklu nie jest prowadzony odczyt, to w następnym cyklu należy przeczytać ten sam element, natomiast w przeciwnym razie należy przeczytać następny element w kolejce. Ta obserwacja pozwala wyliczyć adres odczytu cykl wcześniej, co umożliwia użycie pamięci z synchronicznym odczytem. Pamięci synchroniczne są natomiast tańszym i szybszym zasobem na FPGA. Optymalizacja ta spowodowała wzrost maksymalnej częstotliwości do 41 MHz.

Wprowadzono także kilka mniejszych zmian, mających na celu skrócenie ścieżek krytycznych, m.in. dodano bufor wejściowy w skalarnym ALU (ang. Arithmetic Logic Unit), aby rozdzielić logikę wyboru z RS instrukcji do wykonania od operacji arytmetyczno-logicznych; oraz odseparowano logikę wyboru instrukcji z `VVRS` od wstępnego przetwarzana tej instrukcji w `VectorExecutor`, rozbijając je na dwa osobne cykle.

W ramach optymalizacji ścieżek krytycznych zmieniono także implementację dodawania w `VectorAlu`. W pierwszej wersji redukowano liczbę używanych operacji dodawania, przy jednoczesnym zapewnieniu wymaganego wsparcia dla dodawania równoległego wszystkich elementów znajdujących się w `ELEN`. Stworzono strukturę drzewiastą u podstaw której znajdowało się `ELEN`/8 sumatorów 8-bitowych, które dodawały do siebie kolejne bajty z `ELEN`. Następnie w zależności od wybranego `EEW` do bardziej znaczących bajtów był dodawany bit przeniesienia z mniej znaczących bajtów. W związku z tym dla `ELEN=32` używano tylko 4 sumatorów 8-bitowych i 3 inkrementatorów (dwa 16-bitowe i jeden 32-bitowy). Jednakże ścieżka krytyczna biegła przez sumator najmniej znaczących bajtów i dwa inkrementatory.

Podczas analiz okazało się, że taki projekt jest nieefektywny na FPGA, gdzie sumatory są implementowane przy pomocy gotowych bloków sprzętowych, natomiast połączenia między nimi są implementowane programowo. Dlatego zamiast powyższej implementacji zdecydowano się użyć wielu sumatorów – po jednym dla każdej pary elementów, dla każdego z `EEW` – które będą działały równolegle. Wynik ostateczny jest wybierany na podstawie `EEW` przez multiplekser do którego połączone są wszystkie sumatory.

W oczywisty sposób skraca to ścieżkę krytyczną o programowalną logikę przesyłania danych między sumatorami i inkrementatorami, jednakże wydawać by się mogło, że odbędzie się to kosztem większego zużycia zasobów ze względu na większą liczbę sumatorów (7 sumatorów o łącznej szerokości 96 bitów). Tymczasem zaobserwowano spadek zużycia zasobów używanych przez układ dodający o 38%. Uzasadnić to można tym, że na FPGA sumatory są tańsze niż logika związana z przesyłaniem danych między nimi. Optymalizacja ta wraz z przecięciem ścieżek krytycznych w `VectorExecutor` i w ALU skalarnym, pozwoliła zwiększyć maksymalną częstotliwość do około 45 MHz.

---

aktywacji Transactrona. Pomijanie części warunków aktywacji ma pozytywny wpływ na długość ścieżek krytycznych.

3.2. Optymalizacje pliku rejestrów wektorowych

Optymalizując rdzeń wektorowy szczególnie dużo uwagi poświęcono wektorowemu plikowi rejestrów. Jego logika kombinacyjna zużywała bowiem 24,4 tys. LUT[2], podczas gdy cały moduł wektorowy zużywał 30 tys. LUT, a cały procesor 40 tys. LUT. Tak duży bok sprzętowy ma znaczący wpływ nie tylko na liczbę taśm, które można umieścić w rdzeniu, ale także na długość ścieżek krytycznych. Ścieżki te mogą być bowiem niepotrzebnie wydłużane ze względu na brak wolnych lokalnych zasobów. W celu wygenerowania informacji o zużyciu zasobów wykorzystano toolchain Intel Quartus[3].

Pierwszą i najłatwiejszą zmianą było usunięcie nadmiarowych warstw abstrakcji z banków tworzących plik rejestrów. W domyślnej konfiguracji znajduje się 40 rejestrów fizycznych, więc zgodnie z podjętymi decyzjami projektowymi przekłada się to na 40 banków 1R1W znajdujących się w pliku rejestrów. W związku z tym nawet minimalny narzut związany z abstrakcją wewnątrz jednego banku staje się kosztowny, albowiem jest mnożony razy 40.

Początkowa implementacja banków w wektorowym pliku rejestrów korzystała z abstrakcji nad pamięcią dostarczanej przez Transactrona. Abstrakcja ta utrzymywała względem siebie porządek zapisów i odczytów, jednocześnie udostępniając interfejs niewrażliwy na opóźnienia. Koszt pojedynczej instancji takiej abstrakcji związany z logiką kombinacyjną to 100 LUT, wydaje się więc niewielki. Jednakże przy 40 instancjach sumaryczny narzut wynosi 4 tys. LUT, czyli 10% zasobów zużytych przez cały procesor. Usunięto więc te abstrakcje z implementacji i zamiast tego dostępy do pamięci zostały zrealizowane bezpośrednio w banku pliku rejestrów.

Dodatkowo sprawdzono trzy różne implementacje łączenia portów stanowiących interfejs VRF z portami poszczególnych banków:

- bez buforowania;
- z buforem dwuelementowym na każdym połączeniu;
- z buforem dwuelementowym dla każdego portu VRF.

Jak można się spodziewać w przypadku bez buforowania (który stanowił początkową implementację) powstają dłuższe ścieżki krytyczne i bardziej skomplikowana logika kombinacyjna, niż w pozostałych dwóch wersjach. Jednakże wybór pomiędzy

[2] ang. Lookup Table – podstawowa jednostka budująca układy kombinacyjna na FPGA.

[3] Należy jednak zaznaczyć, że wyniki uzyskiwane z użyciem Intel Quartus nie pokrywają się z w pełni tymi uzyskiwanymi przy pomocy narzędzi *open source*, ze względu na różne architektury FPGA. W szczególności ten sam opis układu – syntezowany w Intel Quartus na Cyclone V – uzyskuje maksymalną częstotliwość taktowania w pesymistycznych warunkach o 15-20 MHz większą (60-70 MHz vs 40-50 MHz) niż syntezowany w Yosys+Nextpnr na ECP5. Jeśli nie zaznaczono inaczej to raportowane w tej pracy częstotliwości są uzyskane z użyciem toolchainu Yosys+Nextpnr.

pozostałymi dwoma sposobami łączenia nie jest oczywisty, bowiem w zależności od używanego toolchaina uzyskiwane wyniki są znacząco różne.

Intel Quartus lepiej radzi sobie z buforowaniem na każdym połączeniu i dla takiego układu uzyskuje 66 MHz, natomiast zmiana implementacji na bufory dwuelementowe dla każdego portu VRF powoduje spadek maksymalnej częstotliwości do 62 MHz. Natomiast w przypadku Yosys+Nextpnr ta sama zmiana powoduje wzrost z 41 MHz do 43 MHz.

Ostatecznie zdecydowano się na użycie implementacji z buforami dla portów VRF, bowiem Yosys+Nextpnr jest głównym toolchainem używanym w Coreblocks, więc przede wszystkim interesujące są wyniki tych narzędzi. Dodatkową zaletą takiego rozwiązania jest mniejsze zużycie zasobów i lepsza skalowalność ze względu na liniową, a nie kwadratową, liczbę buforów. Powyższe zmiany pozwoliły zredukować zużycie logiki kombinacyjnej w VRF z 24,4 tys. LUT do około 8 tys. LUT.

Analizując VRF starano się także skrócić ścieżki krytyczne biegnące przez tę strukturę, bowiem ze względu na swoje skomplikowanie ograniczały one częstotliwość maksymalną procesora. Na skutek tych prac wykryto ścieżkę kombinacyjną na sygnałach sterowania biegnącą od portów wejściowych VRF do portów wyjściowych VRF. Powstała ona pomimo tego, że dane potrzebowały co najmniej 3 cykli by przebyć tę drogę. Dlatego zdecydowano się dodać dodatkowy bufor wewnątrz banku pliku rejestrów, którego celem miało być rozcięcie tej ścieżki. Zmiana ta przyniosła bardzo dobre rezultaty, bowiem udało się zwiększyć częstotliwość aż do 51 MHz. Tym samym dla Coreblocksa z rozszerzeniem „V” (`rv32i_zve32x`) osiągnięto znacząco lepszą częstotliwość maksymalną niż dla konfiguracji ze wszystkimi modułami poza wektorowym (`rv32icbm`), dla której częstotliwość maksymalna wynosi 48 MHz.

	1 taśma	2 taśmy	full	Dostępne zasoby
Flip-Flop	17440 (20%)	20520 (24%)	8169 (9%)	83640
Logika kombinacyjna	55866 (66%)	74003 (88%)	26616 (31%)	83640
RAM	1434 (13%)	1829 (17%)	263 (2%)	10455
FMax	51,29 MHz	49,55 MHz	47,98 MHz	N/A

Tabela 3.2: Porównanie wyników syntezy po optymalizacjach z użyciem Yosys+Nextpnr na FPGA ECP5 dla trzech konfiguracji Coreblocksa: z jedną taśmą wektorową, z dwoma taśmami wektorowymi oraz dla konfiguracji ze wszystkimi jednostkami funkcyjnymi poza „V” (konfiguracja „full”).

3.3. Wpływ Transactrona na wyniki rdzenia

Transactron będąc wysokopoziomową biblioteką do opisu układów ukrywa dużą część trudności za warstwami abstrakcji, jednakże każda abstrakcja wiąże się z jakimś kosztem. Jest on związany albo z dodatkowymi elementami zapewniającymi odpowiedni interfejs (np. narzut powierzchniowy związany z korzystaniem z interfejsu transakcyjnego do pamięci w wektorowym pliku rejestrów) albo wynika on z utraconych szans na optymalizację.

Ważnym wnioskiem z prac nad rdzeniem wektorowym jest to, że transakcje symultaniczne w Transactronie w aktualnej formie nie nadają się do pisania kodu z wieloma równoczesnymi transakcjami, ze względu na złą skalowalność. Takie transakcje udostępniają semantykę znaną ze zwykłych języków programowania – metoda może być wołana pod ifem i jeśli warunek w ifie będzie fałszywy, to nie będzie żadnych efektów ubocznych wynikających z ciała ifa[4]. Jednakże ta semantyka jest okupiona złożonością, która w pesymistycznym przypadku może być wykładnicza względem liczby wyrażeń warunkowych. Są one bowiem zamieniane na transakcje, z których wybierany jest podzbiór do wykonania spełniający warunki i nie konfliktujący ze sobą. Każdy taki podzbiór jest reprezentowany przez jedną transakcję klasyczną.

Powyższy wniosek jest wynikiem prac nad pierwszą wersją pliku rejestrów wektorowych, która korzystała właśnie z transakcji symultanicznych w postaci wyrażeń warunkowych. Okazało się, że układ generowany dla 40 banków i 4 portów był tak duży, że środowisko symulacji nie było w stanie go zainicjalizować w rozsądnym czasie. Zrezygnowano więc z transakcji symultanicznych w standardowym kodzie, a zamiast nich użyto normalnych, z poprzedzającym je buforem dwuelementowym. Taka konstrukcja udostępnia podobną semantykę wyrażeń warunkowych jak transakcje symultaniczne, niemniej kosztem jednego cyklu opóźnienia.

Pomimo tego, że używając Transactrona należy wystrzegać się pułapek, to należy nadmienić, że znacząco ułatwia on optymalizację kodu. Oferuje bowiem odporność na opóźnienia metod, dzięki czemu przecięcie ścieżki krytycznej wprowadza tylko lokalne zmiany i nie wpływa na kod układów używających danego modułu. Bardzo dobrze obrazują to zmiany – przedstawione w kodzie 3.1 – jakie były potrzebne by rozciąć ścieżkę krytyczną w `VectorExecutor` na dwa osobne cykle.

Cała modyfikacja ograniczyła się do zamiany `Connect` – korzystającego z transakcji symultanicznych, by wywołać dwie transakcje w tym samym cyklu – na `BasicFifo`, które buforuje argumenty wyjściowe z pierwszej transakcji i przekazuje je do drugiej transakcji w następnym cyklu. Interfejsy obu tych struktur są

[4]Warto nadmienić, że ta własność nie zachodzi dla normalnych transakcji Transactrona, bowiem występuje efekt uboczny w postaci zablokowania wywołania metody – żadna inna transakcja nie może jej w tym samym cyklu użyć.

```
@@ -118,7 +118,7 @@ class VectorExecutor(Elaboratable):
  )
  uploader = VectorElemsUploader(self.gen_params, self.write_vrf,
                 old_dst_fifo.read, mask_out_fifo.read, self.end)

- issue_connect = Connect(self.layouts.executor_in)
+ issue_connect = BasicFifo(self.layouts.executor_in, 2)
  self.issue.proxy(m, issue_connect.write)
  self.read_req.proxy(m, serializers[2].serialize_in[1])
  self.read_resp.proxy(m, serializers[2].serialize_out[1])
```

Kod źródłowy 3.1: Sformatowany dla lepszej czytelności diff z komita `911465ad7379abbf326ddf2aa040eb99fd7518e6` przedstawiający zmiany potrzebne by przeciąć ścieżkę krytyczną w `VectorExecutor` i rozdzielić ją na dwa osobne cykle.

takie same, a odporność na opóźnienia zapewniana przez Transactrona zapewnia ich wymienność.

Łatwość wprowadzania zmian w Transactronie można porównać z biblioteką Chisel, która także jest biblioteką wyższego poziomu do opisu sprzętu, jednakże napisaną w Scali. Podczas prac nad Hwacha implementowanej w Chiselu, autorzy potrzebowali ok. 100 iteracji i 7 dni pracy [Lee et al. 2015], aby zwiększyć częstotliwość taktowania o ok. 50%. Dla porównania ten sam uzysk w rdzeniu wektorowym zaimplementowanym w Coreblocksie uzyskano po około 25 iteracjach, których przeprowadzenie zajęło 2 dni.

Rozdział 4.

Podobne prace

Współczesne publikacje związane z procesorami wektorowymi można podzielić na dwie grupy – stworzone przed 2019 rokiem i na te stworzone po nim. Wynika to z faktu, że 13 czerwca 2019 roku została opublikowana pierwsza publiczna wersja specyfikacji rozszerzenia V do RISC-V[1]. Prace wydane przed opublikowaniem RVV były nieliczne i tworzyły własne architektury. Przykładami mogą być VESPA [Yiannacouras, Steffan, and Rose 2008], VEGAS [Chou et al. 2011] czy też Hwacha [Lee et al. 2014], natomiast prawie wszystkie prace po 2019 roku (wyjątkiem są kolejne wersje Hwacha) implementują którąś z wersji RVV.

Akceleratory opublikowane na początku lat dwutysięcznych celowały w FPGA, w związku z ówcześnie bardzo wysokim kosztem projektowania własnych układów ASIC. VESPA [Yiannacouras, Steffan, and Rose 2008] implementuje architekturę VIRAM w sposób *in-order* bazując na pamięciach wbudowanych w FPGA. Instrukcje wektorowe są tłumaczone w locie na instrukcje SIMD i wykonują się równolegle na taśmach. Później VESPA została rozszerzona o wsparcie chainingu i bankowane rejestry.

Wśród procesorów wektorowych z pewnością wyróżnia się VEGAS [Chou et al. 2011], który został zaprojektowany jako rozszerzenie wektorowe Nios II na FPGA. Autorzy tej pracy zdecydowali się przeznaczyć prawie całą pamięć dostępną w FPGA na plik rejestrów, który jest widziany jako ciągła pamięć, a każdy wektor jest opisany przez wskaźnik na początek wektora jak i jego długość. Tym samym programista dostał do ręki bardzo elastyczne narzędzia, gdyż mógł samodzielnie zarządzać konstrukcją pliku rejestrów. Z drugiej strony elastyczność ta jest okupiona tym, że kod trzeba pisać w asemblerze.

Procesory wektorowe stworzone po 2019 są zazwyczaj bardzo podobne do siebie – implementują RVV i różnią się detalami. Wśród nich można wyróżnić Arę [Perotti et al. 2022], która w najnowszej wersji implementuje bardzo szeroki podzbiór instrukcji z RVV 1.0. Opiera się na taśmach, a każda z nich zawiera 8 banków

[1] `https://github.com/riscv/riscv-v-spec/releases/tag/0.7.1`

1RW tworzących fragment pliku rejestrów. Wyjątki są zgłaszane przed rozpoczęciem wykonania instrukcji lub w bloku odpowiedzialnym za komunikację z pamięcią.

Vitriuvus+ [Minervini et al. 2023] jest z kolei implementacją *out-of-order* rozszerzenia RVV, która komunikuje się z procesorem skalarnym z użyciem *Open Vector Interface*. Tym samym jest zgodna z założeniami konkursu organizowanego przez Unię Europejską na najwydajniejszy akcelerator wektorowy.

Vitriuvus+ podobnie jak Ara używa banków 1RW, jednakże na wszystkich wykonuje równolegle tę samą operację. To znaczy: najpierw odczytuje równolegle z każdego banku pierwszy operand, później drugi i trzeci, a na końcu zapisuje do nich wynik. Vitriuvus+ posiada 5 banków pamięci, więc po trzech operacjach odczytu powstaje 5 krotek operandów, które produkują 5 wyników zapisywanych w jednym cyklu. Wartość ta jest tak dobrana, by zgadzała się z liczbą stanów maszyny kontrolującej operacje na pliku rejestrów, która poza odczytami i zapisami operandów ma jeszcze piąty stan na obsługę operacji na pamięci (zapis danych pochodzących z LOAD, albo odczyt danych dla STORE).

Dodatkowo Vitriuvus+ implemetuje kilka innych ciekawych optymalizacji. Przykładowo używanie przenazywania w procesorze wektorowym pozwala mu na bardzo efektywne kopiowanie rejestrów wektorowych, poprzez inkrementację licznika referencji rejestru fizycznego i zmapowanie docelowego adresu logicznego, na źródłowy adres fizyczny. W niektórych przypadkach redukuje to liczbę wykonywanych instrukcji wektorowych o ponad 25%.

W Vitriuvus+ przeanalizowano także temat sieci połączeń między taśmami, która musi łączyć każdą taśmę z każdą, aby zapewnić obsługę instrukcji redukcji. Sieć ta jest bowiem elementem, który znacząco utrudnia skalowalność (przykładowo w pierwszej wersji Ary konfiguracja z 16 taśmami była o ponad 20% wolniejsza niż konfiguracja z 2 taśmami). Najefektywniejszym okazało się połączenie taśm w topologii pierścienia, który na początku wykonywania instrukcji jest konfigurowany do przesyłu danych w lewo bądź w prawo (użyty pierścień jest więc typu *half-duplex*).

Zarówno Vitriuvus+ jak i Ara, to procesory stworzone przez duże zespoły finansowane w ramach grantów z Unii Europejskiej w związku z działaniami prowadzonymi w celu uniezależnienia dostaw układów scalonych do Europy od państw ościennych (European High Performance Computing Joint Undertaking). Projekty te są więc rozbudowane i dopracowane. Można jednakże znaleźć także kilka mniejszych procesorów wektorowych, wśród których na uwagę zasługuje Vicuna [Platzer and Puschner 2021], która stawia za cel przewidywalność czasu w którym skończą się obliczenia, tak by można było takiego procesora użyć w miejscach gdzie występują krytyczne obliczenia czasu rzeczywistego; czy też Risc-V^2 [Patsidis et al. 2020], który jest jednym z pierwszych podejść do wektorowych procesorów *out-of-order*.

Rozdział 5.

Porównanie wydajności zaimplementowanego rozszerzenia

Przygotowaną implementację rozszerzenia wektorowego przetestowano z użyciem benchmarków w celu określenia wydajności otrzymanego modułu. Dla porównania wybrano dojrzały rdzeń Ara – rozwijany od ponad 4 lat przez ETH Zurich – który implementuje RVV 1.0 i został stworzony na bazie procesora *in-order* CVA6. Kod źródłowy Ary jest open source[1], co umożliwiło jego przetestowanie z użyciem własnych benchmarków. Należy jednakże zaznaczyć, że Ara została zaprojektowana z myślą o ASIC, a nie o FPGA, więc porównanie pod względem wyników syntezy może nie być uczciwe[2].

5.1. Czas wykonania przykładowych programów

Ponieważ zakres instrukcji wektorowych, które zaimplementowano w Coreblocksie jest ograniczony, to nie było możliwym zastosowanie wcześniej istniejących benchmarków. W związku z tym stworzono cztery nowe benchmarki, których rdzenie (ang. kernel) zaimplementowano w asemblerze, a następnie przeprowadzono jego inlining do kodu w C. Tak przygotowane programy skompilowano i uruchomiono na procesorach z użyciem wcześniej istniejącej dla nich infrastruktury, a jako symulatora użyto Verilatora. Kody źródłowe rdzeni benchmarków zostały przedstawione w dodatku B, natomiast wyniki dla różnych konfiguracji przedstawiono w tabeli 5.1.

Wyniki Coreblocksa na tle Ary są rozczarowujące: w benchmarkach opartych na operacjach arytmetyczno-logicznych radzi sobie średnio 3-4 razy gorzej, natomiast

[1] `https://github.com/pulp-platform/ara`

[2] W trakcie pisania pracy autorowi nie udało się znaleźć żadnego dojrzałego modułu wektorowego do RISC-V zaprojektowanego na FPGA.

Konfiguracja	vadd	vadd-lot-of-scalars	vadd-mem	vmem
Coreblocks 1-taśma	2542	2571	10551	24306
Ara 1-taśma	-	-	-	-
Coreblocks 2-taśmy	1726	1755	8119	23506
Ara 2-taśmy	552	568	2286	3036
Coreblocks 4-taśmy	1318	1347	6903	23106
Ara 4-taśmy	384	562	1617	2024
Coreblocks 8-taśm	1114	1277	6295	22906
Ara 8-taśm	262	558	1264	1564
Coreblocks 16-taśm	1054	1271	5991	22806
Ara 16-taśm	259	558	1112	1359

Tabela 5.1: Liczba cykli potrzebnych procesorom Ara i Coreblocks na wykonanie benchmarków: „vadd", „vadd-lot-of-scalars", „vadd-mem", „vmem". Oba procesory zostały skonfigurowane z VLEN=1024. Ara nie wspiera configuracji z 1 taśmą, więc niemożliwe było uzyskanie dla niej wyników.

w przypadku benchmarku „vmem" różnica wydajności na korzyść Ary jest około dziesięcio-, piętnastokrotna.

W przypadku „vmem" można było się spodziewać takiego wyniku, ze względu na ograniczenia magistrali dostępu do pamięci o których pisano w sekcji 2.1. W związku z nimi `VectorLSU` jest w stanie przeprowadzić tylko jedną 32-bitową operację w ciągu 4 cykli, natomiast Ara używa magistrali AXI o zmiennej szerokości opisanej wzorem `32 * liczba_lane`, dzięki czemu jest w stanie pobierać dane cykl po cyklu operując na raz na większych pakietach niż Coreblocks.

Natomiast słabe wyniki operacji arytmetyczno-logicznych są wynikiem połączenia kilku aktualnych ograniczeń Coreblocksa i modułu wektorowego:

- Coreblocks nie ma zaimplementowanej spekulacji skoków i na instrukcjach skoku czeka, aż wszystkie wcześniejsze instrukcje w porządku programu zostaną wdrożone do stanu architektonicznego - znacząco ogranicza to zyski z wykonania *out-of-order*.

- Moduł wektorowy ma bardzo duże opóźnienie od otrzymania instrukcji od rdzenia skalarnego do rozpoczęcia jej wykonywania w `VectorAlu`, wynosi ono około 12 cykli, a z powodu braku spekulacji opóźnienie to nie jest ukrywane podczas wykonania kodu.

- Instrukcje wektorowe oczekujące na operand skalarny mogą zablokować inne instrukcje wektorowe w `VXRS` – które potencjalnie mogą się od razu wykonać – co ogranicza zyski z wykonania *out-of-order*.

Jednocześnie pomimo tego, że Ara jest rozszerzeniem do procesora *in-order*, to została zaimplementowana tak, by nie blokować wykonania instrukcji skalarnych. Innymi słowy mówiąc, instrukcje wektorowe – inne niż zwracające skalar i operujące na pamięci – są wdrażane do stanu architektonicznego po sprawdzeniu poprawności operandów, a zanim rozpocznie się ich właściwe wykonywanie w jednostkach funkcyjnych. Dzięki temu pomimo bycia *in-order* Ara jest w stanie ukryć opóźnienia związane z instrukcjami wektorowymi wykonując w tym samym czasie instrukcje skalarne. Dodatkowo Ara posiada bufor na instrukcje wektorowe do wykonania, więc może zgłosić do rdzenia skalarnego informację o tym, że instrukcja jest już wykonana pomimo tego, że kilka wcześniejszych instrukcji wektorowych nie skończyło jeszcze swojej pracy.

5.2. Zużywane zasoby sprzętowe

Zdecydowano się także porównać zasoby wymagane do syntezy przez Arę i Coreblocksa z modułem wektorowym. Chociaż – jak już wspomniano – nie jest to miarodajne porównanie, ze względu na to, że Ara jest zoptymalizowana z myślą o syntezie na krzem, natomiast Coreblocks celuje w układy FPGA.

Jako platformę na której zostaną porównane wyniki syntezy zdecydowano się wybrać syntezę do krzemu, ponieważ:

- Była to dobra okazja by sprawdzić, czy Coreblocksa można zsyntezować do krzemu, gdyż nikt wcześniej tego nie próbował.
- Wyniki syntezy dla Ary są opisane w publikacji [Perotti et al. 2022].
- Kod Veriloga generowany przez bibliotekę Amaranth opisujący Coreblocksa jest bardziej przenośny i kompatybilny z większą liczbą narzędzi, niż kod System Veriloga 2007, w którym jest zaimplementowana Ara.

Coreblocks został zsyntezowany z użyciem toolchaina OpenRoad Flow Scripts, który jest zbiorem narzędzi o otwartym kodzie źródłowym pozwalającym przeprowadzić syntezę układu od Veriloga aż do bramek w krzemie. Podczas syntezy użyto w większości domyślnych parametrów OpenRoad Flow Scripts, niemniej:

Wykorzystanie powierzchni – skonfigurowano jako średnie poprzez ustawienie `PLACE_DENSITY=0.60` oraz `CORE_UTILISATION=40`.

Częstotliwość zegara – jako cel syntezy ustawiono 100 MHz.

Proces technologiczny – nangate45, czyli proces o otwartej licencji, stworzony na potrzeby badawcze, jednakże nie implementowany przez jakąkolwiek fabrykę układów scalonych.

Przedstawione wyniki syntezy Ary pochodzą z pracy [Perotti et al. 2022] i zostały uzyskane z użyciem toolchaina Synopsys, który choć ma zamknięty kod źródłowy, to dostarcza więcej funkcjonalności i optymalizacji niż narzędzia *open source*.

	Coreblocks	Ara
Proces technologiczny	Nangate45	Global Foundries 22FDX FD-SOI
Gate equivalent [μm^2]	0.798	0.199
Powierzchnia [mm^2]	2.69	0.81
kGE	3380	4070
Częstotliwość [MHz]	210	1340

Tabela 5.2: Porównanie wyników syntezy na krzem. Wyniki procesora Ara zostały opracowane na podstawie [Cavalcante et al. 2020; Perotti et al. 2022], natomiast Coreblocks został zsyntezowany z użyciem toolchainu open source OpenRoad Flow Scripts. Oba procesory zostały skonfigurowane z `VLEN=4096` oraz z 4 taśmami.

Wyniki syntezy przedstawiono w tabeli 5.2, gdzie przeliczono otrzymaną powierzchnię na jej ekwiwalent w bramkach dla danego procesu, tak by można było porównać wyniki. Coreblocks zajmuje mniejszą powierzchnię niż Ara (3380 kGE vs 4070 kGE), jednakże należy wziąć poprawkę na to, że nie implementuje on operacji zmiennopozycyjnych i mnożenia, które zajmują znaczącą powierzchnię Ary (55%). Z drugiej strony Coreblocks nie został zoptymalizowany na krzem, w związku z tym używa pamięci 1R1W, które są na FPGA tanie, jednakże na krzemie zajmują znacząco więcej miejsca niż pamięci 1RW (ponad 50% więcej według Minervini et al. 2023).

Typ pamięci wybranej do implementacji VRF wpływa także negatywnie na uzyskiwane ścieżki krytyczne, co przekłada się na mniejszą częstotliwość maksymalną. Zgodnie z Minervini et al. 2023 pamięci 1R1W mają nawet dwukrotnie dłuższą ścieżkę krytyczną niż pamięci 1RW.

Rozdział 6.

Możliwości dalszego rozwoju

Podczas implementacji, ze względu na ograniczenia czasowe, zrezygnowano z części potencjalnych usprawnień i nowych funkcjonalności. Praca ta prezentuje pierwsze podejście do implementacji rozszerzenia wektorowego w procesorze Coreblocks, co spowodowało, że nie ustrzegła się pewnych błędów projektowych. Ich naprawa będzie priorytetem w następnych rewizjach tego modułu.

6.1. Błędy projektowe

Na etapie planowania modułu wektorowego popełniono dwa znaczące błędy projektowe. Po pierwsze zdecydowano się, że każdy rejestr fizyczny będzie osobnym bankiem, a po drugie zaimplementowano wektorowy RS w sposób ograniczający możliwości wykonania instrukcji *out-of-order*.

6.1.1. Architektura VRF

Implementacja w której każdy wektorowy rejestr fizyczny jest osobnym bankiem, jest bardzo przystępna do stworzenia. System adresowania jest prosty, a konflikty w odczytach z banków występują wyłącznie jak czytamy z tego samego rejestru. Jednakże powoduje to konieczność stworzenia skomplikowanej sieci routującej dane między portami. Plik rejestrów mający 5 portów do komunikacji z resztą rdzenia (4 porty do odczytu i jeden do zapisu) potrzebuje połączyć każdy port wejściowy z każdym bankiem.

Przykładowo dla domyślnej konfiguracji 40 rejestrów fizycznych (użytej zarówno w Coreblocksie jak i w Vitriuvus+ [Minervini et al. 2023]) daje to w przybliżeniu $40 + 4 * 40 * 2 = 360$ różnych połączeń (mnożnik 2 wynika z tego, że każdy port do zapisu ma jedno połączenie na zlecenie odczytu, a drugie połączenie na przekazanie wartości). Jeśli weźmiemy jeszcze pod uwagę, że średnio połowa z tych połączeń ma

szerokość większą niż `ELEN`, to okaże się, że cała sieć routingu ma kilka tysięcy połączeń. Do nich z kolei należy dodać w odpowiednich miejscach bufory i multipleksery jeszcze bardziej ją komplikujące.

W związku z powyższym należy dążyć do ograniczenia liczby połączeń. Można to robić w dwojaki sposób: ograniczając porty wejściowe do VRF, albo liczbę banków. W celu ograniczenia portów wejściowych można zauważyć, że jednym z operandów dla instrukcji wektorowych jest zawsze jeden i ten sam rejestr – `v0` – z którego potencjalnie pobierana jest maska dla operacji wektorowej. Tym samym wydzielenie z pliku rejestru specjalnej struktury do przechowywania maski pozwoli zredukować liczbę portów wejściowych VRF o jeden. Jednakowoż spowoduje to, że rdzeń wektorowy będzie musiał potencjalnie przed niektórymi instrukcjami kopiować dane ze standardowego pliku rejestrów do wydzielonego rejestru `v0`, co może spowodować zwiększenie opóźnienia wykonania instrukcji.

Drugi sposób optymalizacji, czyli redukcja liczby banków, został wykorzystany w procesorze Ara, gdzie 32 rejestry fizyczne zostały kolejno przydzielone do 8 banków pamięci. Tym samym zwiększono prawdopodobieństwo konfliktów w dostępie do banków. Należy jednakże zaznaczyć, że im większe `LMUL`, tym ta szansa jest mniejsza, bowiem dzięki temu, że rejestry są zmapowane kolejno do banków pamięci, to po wydłużeniu i sklejeniu sąsiednich rejestrów cały czas mapują się na minimalną możliwą liczbę banków (dla `LMUL<8` to jest jeden bank, a dla `LMUL=8` każdy rejestr zużywa w pełni dwa banki).

Użycie mniejszej liczby banków, tak jak to robi Ara, ma jeszcze jedną wadę, gdyż należy dodać bufory służące do buforowania operandów i zapytań na wypadek konfliktów w dostępie do banków. Z drugiej strony w aktualnej implementacji rdzenia wektorowego Coreblocksa także występują takie bufory, jednakże używane w celu przecinania ścieżek krytycznych. W związku z tym przejście na architekturę VRF, wywodzącą się z Ary, powinno być możliwe bez wprowadzania dodatkowych nieoptymalności.

6.1.2. Architektura VXRS

Frontend jednostki wektorowej potrzebuje znać kolejność instrukcji w porządku programu. Wynika to z tego, że instrukcje wektorowe używają stanu globalnego w postaci CSR-ów `vtype` i `vl`, a dostępy i modyfikacje na takim stanie muszą być odpowiednio uporządkowane. Implementując rdzeń zdecydowano się na użycie `FifoRS` jako implementacji `VXRS`-a, co spowodowało przekazywanie instrukcji z gotowymi rejestrami skalarnymi do `VectorFrontend` w porządku programu. Jednakże tworząc taką implementację nie wzięto pod uwagę faktu, że może to ograniczać wykonanie *out-of-order*. Przykład przedstawiono w kodzie 6.1.

W sytuacji gdy rejestr `v3` jest gotowy, a `x2` jeszcze nie, to można by oczekiwać, że druga instrukcja `vadd` zacznie się wykonywać, a w tym czasie pierwsza będzie

```
1 lw x2, 0(x1)
2 vadd.vx v2, v1, x2
3 vadd.vv v4, v3, v3
```

Kod źródłowy 6.1: Kod w asemblerze RVV, który – jeśli `v3` jest gotowe – wykona się nieoptymalnie na aktualnej implementacji rozszerzenia „V" w Coreblocksie, ze względu na implementację `VXRS`.

czekała na operand skalarny. Jednakże z powodu użycia `FifoRS` do implementacji `VXRS`, instrukcje wektorowe są przekazywane do `VectorFrontendu` w porządku programu, więc drugie `vadd` zostanie przekazane, dopiero jak wszystkie operandy skalarne pierwszego `vadd` będą gotowe.

Aby rozwiązać powyższy problem należałoby zrównoleglić `VectorFrontend` i `VXRS`. Instrukcje zlecane do wykonania przez rdzeń skalarny byłyby umieszczane w `VXRS`, a następnie wraz z identyfikatorem pozycji przekazywane od razu do `VectorFrontend`, który przetwarzałby instrukcje tak długo jak to możliwe bez posiadania operandów skalarnych. W przypadku instrukcji `vset{i}vl{i}` byłoby to w zasadzie tożsame z aktualną implementacją, ponieważ instrukcja od razu byłaby stallowana i oczekiwałaby na operandy skalarne. Natomiast w przypadku instrukcji wektorowych, można by było przetworzyć je aż do umieszczenia ich w `VVRS`, a instrukcje z `VVRS` mogą już wykonywać się *out-of-order*. Przy czym należałoby dodać możliwość przekazywania informacji o gotowości operandów skalarnych z `VXRS` do `VVRS`.

6.2. Możliwe rozszerzenia i optymalizacje

6.2.1. Redukcja opóźnienia

W przyszłych rewizjach rdzenia wektorowego można poprawić opóźnienia przetwarzania instrukcji wektorowych przez Coreblocksa. Podczas implementacji zakładano, że opóźnienie to będzie ukrywane przez instrukcje skalarne, jednak jak widać w wynikach wykonania programów przedstawionych w tabeli 5.1 wydajność jest poniżej oczekiwań i opóźnienie instrukcji odgrywa w tym znaczącą rolę. Od przekazania instrukcji wektorowej do wykonania, do wyciągnięcia jej z `VVRS` mija 12 cykli zegarowych (zakładając, że wszystkie argumenty są od razu gotowe).

Można przeprojektować `VectorFrontend`, tak by z aktualnych 5 cykli, skrócić czas wykonania do 2 cykli. W tym celu należałoby zrównoleglić wykonanie `VectorStatus`, `VectorTranslator` i `VectorAlloc`, jako że bloki te nie zależą od siebie w aktualnej implementacji (choć w przyszłości, alokator rejestrów wektorowych może brać przykładowo pod uwagę `EEW`).

Opóźnienie można zredukować także poprzez scalenie `VectorEnder` z `VectorAnnouncer`. Pierwszy blok odpowiada bowiem za zebranie informacji o zakończeniu pracy z `VectorExecutor`ów, natomiast drugi zbiera wyniki z całego `VectorCore`, a więc także z `VectorEnder`. Usunięcie warstwy abstrakcji w postaci `VectorEnder`a pozwoliłoby więc potencjalnie na zredukowanie jednego cyklu opóźnienia.

Trzecim miejscem, w którym można szukać możliwości zredukowania opóźnienia jest VRF. Jego aktualna implementacja wprowadza 5 cykli opóźnienia między zleceniem odczytu, a uzyskaniem wyniku. Wynika to przede wszystkim z buforów, które są potrzebne w sieci routingu, aby przeciąć ścieżki krytyczne (patrz sekcja 3.2.). Zredukowanie liczby banków i tym samym sieci routującej – opisane w poprzedniej sekcji – mogłoby się więc przełożyć na mniejszą liczbę potrzebnych buforów i mniejsze opóźnienie.

6.2.2. Zarządzanie rejestrami wektorowymi z schedulera skalarnego

Implementując rdzeń wektorowy zdecydowano się sterować rejestrami wektorowymi z jednostki funkcyjnej `VectorCore`, co wynikało z tego, że alokacja rejestrów wektorowych zależy od `LMUL` będącego częścią CSR-a `vtype`. Nie jest to jednak najbardziej optymalne rozwiązanie.

W specyfikacji RVV wymienione są trzy instrukcje do kontrolowania CSR-a `vtype`: `vsetvl`, `vsetvli`, `vsetivli`. Można zauważyć, że zarówno dla `vsetivli` jak i dla `vsetvli`, `LMUL` będące wynikiem wykonania instrukcji jest znane już na etapie dekodowania, bowiem jest częścią kodowania binarnego instrukcji. Jedynie w `vsetvl` wartość `LMUL` zależy od rejestru skalarnego. Niemniej jednak specyfikacja RVV stwierdza w komentarzu, że `vsetvl` jest przeznaczone do przywracania kontekstu. Można więc wnioskować, że będzie znacząco rzadziej używane niż instrukcje kodujące `LMUL` jako stałą.

Zgodnie z zasadą optymalizacji najczęstszych przypadków, możemy więc przeprowadzić optymalizację wykonania instrukcji wektorowych, kosztem mniejszej wydajności procesora podczas przetwarzania instrukcji `vsetvl`. Zarządzanie rejestrami wektorowymi można bowiem zintegrować ze schedulerem skalarnym, który będzie przechowywał lokalną kopię `LMUL`. W przypadku przetworzenia `vsetivli` bądź `vsetvli` kopia ta będzie aktualizowana przed skierowaniem instrukcji do właściwego wykonania. Dzięki temu następujące instrukcje wektorowe będą miały już poprawną wartość `LMUL`, mimo że wykonanie poprzedzającej instrukcji `vsetivli`/`vsetvli` jeszcze się nie zaczęło. Pozwoli to zrównoleglić zarządzanie rejestrami skalarnymi i wektorowymi i tym samym zredukować opóźnienie wykonania instrukcji wektorowych.

Kosztem powyższego rozwiązania będzie blokada schedulera skalarnego przez `vsetvl` do czasu zakończenia wykonania tej instrukcji. Ustawia ona bowiem `LMUL` na podstawie rejestru, więc instrukcje wektorowe w schedulerze będą musiały czekać

aż się ona zakończy, zanim będą mogły zaalokować rejestry i przejść do następnego etapu.

Oczywiście i ten przypadek, choć rzadki, można optymalizować. Jeśli założymy zgodnie z sugestią specyfikacji RVV, że `vsetvl` służy do przywracania kontekstu, to przed użyciem tej instrukcji kiedyś musieliśmy odczytać `vtype`, podczas zapisywania kontekstu. Można więc potencjalnie przechowywać ostatni `vtype` i próbować spekulować, że `vsetvl` ustawi właśnie ten `vtype`. Jednakże potencjalny zysk jest minimalny, więc tak jak przywracanie stanu predyktora skoków po zmianie kontekstu, tak i spekulacja na przywracaniu `vtype` to raczej rozważania teoretyczne.

Warto zauważyć, że poza redukcją opóźnienia, przeniesienie zarządzania rejestrami wektorowymi do schedulera skalarnego ma jeszcze jedną zaletę. Mapowanie docelowych rejestrów logicznych na docelowe rejestry fizyczne można ustawić podczas alokowania wpisu w ROB-ie. Dzięki temu nie ma potrzeby wykorzystywania CAM w `VectorRetirement`, bowiem wszystkie potrzebne informacje przechowujemy w ROB.

6.2.3. Zwiększenie wykorzystania taśm

Funkcjonalnością, którą na pewno będzie warto dodać w przyszłych rewizjach rdzenia „V", jest balansowanie obciążenia między różnymi taśmami. Aktualnie fragmenty pliku rejestrów należące do taśmy mapują się na kolejne ciągłe fragmenty rejestru, tzn. taśma 0 operuje na elementach o adresach $[0, X)$, taśma 1 na adresach $[X, 2 * X]$ itp. Powoduje to, że w przypadku gdy rejestr nie jest w pełni zapełniony, elementy są niezbalansowane między taśmami i może w skrajnych przypadkach prowadzić do sytuacji, w której tylko jedna taśma działa, a pozostałe nic nie robią.

W docelowym rozwiązaniu kolejne elementy, powinny mapować się na kolejne taśmy, tzn. element 0 na taśmę 0, element 1 na taśmę 1, element n+1 na taśmę 0 (gdzie n to liczba taśm) itd. Dzięki temu różnica w liczbie elementów do przetworzenia przez różne taśmy będzie co najwyżej jeden.

Wadą powyższej funkcjonalności będzie konieczność wprowadzenia jednostki translacji adresów. Wynika to z faktu, że instrukcje takie jak operacje na pamięci oraz przesunięcia elementów w rejestrach operują na adresach logicznych, a te będą różne od fizycznych.

Rozdział 7.

Podsumowanie

W ramach niniejszej pracy zaimplementowano szkielet podzbioru rozszerzenia wektorowego RISC-V w mikroarchitekturze *out-of-order*. Stworzony rdzeń umożliwia:

- równoległe wykonywanie instrukcji wektorowych i skalarnych;
- wykonywanie instrukcji wektorowych poza porządkiem;
- przenazywanie rejestrów wektorowych.

Jednocześnie zademonstrowano możliwości Transactrona w tworzeniu kodu opisującego skomplikowane układy. Dzięki wprowadzanej abstrakcji umożliwia on bardziej modularne i przenośne pisanie kodu niż języki niskiego poziomu (np. Verilog). Jednocześnie łatwiej w nim jest optymalizować kod niż w innych aktualnie popularnych bibliotekach wysokopoziomowego opisu sprzętu, co pokazano na przykładzie Hwacha pisanej w Scali.

Jednakże pomimo powyższych sukcesów pozostało sporo pracy do wykonania zanim rdzeń wektorowy w Coreblocksie będzie w pełni funkcjonalny i będzie można porównywać go do procesorów *State Of The Art*, tworzonych w Barcelona Supercomputing Center czy w ETH Zurich. W szczególności należy:

- zoptymalizować liczbę banków w pliku rejestrów;
- poszerzyć zbiór wspieranych instrukcji;
- usunąć ograniczenia rdzenia skalarnego.

Dodatek A

Uwagi

A.1. Wykorzystane narzędzia i biblioteki

amaranth f96604f667516ba9fbf125db8dff8b3742f61f1d
`https://github.com/amaranth-lang/amaranth`

python 3.11.3 `https://github.com/python/cpython`

yosys 0.25 `https://github.com/YosysHQ/yosys`

nextpnr-ecp5 b5d30c73877be032c1d87cd820ebdfe4db556fdb
`https://github.com/YosysHQ/nextpnr`

docker 24.0.5 `https://github.com/docker`

OpenROAD Flow Scripts bf9f1e81bfabb85c0bfe794d18a4ea82dda83872
`https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts`

verilator 5.014 `https://github.com/verilator/verilator`

gcc 12.2.0 `https://gcc.gnu.org/`

LLVM 15.0.7 `https://github.com/llvm/llvm-project`

Intel Quartus 22.1
`https://www.intel.com/content/www/us/en/software-kit/773997/intel-quartus-prime-lite-edition-design-software-version-22-1-1-for-linux.html`

A.2. Wkład twórczy

- Praca ta jest kontynuacją projektu „Kuźnia Rdzeni” w ramach którego rozwijany jest procesor Coreblocks.

- Szczegółową listę zmian wprowadzonych w Coreblocksie w ramach tej pracy można znaleźć w `https://github.com/kuznia-rdzeni/coreblocks/pull/395`.
- W trakcie prac wykorzystano bibliotekę Transactron oraz moduły Coreblocksa stworzone przez innych współautorów „Kuźni Rdzeni".

Dodatek B

Kod źródłowy benchmarków

Poniżej przedstawiono kody źródłowe stanowiące rdzenie benchmarków „vadd", „vadd-lot-of-scalars", „vadd-mem" i „vmem". Rdzenie te zostały wstawione do wcześniej istniejącej infrastruktury testowej procesorów Ara i Coreblocks. Wszystkie rdzenie zapisane są w składni *GCC Extended Inline Assembly*.

```
1       addi x0, x0, 0
2       vsetvli x0, %[LEN], e32,m1,ta,ma
3       vle32.v v1, (%[tab_in])
4       vadd.vi v2, v1, 0
5   start_vadd_%=:
6       vadd.vv v2, v2, v1
7       addi %[counter], %[counter], -1
8       bne x0, %[counter], start_vadd_%=
9       vse32.v v2, (%[tab_out])
```

Kod źródłowy B.1: Rdzeń benchmarku „vadd", którego celem jest sprawdzenie wydajności operacji wektorowych artymetyczno-logicznych.

```
1       addi x0, x0, 0
2       vsetvli x0, %[LEN], e32,m1,ta,ma
3       vle32.v v1, (%[tab_in])
4       vadd.vi v2, v1, 0
5   start_vadd_%=:
6       vadd.vv v2, v2, v1
7       addi %[counter], %[counter], -1
8       li %[buf1], 2
9       li %[buf2], 4
10      add %[buf1], %[buf2], %[buf1]
11      add %[buf1], %[buf2], %[buf1]
12      addi %[buf1], %[buf1], -1
13      bne x0, %[counter], start_vadd_%=
```

```
14      vse32.v v2, (%[tab_out])
```

Kod źródłowy B.2: Rdzeń benchmarku „vadd-lot-of-scalars", testującego wpływ operacji skalarnych między instrukcjami wektorowymi na wydajność rdzenia.

```
1       addi x0, x0, 0
2       vsetvli x0, %[LEN], e32,m1,ta,ma
3       vle32.v v1, (%[tab_in])
4       vadd.vi v3, v1, 10
5       vadd.vi v2, v1, 0
6   start_vadd_%=:
7       vle32.v v1, (%[tab_in])
8       vadd.vv v2, v2, v3
9       vadd.vv v2, v2, v3
10      vadd.vv v2, v2, v1
11      addi %[counter], %[counter], -1
12      bne x0, %[counter], start_vadd_%=
13      vse32.v v2, (%[tab_out])
```

Kod źródłowy B.3: Rdzeń benchmarku „vadd-mem", sprawdzającego wydajność przeplatających się wektorowych instrukcji operujących na pamięci i wektorowych instrukcji arytmetyczno-logicznych.

```
1       vsetvli x0, %[LEN], e32,m1,ta,ma
2   start_vadd_%=:
3       vle32.v v1, (%[tab_in])
4       vle32.v v2, (%[support_tab])
5       vadd.vv v2, v2, v1
6       vse32.v v2, (%[support_tab])
7       addi %[counter], %[counter], -1
8       bne x0, %[counter], start_vadd_%=
9       vle32.v v2, (%[support_tab])
10      vse32.v v2, (%[tab_out])
```

Kod źródłowy B.4: Rdzeń benchmarku „vmem", testującego wydajność wektorowych operacji na pamięci.

Bibliografia

Cavalcante, Matheus et al. (2020). "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.2, pp. 530–543. DOI: 10.1109/TVLSI.2019.2950087.

Chou, Christopher H. et al. (2011). "VEGAS: Soft Vector Processor with Scratchpad Memory". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. Monterey, CA, USA: Association for Computing Machinery, pp. 15–24. ISBN: 9781450305549. DOI: 10.1145/1950413.1950420. URL: https://doi.org/10.1145/1950413.1950420.

Lee, Yunsup et al. (2014). "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators". In: *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, pp. 199–202. DOI: 10.1109/ESSCIRC.2014.6942056.

Lee, Yunsup et al. (Dec. 2015). *Hwacha Preliminary Evaluation Results*. Version 3.8.1. URL: https://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2015-264.pdf.

Minervini, Francesco et al. (Mar. 2023). "Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications". In: *ACM Trans. Archit. Code Optim.* 20.2. ISSN: 1544-3566. DOI: 10.1145/3575861. URL: https://doi.org/10.1145/3575861.

Patsidis, K. et al. (2020). "RISC-V^2: A Scalable RISC-V Vector Processor". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5.

Perotti, Matteo et al. (July 2022). "A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design". In: *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. DOI: 10.1109/asap54787.2022.00017. URL: https://doi.org/10.1109%2Fasap54787.2022.00017.

Platzer, Michaela D. and Peter P. Puschner (2021). "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation". In: *Euromicro Conference on Real-Time Systems*. URL: https://api.semanticscholar.org/CorpusID:235736845.

RISC-V International (Sept. 2021). *RISC-V "V" Vector Extension*. URL: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

RISC-V International (Sept. 2023). *History of RISC-V*. URL: https://riscv.org/about/history/.

Russell, Richard M. (Jan. 1978). "The CRAY-1 Computer System". In: *Commun. ACM* 21.1, pp. 63–72. ISSN: 0001-0782. DOI: 10.1145/359327.359336. URL: https://doi.org/10.1145/359327.359336.

Shen, John Paul and Mikko H. Lipasti (2002). "Modern Processor Design: Fundamentals of Superscalar Processors". In.

Xu, Yinan et al. (2022). "Towards developing high performance RISC-V processors using agile methodology". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, pp. 1178–1199.

Yiannacouras, Peter, J. Gregory Steffan, and Jonathan Rose (2008). "VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors". In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '08. Atlanta, GA, USA: Association for Computing Machinery, pp. 61–70. ISBN: 9781605584690. DOI: 10.1145/1450095.1450107. URL: https://doi.org/10.1145/1450095.1450107.