AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

Projekt dyplomowy

Programowalny moduł dla układu rekonfigurowalnego wspierający obliczenia zmiennopozycyjne

Programmable IP core supporting floating point calculations

Autor: *Daniel Dominik Cichon*
Kierunek studiów: *Automatyka i Robotyka*
Opiekun pracy: *dr inż. Krzysztof Kołek*

Kraków, 2021r.

Spis treści

1. Wstęp

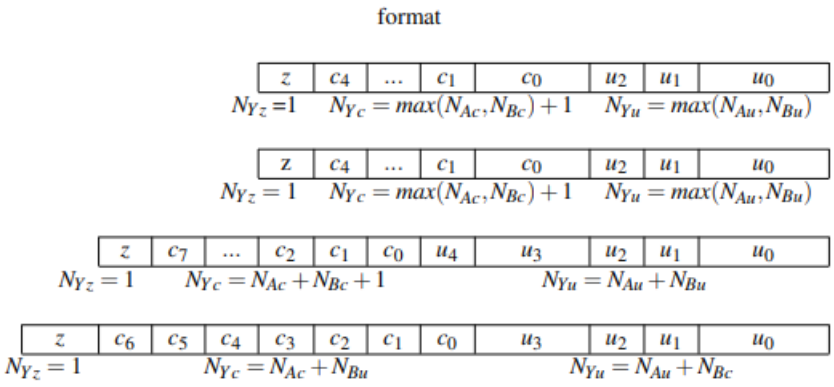
W dzisiejszych czasach mnóstwo procesorów posiada osobne jednostki odpowiedzialne za obliczenia zmiennopozycyjne. Celem projektu była implementacja podobnej jednostki w układzie rekonfigurowalnym. W dalszej części moduł ten będzie określany jako FPU, bądź soft procesor. Ideą modułu było wspieranie obliczeń zmiennopozycyjnych, czyli niejako odciążanie głównej, osobnej jednostki (np. CPU) od tych operacji. Założono możliwość wykonywania ośmiu operacji arytmetycznych na liczbach zmiennoprzecinkowych pojedynczej oraz podwójnej prcyzji: dodawania, odejmowania, mnożenia, dzielenia, odwrotności, pierwiastka kwadratowego, logarytmu naturalnego oraz eksponenty. Wszystkie operacje wykonywane są na danych, więc konieczna była implementacja pamięci do pomieszczenia zarówno operandów jak i wyników operacji. Do określenia operacji arytmetycznych oraz kolejności ich wykonywania potrzebna była dodatkowa pamięć zawierająca instrukcje. Instrukcje to w praktyce jedynie ciągi binarne interpretowane w konkretny sposób. Postanowiono zaimplementować osobny moduł do podziału instrukcji na poszczególne sygnały kierowane do reszty komponentów soft procesora. Moduł ten nazwano jednostką sterującą. Każdy procesor zawiera wewnętrzne rejestry, do których ładowane są operandy z pamięci dancyh oraz do których trafiają wyniki, zanim zapisane zostaną w pamięci. Postąpiono podobnie implementując plik rejestrów zawierający wszystkie rejestry dla liczb pojedynczej oraz podwójnej precyzji. Ich strukturę opisano dokładnie w rozdziale 3.2. Aby usprawnić proces pisania programów do wykonywania przez soft procesor uznano za konieczne zbudowanie programu - kompilatora, który instrukcje w postaci assemblera przekształcał na ciągi binarne, które następnie trafiały do pamięci instrukcji. Nieuniknioną częścią pisania oprogramowania są testy, dlatego również w tym projekcie postanowiono testować symulacyjnie każdy komponent z osobna testami jednostkowymi oraz całość testem integracyjnym. Ostatecznym krokiem było wgranie soft procesora na układ rekonfigurowalny FPGA oraz przetestowanie go w sprzęcie.

W drugim rozdziale opisane zostały liczby zmiennoprzecinkowe oraz charakter operacji na nich przeprowadzanych. Wytłumaczono dlaczego operacje te nie są tak intuicyjne jak mogłoby się wydawać, co zostało poparte przykładowymi programami. W kolejnym rozdziale opisano implementację wszystkich komponentów składających się na soft procesor. Rozdział czwarty poświęcony został implementacji kompilatora napisanego w środowisku *MATLAB.* Testy symulacyjne oraz sprzętowe i ich wyniki zostały opisane w rozdziale piątym, po którym podsumowano przebieg całego projektu.

2. Metody realizacji obliczeń matematycznych w procesorach

2.1. Liczby stałoprzecinkowe

Liczby stałoprzecinkowe (*ang. Fixed Point*) służą do obliczeń matematycznych w których potrzebna jest część ułamkowa. Do ich zapisu w pamięci komputera możemy przeznaczyć różną liczbę bitów – nie jest określone z góry, że musi to być 32 lub 64 bity. W zapisie liczby stałoprzecinkowej pierwszy bit przeznaczony jest na znak, następnie kilka bitów na część całkowitą oraz reszta na część ułamkową. Ilość bitów przeznaczona na części ułamkową i całkowitą jest kwestią umowną. Zaletą liczb stałej precyzji jest mniejszy koszt bloków DSP odpowiedzialnych za obliczenia na tych liczbach. Jeśli natomiast chodzi o łatwość implementacji na mnożarkach DSP, to liczby zmiennoprzecinkowe w dzisiejszych czasach górują [1]. Patrząc na własności liczb stałoprzecinkowych pokazane w rysunku 2.1.1 widzimy, że po każdej operacji operacji liczba bitów do zapisu wyniku powinna się zwiększyć.



Rys 2.1.1 *Szerokość rejestrów zawierających wyniki operacji arytmetycznych (źródło: [2])*

Oznaczenia do rysunku 2.1.1:

z - bit znaku,

c_x - bit części całkowitej na pozycji x,

u_x - bit części ułamkowej na pozycji x,

N_{Ac}, N_{Bc} - liczba bitów części całkowitej operandu kolejno *a* oraz *b*,

N_{Au}, N_{Bu} - liczba bitów części ułamkowej operandu kolejno *a* oraz *b*,

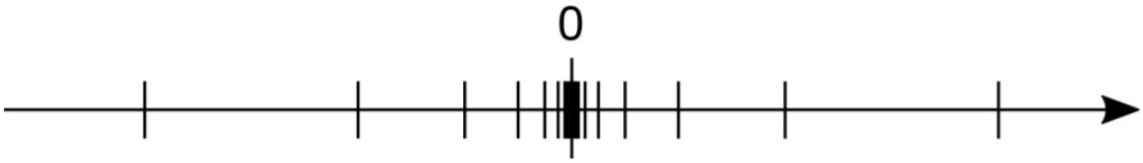
N_{Yz} - liczba bitów znaku liczby wynikowej *y*,

N_{Yc}, N_{Yu} - liczba bitów kolejno części całkowitej i ułamkowej liczby wynikowej *y*.

Na rys 2.1.1 cztery wiersze przedstawiają kolejno szerokości jakie powinny przyjąć rejestry w wyniku dodawania, odejmowania, mnożenia oraz dzielenia liczb stałoprzecinkowych. Powiększanie rejestru wynikowego jest koniecznym zabezpieczeniem przed sytuacjami, w których wynik jest na tyle duży, że nie zmieści się w ilości bitów przeznaczonej dla operandów. Jak wiadomo w procesorach oraz układach ASIC wszystkie rejestry są ustalonej wartości, a co za tym idzie nie mamy elastyczności wyboru długości liczb stałoprzecinkowych. Gdybyśmy przeprowadzali obliczenia na liczbach, których dokładność byłaby satysfakcjonująca dla 9 bitów, musielibyśmy te liczby zapisać w komórkach pamięci, których szerokość jest większą, kolejną potęgą dwójki, czyli w tym przypadku 16. Od razu rzuca się w oczy strata całych 7 bitów, które stanowią aż 43% naszej komórki. Oczywiście można powiedzieć, że to tylko 7 bitów, lecz ilość straconej pamięci rośnie wraz z ilością danych wejściowych wykonywanego algorytmu. W tym momencie pojawia się ogromna zaleta układów FPGA, w których to projektant decyduje o szerokości każdego rejestru. Elastyczność układów FPGA pozwala na definiowanie własnych szerokości rejestrów potrzebnych do wykonania każdego algorytmu. Dzięki temu implementując pożądany algorytm na liczbach stałoprzecinkowych możemy wykorzystać cały zasób pamięci do maksimum nie tracąc ani jednego, czasem kluczowego bitu.

2.2. Liczby zmiennoprzecinkowe

Liczby zmiennoprzecinkowe (*ang. Floating Point*) to w pamięci programu zmienne, w których możemy zapisać liczby z niewyobrażalnie wielką precyzją. Liczby te dzielą się na dwie kategorie: liczby pojedynczej oraz liczby podwójnej precyzji. Liczby pojedynczej precyzji określa się jako *float* a podwójnej precyzji jako *double*. *Float* zapisane są na 32 bitach, a *double* w swojej notacji wykorzystują 64 bity. Wykorzystując te liczby zyskujemy precyzję znacznie większą niż korzystacjąc z liczb stałoprzecinkowych. Różnice między kolejnymi liczbami stałoprzecinkowymi zawsze są identyczne, jeśli nie jest zmieniany format tejże liczby (np. 1 bit znaku, 20 bitów na część całkowitą oraz 11 bitów na część ułamkową), natomiast różnice między kolejnymi liczbami zmiennoprzecinkowymi są różne w zależności od wielkości liczby. Szacuje się, że różnica między dwoma kolejnymi liczbami możliwymi do zapisania w formacie zmiennoprzecinkowym jest 10 milionów razy mniejsza niż te liczby. Zatem różnice w kolejnych liczbach są małe dla małych liczb i rosną one wraz z wielkością tych liczb [1].

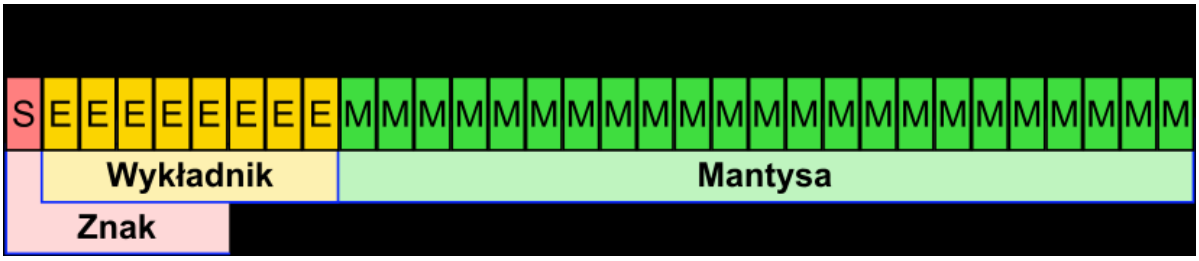


Rys 2.2.1 *Rozkład liczb zmiennoprzecinkowych w pobliżu zera (źródło: [3])*

Skoro szerokość liczb zmiennoprzecinkowych nie zmienia się, a zostaje zachowana w wyniku każdej operacji, to w układach scalonych o ustalonych parametrach wykorzystywane są zawsze 32 lub 64 bity na zapis danej liczby. Przewaga układów FPGA polegająca na oszczędności pamięci znika. Czy zatem warto implementować obliczenia zmiennoprzecinkowe w tych układach? Przewaga liczb zmiennoprzecinkowych nad stałoprzecinkowymi leży w precyzji. Jest on na tyle duża, że firmy zajmujące się produkcją układów FPGA zaczęły natywnie wspierać obliczenia zmiennoprzecinkowe implementując dedykowane bloki DSP w sprzęcie [4]. Przykładem może być firma Altera, o której mowa w rozdziale 1.5. Osadzając w układach FPGA połączenia dedykowane obliczeniom na liczbach zmiennoprzecinkowych niwelowana zostaje przewaga liczb stałoprzecinkowych co do szybkości obliczeń. Większa szybkość obliczeń stałoprzecinkowych wynikała z tego, że obliczenia zmiennoprzecinkowe implementowane były za pomocą algorytmów, w których wydzielało się osobno bit znaku, wykładnik oraz mantysę a następnie przetwarzało je logiką zaimplementowaną w kodzie HDL. Dzięki blokom DSP dedykowanym do obliczeń zmiennoprzecinkowych nie ma już potrzeby implementacji tych algorytmów, a szybkość wykonywania obliczeń dorównała obliczeniom stałoprzecinkowym [1].

2.3. Standard IEEE 754

Standard IEEE określa notację oraz obliczenia na liczbach zmiennoprzecinkowych. Na rysunku 3.3.1 pokazany został zapis liczby pojedynczej precyzji. "S" to pierwszy bit odpowiedzialny za znak, bity oznaczone literką "E" to wykładnik (znany również jako cecha), a "M" to mantysa. Wartość liczby zmiennoprzecinkowej wyrażana jest wzorem (3.3.1), gdzie "S", "M" oraz "E" pokrywa się ze znakami z rysunku, natomiast "B" to podstawa systemu liczbowego (2 dla systemów komputerowych). Liczba podwójnej precyzji wykorzystuje 11 bitów na znak oraz 54 bity na mantysę [5].



Rys.2.3.1 *Zapis liczb zmiennoprzecinkowych (źródło: [5])*

Dla bitu "S" wartość 0 oznacza liczbę dodatnią, a dla wartości "1" – liczbę ujemną. Kodowanie cechy jest kodowaniem z nadmiarem (bias = 127), co daje zakres od -127 do 128 włącznie z pominięciem minimalnej oraz maksymalnej liczby możliwej do zapisania na tych ośmiu bitach. Gdy w wykładniku występują same zera mamy do czynienia z liczbą zdenormalizowaną, co pozwala na zapis jeszcze mniejszych liczb. Natomiast gdy cecha składa się z samych jedynek i jednocześnie w mantysie znajdują się same zera, wówczas mamy do czynienia z nieskończonością, która w zależności od bitu znaku jest dodatnia bądź ujemna. W przypadku gdy w wykładniku znajdują się same jedynki, a w mantysie jest co najmniej jedna oznacza to, że działamy na NaN – Not a Number. Pomijając widący bit mantysy otrzymujemy ich 22 na jej zapis, co przekłada się na 7-8 miejsc dziesiętnych po przecinku. Zatem najmniejsza liczba możliwa do zapisania w tej notacji to w przybliżeniu $1.4013 \cdot 10^{-45}$, a największa to $3.4028 \cdot 10^{38}$.

2.4. Stopień skomplikowania operacji na liczbach zmiennoprzecinkowych

Pierwszą wadą liczb zmiennoprzecinkowych jest fakt, iż nie można na nich zapisać zera. Najmniejsza liczba możliwa do zapisania w zmiennej pojedynczej precyzji to 1.4×10^{-45} a w podwójnej precyzji 5×10^{-324} [3]. Kolejna sprawa to liczby całkowite, które da się zapisać na liczbach zmiennoprzecinkowych. Działania wykonują się dokładnie, bez żadnych błędów zaokrągleń, natomiast tylko do pewnej maksymalnej wartości, od której błąd między kolejnymi liczbami jest większy niż 1. Wartością tą dla liczby pojedynczej precyzji jest 16 777 216. Jak widać na rys. 3.4.1 inkrementacja do liczby 16 777 217, oraz jej zapis są niemożliwe. Kolejne liczby "przeskakują" co 2, następnie co 4, itd. [3].

```
int main()
{
    float x = 16777215;
    std::cout << "x = " << std::setprecision(10) << x << std::endl;
    x += 1;
    std::cout << "x = " << std::setprecision(10) << x << std::endl;
    x += 1;
    std::cout << "x = " << std::setprecision(10) << x << std::endl;
    float f = 16777217;
    std::cout << "f = " << std::setprecision(10) << f << std::endl;
    return 0;
}
```

```
Microsoft Visual Studio Debug Console
x = 16777215
x = 16777216
x = 16777216
f = 16777216
```

Rys 2.4.1 *Kod przedstawiający graczną wartość poprawnej liczby całkowitej zapisanej na liczbie zmiennoprzecinkowej pojedynczej precyzji (źródło: opracowanie własne)*

Istotną wadą liczb zmiennoprzecinkowych jest fakt, że z pozoru te same liczby mogą się różnić na ostatnich miejscach bardzo nieznacząco, dlatego nie należy tych liczb bezpośrednio porównywać operatorem porównania. Aby poprawnie porównać ze sobą dwie liczby zmiennoprzecinkowe należy odjąć jedną liczbę od drugiej oraz sprawdzić czy różnica jest mniejsza niż pewna mała wartość, która może być wynikać z błędu zaokrąglenia [3].

```
float t = 1.0f / 10.0f;
float g = 1.0f - 0.9f;
std::cout << std::setprecision(10) << "t = " << t << ", g = " << g << std::endl;
std::cout << ((g == t) ? "Sa rowne" : "Nie sa rowne") << std::endl;
std::cout << ((g - t < 0.00001) ? "Sa rowne" : "Nie sa rowne") << std::endl;
return 0;
```

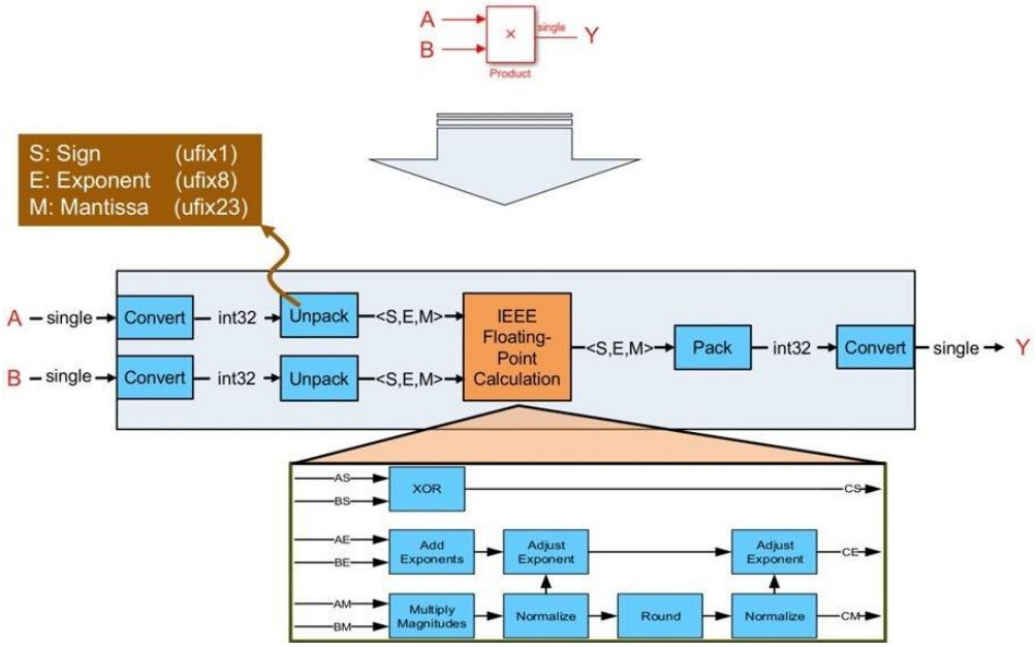
```
Microsoft Visual Studio Debug Console
t = 0.1000000015, g = 0.1000000238
Nie sa rowne
Sa rowne
```

Rys 2.4.2 *Kod przedstawiający dlaczego nie należy porównywać bezpośrednio liczb zmiennoprzecinkowych (źródło : opracowanie własne na podst. [3])*

Arytmetyka liczb zmiennoprzecinkowych nie jest *łączna*, ani *rozdzielna* a co za tym idzie: $(x+y)+z \neq x+(y+z)$ oraz $(xy)z \neq x(yz)$ i $x(y+z) \neq (xy)+(xz)$ [6].

2.5. Różne podejścia implementacji obliczeń zmiennoprzecinkowych w układach FPGA

Znana firma Mathworks pozwala na generację kodu HDL w programi *MATLAB* Simulink wykonującego obliczenia zmiennoprzecinkowe [7]. Rysunek 2.5.1 przedstawia kolejne kroki implementacji iloczynu liczb zmiennoprzecinkowych pojedynczej precyzji, które stosuje firma Mathworks. Jak widać na wejściu liczby pojedynczej precyzji są konwertowane na typ int32, z którego wyciągany jest do osobnych zmiennych kolejno znak, eksponenta oraz mantysa. Na tych zmiennych wygenerowana logika w jeżyku VHDL lub Verilog przeprowadza operacje pokazane w bloku "IEEE Floating Point Calculation" zgodne ze standardem IEEE 754 omawianym w rozdziale 2.3. Gdy te operacje zostaną zakończone ostatnim krokiem jest złożenie wynikowych bitów znaku, eksponenty oraz mantysy w jedną liczbę typu int32 oraz przekonwertowanie jej na liczbę zmiennoprzecinkową pojedynczej precyzji [7].



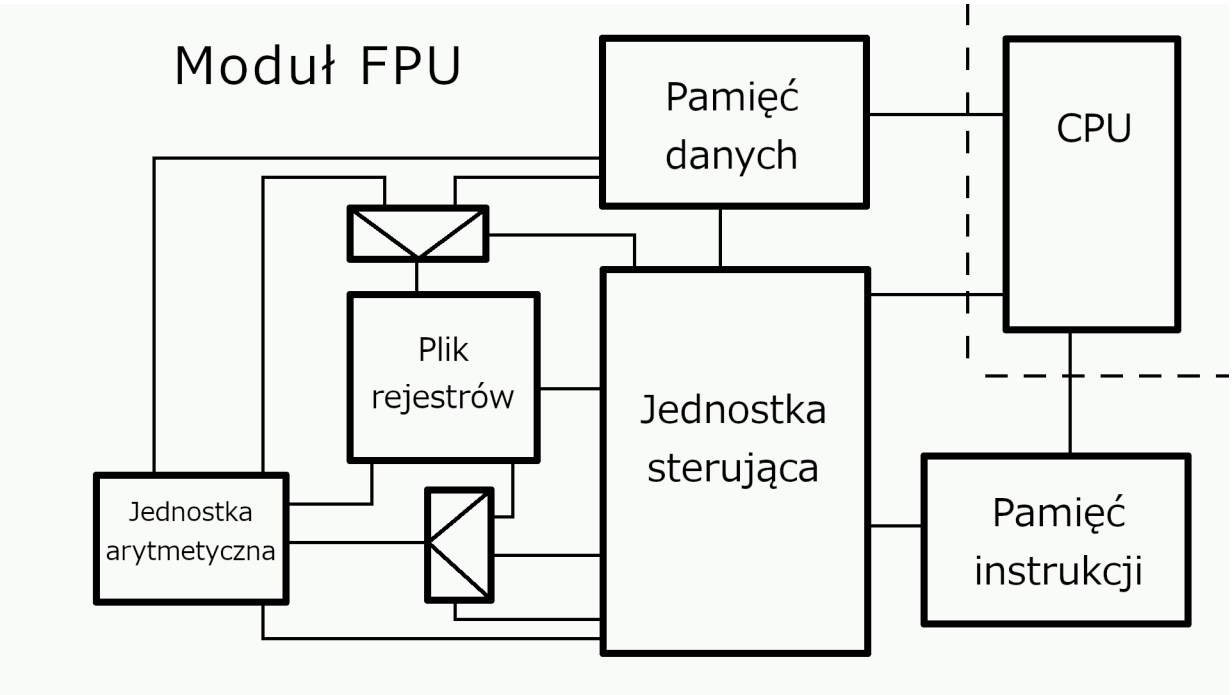
Rys 2.5.1 *Implementacja mnożenia liczb zmiennoprzecinkowych firmy Mathworks (źródło: [7])*

W 2014 roku firma Altera wydała układ FPGA natywnie wspierający obliczenia zmiennoprzecinkowe pojedynczej precyzji. Było to przełomowe przedsięwzięcie, gdyż dotychczas układy FPGA wspierały natywnie jedynie obliczenia na liczbach

stałoprzecinkowych. Wydarzenie to dało projektantom możliwość implementacji algorytmów na liczbach zmiennoprzecinkowych z taką samą wydajnością jak na liczbach stałoprzecinkowych [8]. W dziesiątej generacji Altera FPGA zostały dodane bloki DSP zawierające dodawarki oraz mnożarki liczb zmiennoprzecinkowych. Wydajność układów Arria 10 z rodziny 20nm szacuje się na 140 GigaFLOPS do 1.5 TeraFLOPS, przy czym układy Stratix 10 z rodziny 14nm używając tej samej architektury osiąga do 10 TeraFLOPS. Część bloków DSP jest pogrupowana w kolumny których zadaniem jest wspieranie typowych obliczeń takich jak transformata Fouriera (FFT), filtry o skończonej odpowiedzi impulsowej (FIR), czy też zwykłe obliczenia stosowane w algebrze liniowej [8]. Altera zapewnia około 70 funkcji w bibliotece math.h zgodnych ze specyfikacją OpenCL 1.2, które są zoptymalizowane pod względem nowej, natywnej dla obliczeń zmiennoprzecinkowych architektury FPGA [8].

3. Programowa implementacja soft procesora

3.1. Schemat blokowy

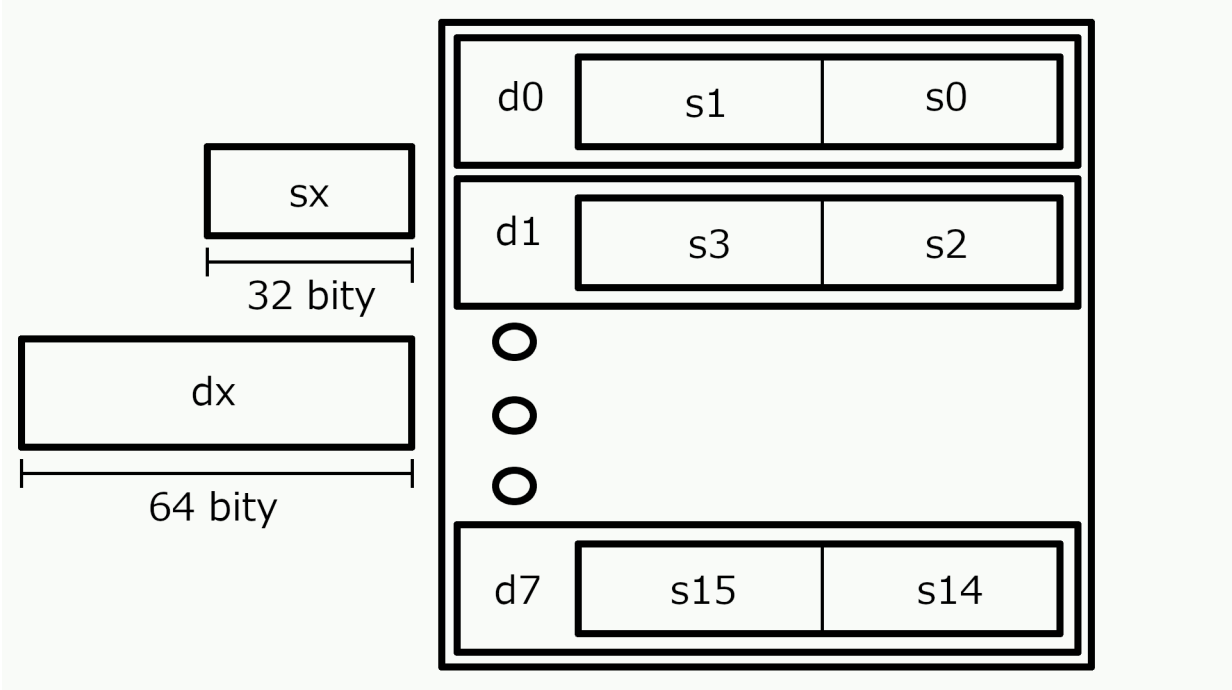


Rys 3.1.1 *Schemat blokowy soft procesora (źródło: opracowanie własne)*

Na rysunku 3.1.1 został pokazany schemat blokowy soft procesora (modułu FPU). Jak widać na schemacie jednostka zewnętrzna CPU komunikuje się z pamięcią danych, pamięcią instrukcji oraz jednostką sterującą. CPU informuje jednostkę sterującą o tym, że będzie zapisywać dane do pamięci danych oraz instrukcje do pamięci instrukcji. Gdy zakończy zapisywanie informuje o tym jednostkę sterującą, która następnie zaczyna ładować instrukcje. 64 bitowe instrukcje następnie dzieli na kawałki i wystawia odpowiednie sygnały w pliku rejestrów, pamięci danych, jednostce arytmetycznej oraz w multiplekserach. Po zakończonym programie informuje jednostkę CPU, aby ta mogła odczytać wyniki z pamięci danych.

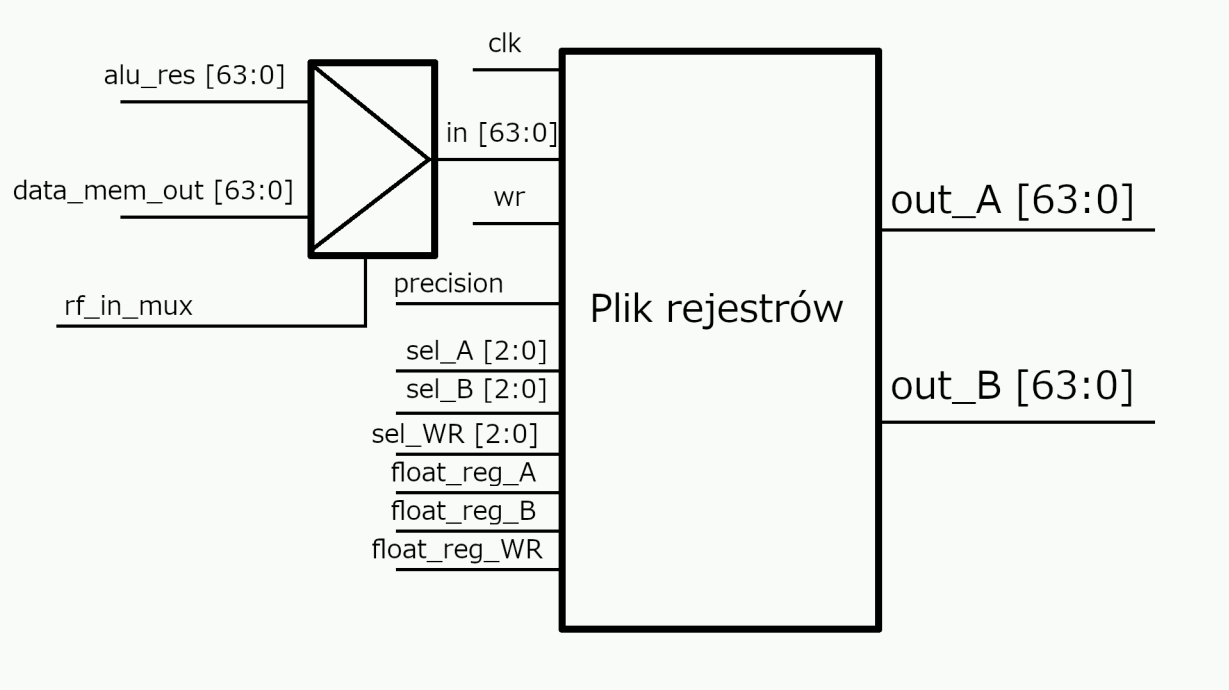
3.2. Rejestry

Rejestry są kluczowymi komórkami pamięci każdego procesora. To one przechowują operandy potrzebne do wszelkich operacji możliwych do wykonania w całej jednostce. Wszystkie rejestry są połączone w jedną całość i przechowywane w jednym miejscu nazwanym *Plikiem Rejestrów.* Układ tych rejestrów został graficznie ukazany na rysunku 3.2.1.



Rys 3.2.1 *Układ rejestrów (źródło: opracowanie własne)*

Zostały zaimplementowane dwa typy rejestrów. Komórki oznaczone literką *s (single precision)* ozaczają rejestry przechowujące liczby pojedynczej precyzji. Posiadają one 32 bity długości i jest ich łącznie 16. Rejestry te pogrupowano parami, a następnie pary te oznaczono jako rejestry *d (double precision)* – podwójnej precyzji. Taki układ został zainspirowany bankiem rejestrów liczb zmiennoprzecinkowych implementowanym w procesorach ARM z rodziny Cortex-M [9].



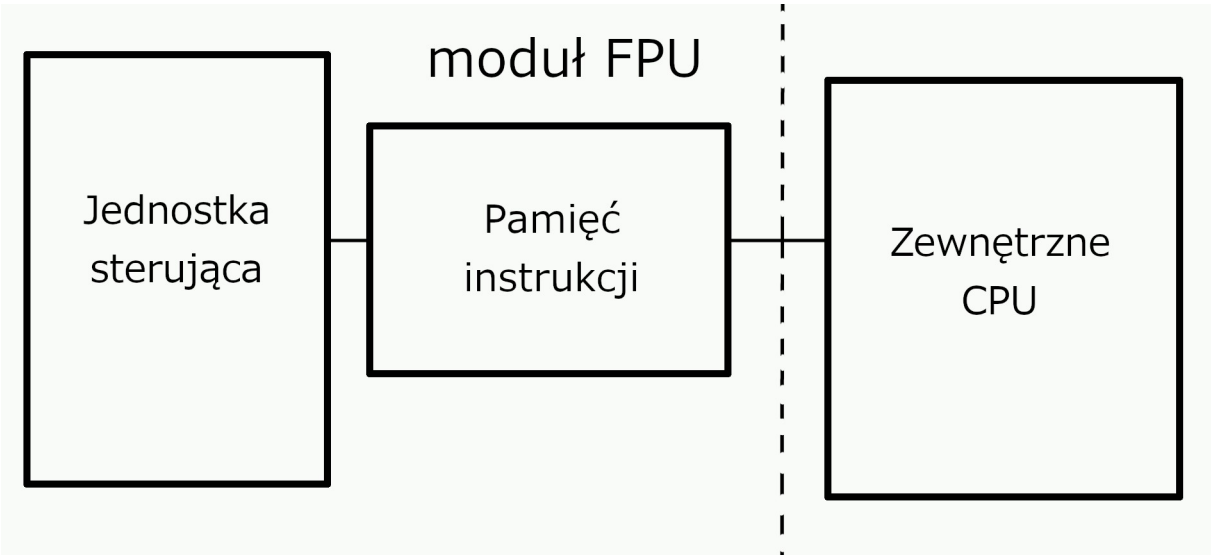
Rys 3.2.2 *Sygnały wchodzące oraz wychodzące z pliku rejestrów (źródło: opracowanie własne)*

Cały plik jest taktowany głównym sygnałem zegarowym. Głównym wejściem, którym wprowadzamy dane do rejestrów to wejście *in* o szerokości 64 bitów. Do tego wejścia mamy możliwość przekazywania wartości wychodzącej z jednostki arytmetycznej, bądź wartości wychodzącej z pamięci danych. Odpowiada za to jednostka centralna wystawiając odpowiedni sygnał na wejście *select* multipleksera – *rf_in_mux*. Dla wartości logicznego zera do rejestrów wchodzi wynik z jednostki arytmetycznej, a dla logicznej jedynki wartość z pamięci danych. Kolejnym istotnym wejściem jest sygnał *wr*. Ustawiając go na 0 sygnalizujemy, że chcemy jedynie odczytać wartości zapisane wewnątrz pliku, natomiast ustawiając sygnał na 1 zapisujemy do niego. W pliku znajdują się dwa typy rejestrów, a co za tym idzie musimy jakoś poinformować jednostkę, na jakich rejestrach chcemy operować. Za to odpowiedzialny jest sygnał *precision*, dla którego 0 to operacje na pojedynczej precyzji, a 1 – na podwójnej. O tym w których miejscach chcemy zapisać wartość decyduje selektor *sel_WR*. Jest on szeroki na 3 bity, co pozwala na zdecydowanie do którego rejestru *d* chcemy zapisać daną wartość. Selektory *sel_A* oraz *sel_B* wybierają natomiast rejestry *d*, z których wartości powinny trafić na wyjścia *out_A* oraz *out_B*. Są to wszystkie wejścia potrzebne do zdefiniowania, aby wykonywać operacje na liczbach podwójnej precyzji. Aby natomiast obliczyć coś na liczbach pojedynczej precyzji należy ustawić sygnał *precision* na 0, a następnie dla danego selektora wybrać rejestr *s,* na którym chcemy operowować. Robimy to za pomocą *float_reg_A, float_reg_B* oraz *float_reg_WR*. Stan niski odpowiada rejestrowi dolnemu, a stan wysoki rejestrowi górnemu w wybranym przez odpowiedni selektor rejestrze

d. Dla przykładu – chcemy wystawić na wyjście *out_A* wartość przechowywaną w rejestrze *s7*, a na wyjście *out_B* wartość z rejestru *s14*. Zaczynamy od ustawienia sygnału *precision = '0'*. *s7* to rejestr *s* komórki *d4,* zatem selektor *sel_A* będzie ustawiony na '100' dwójkowo, co zapisie decymalnym daje 4. Dla tego selektora sygnał *float_reg_A* ustawiamy na 1, gdyż *s7* to górny rejestr pojedynczej precyzji w tej komórce. Dla wyjściowego sygnału *out_B* postępujemy analogicznie. *s14* mieści się w ostatniej komórce *d* na dolnym miejscu. Zatem *sel_B = '111',* a *float_reg_B = '0'*. Sygnał *float_reg_WR* analogicznie odpowiada za wybór dolnego/górnego rejestru *s* komórki *d* wybranej przy pomocy selektora *sel_WR*. W przypadku działań na pojedynczej precyzji sygnały wychodzące z pliku rejestrów zostają zapełniane odpowiednimi wartościami od bitu nr 31 w dół. Górna część rejestrów zapełniana jest zerami.

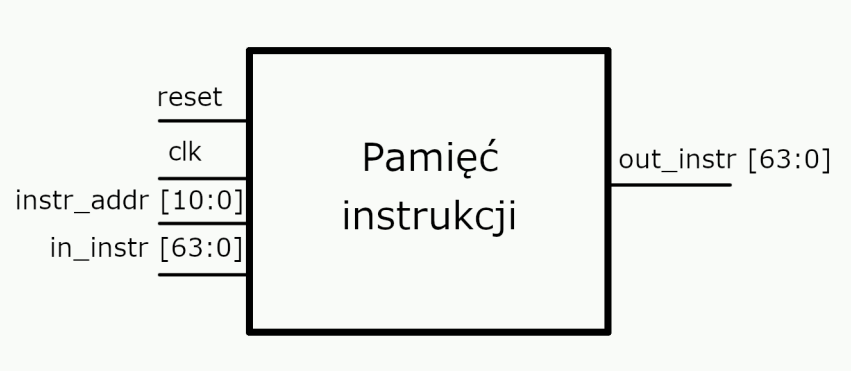
3.3. Pamięć instrukcji

Aby procesor mógł interpretować instrukcje muszą one być najpierw w jakimś miejscu zapisane. Służy ku temu moduł służący za pamięć instrukcji, do którego CPU z zewnątrz jest w stanie ładować dane. Te dane zostają następnie sekwencyjnie ładowane do jednostki sterującej, która je należycie interpretuje.



Rys 3.3.1 *Schemat pamięci instrukcji (źródło: opracowanie własne)*

Pamięć instrukcji składa się z 2048 komórek o szerokości 64 bitów, co pozwala na załadowanie 2048 instrukcji dla jednego programu.

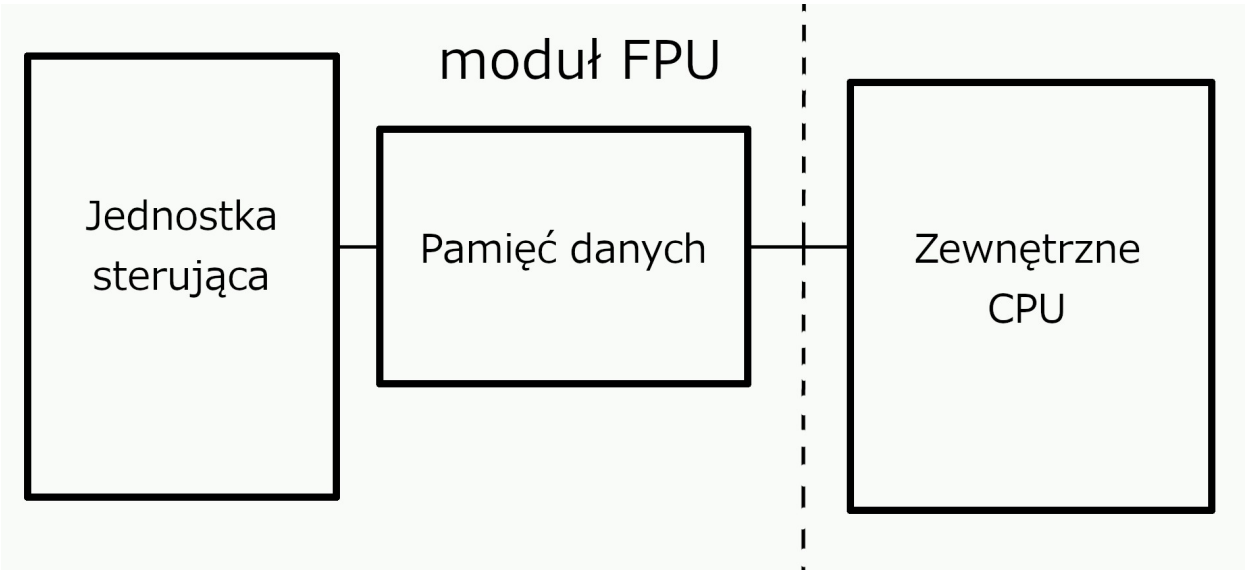


Rys 3.3.2 *Sygnały pamięci instrukcji (źródło: opracowanie własne)*

Do pamięci dane trafiają wejściem *in_instr* pod adres *instr_addr* w momencie gdy sygnał *reset* jest w stanie wysokim. Stan ten informuje o tym, że instrukcje są wgrywane do wewnętrznych rejestrów. Moduł nie posiada latencji, zatem w wewnętrznej pamięci dane są zapisywane na każde zbocze narastające sygnału zegarowego *clk*. Gdy sygnał reset przyjmuje wartość logicznego zera, moduł wystawia na wyjście *out_instr* dane znajdujące się w pamięci pod adresem *instr_addr.*

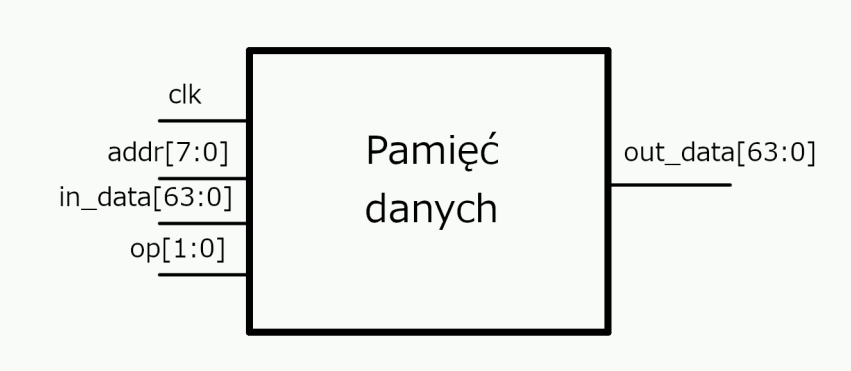
3.4. Pamięć danych

Pamięć danych jest niezbędnym elementem całego projektu. To w niej przechowywane są operandy oraz wyniki przeprowadzonych operacji. Stanowi ona magazyn wspólny dla projektowanego modułu oraz zewnętrznej jednostki.



Rys 3.4.1 *Schemat pamięci danych (źródło: opracowanie własne)*

Pamięć jest podzielona na 256 komórek o szerokości 32 bitów. Można w niej zatem zapisać 256 liczb zmiennoprzecinkowych pojedynczej precyzji albo 128 liczb podwójnej precyzji. Obsługa jednego typu liczb nie wyklucza drugiego – w pamięci możemy jednocześnie przechowywać liczby pojedynczej oraz podwójnej precyzji, należy jednak pamiętać aby nie nadpisywać przypadkowo zajętych adresów. Zapis liczby podwójnej precyzji niesie za sobą ryzyko nadpisania istniejącej liczby znajdującej się pod adresem o jeden większym niż ten do którego następuje zapis. Dla przykładu – jeśli zapisaliśmy wcześniej dane pod adresem 0x04h, musimy uważać, aby nie zapisać liczby podwójnej precyzji pod adresem 0x03h, gdyż zapis taki oznacza zapełnienie komórek 0x03h oraz 0x04h. Format przechowywania danych jest podobny do formatu *Big endian* z tą różnicą, że kolejne komórki nie mają jednego bajtu, lecz 4. Zapis liczby podwójnej precyzji pod adresem 0x08h powoduje podział tejże liczby na dwie 32-bitowe połowy, z których ta bardziej znacząca, zawierająca znak, cechę oraz część mantysy wędruje pod adres 0x08h, a druga połowa do komórki oznakowanej następnym adresem – 0x09h. Poniższy rysunek przedstawia wyprowadzenia sygnałów wchodzących oraz wychodzących z modułu pamięci danych.



Rys 3.4.2 *Sygnały pamięci danych (źródło: opracowanie własne)*

Sygnał zegarowy jest podłączony wejściem oznaczonym *clk*. Wejście *addr* odpowiada za adres komórki, na której chcemy przeprowadzić konkretną operację. Możliwych do wykonania operacji jest 4. Wybieramy je za pomocą sygnału *op*, którego wartości definiują następujące działania:

- ➔ 0x00h – Odczyt liczby zmiennoprzecinkowej pojedynczej precyzji,
- ➔ 0x01h – Zapis liczby zmiennoprzecinkowej pojedynczej precyzji,
- ➔ 0x02h – Odczyt liczby zmiennoprzecinkowej podwójnej precyzji,
- ➔ 0x03h – Zapis liczby zmiennoprzecinkowej podwójnej precyzji.

Na wejście *in_data* wpisane powinny zostać dane, które są przeznaczone do zapisu w komórce wybranej adresem na wejściu *addr*. Jeśli *op* wskazuje na zapis liczby pojedynczej precyzji istotne są dolne 32 bity, natomiast przy zapisie liczby podwójnej precyzji – całe 64 bity. Analogicznie w przypadku czytania wartości – na wyjściu *out_data* pojawiają się istotne dla użytkownika informacje gdy *op* jest równe 0x00h lub 0x02h, przy czym dla 0x00h istotnych jest jedynie 32 dolnych bitów, a przy 0x02h cała szerokość wyjścia z modułu. Moduł nie posiada latencji odczytu - wartości są odczytywane na narastającym zboczu *clk* w tym samym takcie zegara. Zapis z kolei zajmuje jeden takt zegarowy. W przypadku zapisu do pamięci, na wyjście *out_data* wystawiane jest 0.

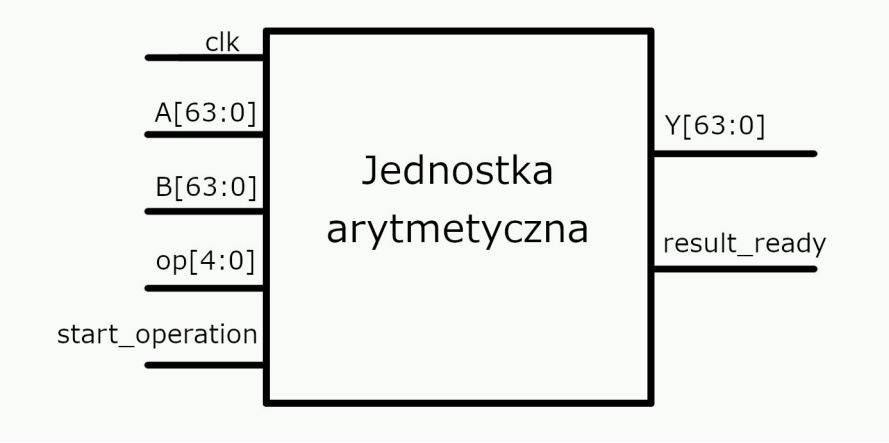
```
reg [31:0] memory [0:255];

assign out_data = (op == 2'b00) ? {32'h00000000, memory[addr][31:0]} :
            ((op == 2'b10) ? {memory[addr][31:0], memory[addr + 1][31:0]} : 64'h0000000000000000);

always @(posedge clk)
begin
    if(op == 2'b01) // zapis liczby pojedynczej precyzji
    begin
        memory[addr][31:0] <= in_data[31:0];
    end
    else if(op == 2'b11) // zapis liczby podwójnej precyzji
    begin
        memory[addr][31:0] <= in_data[63:32];
        memory[addr + 1][31:0] <= in_data[31:0];
    end
end
```

Rys 3.4.3 *Implementacja pamięci danych*

W przypadku zapisywania liczby podwójnej precyzji pod adres ostatniej dostępnej komórki, dolna część liczby zostanie zapisana na samym początku pamięci pod adres 0x00h.

3.5. Jednostka arytmetyczna

Sercem całego projektu jest jednostka arytmetyczna, w której odbywają się wszystkie operacje arytmetyczne. Nie jest to jednostka arymtetyczno – logiczna jak w większości procesorów ze względu na brak zaimplementowanych operacji logicznych. Moduł wykonuje jedynie obliczenia na dostarczonych danych – dodawanie, odejmowanie, mnożenie, dzielenie, odwrotność, pierwiastek kwadratowy, logarytm naturalny oraz eksponenta.



Rys 3.5.1 *Wejścia i wyjścia jednostki arytmetycznej (źródło: opracowanie własne)*

Jednostka jest taktowana głównym sygnałem zegarowym całego modułu *clk*. Na wejście *A* wpisywany jest pierwszy operand operacji, a na wejście *B* drugi operand. Wejścia te są 64 bitowe, dlatego w przypadku operacji na liczbach pojedynczej precyzji wykorzystywane jest jedynie 32 dolnych bitów, a górne 32 bity uzupełniane są zerami. O precyzji świadczy najstarszy bit wejścia *op*. Gdy bit ten ustawiony jest na zero, wykonywane są operacje na liczbach pojedynczej precyzji, natomiast dla wartośći logicznej jedynki – na liczbach podwójnej precyzji. Reszta bitów mówi o tym, jaka operacja jest wykonywana:

- 0x00h – dodawanie,
- 0x01h – odejmowanie,
- 0x02h – mnożenie,
- 0x03h – dzielenie,
- 0x04h – odwrotność,
- 0x05h – pierwiastek,
- 0x06h – logarytm naturalny,
- 0x07h – eksponenta.

Operacja rozpoczywana jest w momencie wykrycia zbocza narastającego na wejściu *start_operation*. Wewnątrz jednostki dla każdej operacji zostały dodane dwa *IP Core* wykonujące daną operację na liczbach pojedynczej oraz podwójnej precyzji. Ze względu na posiadaną latencję tych modułów zaimplementowana została logika, która informuje jednostkę sterującą gdy działanie dobiegnie końca. W tym celu wykorzystuje sygnał *result_ready*, który przyjmuje wartość logicznej jedynki, gdy wynik operacji jest "gotowy do odbioru". W momencie czekania na wynik, licznik programu (*ang. program counter*) nie jest inkrementowany, lecz cała reszta układu czeka na jednostkę sterującą. Gotowy wynik zostaje zapisany w pamięci danych albo jednym z rejestrów, w zależności od bieżącej instrukcji. Na przykładzie dodawania omówię implementację programową wszystkich działań.

```
// sum
wire[31:0] w_sum_f_result_tdata;
wire w_sum_f_result_ready;

alu_add_f sum_f (
    .aclk(clk),
    .s_axis_a_tvalid((op[4:0] == 5'b00000) ? start_operation : 0),
    .s_axis_a_tdata(A[31:0]),
    .s_axis_b_tvalid((op[4:0] == 5'b00000) ? start_operation : 0),
    .s_axis_b_tdata(B[31:0]),
    .m_axis_result_tvalid(w_sum_f_result_ready),
    .m_axis_result_tdata(w_sum_f_result_tdata)
);
```

***Rys 3.5.2** Wykorzystanie FPU IP Core*

Na rysunku 3.6.2 pokazana została instancja *FPU IP Core*. Została ona wcześniej skonfigurowana do dodawania liczb zmiennoprzecinkowych pojedynczej precyzji. Posiada ona latencję równą 1 takt zegarowy, zatem mija 1 cykl zegara między zboczem narastającym sygnału *start_operation* a zboczem narastającym sygnału *result_ready* dla tego *IP Core*.

```
wire[63:0] w_sum_result_tdata;

mux2 sum_mux (
    .in1({32'h00000000, w_sum_f_result_tdata}),
    .in2(w_sum_d_result_tdata),
    .sel(op[4]),
    .out(w_sum_result_tdata)
);
wire w_sum_result_ready;
assign w_sum_result_ready = (op[4] == 0) ? w_sum_f_result_ready : w_sum_d_result_ready;
```

Rys 3.5.3 *Multipleksowanie wyjść dwóch dodawarek*

Na rysunku 3.6.3 pokazane zostało multipleksowanie wyjść z dodawarek pojedynczej oraz podwójnej precyzji. Multiplekser posiada dwa wejścia o szerokości 64 bitów, selektor oraz wyjście również 64 bitowe. W zależności od typu liczb, na których wykonujemy operację, przekazany dalej zostaje wynik z dodawarki 32 lub 64 bitowej.

```
assign Y =  (op[3:0] == 4'b0000) ? w_sum_result_tdata : ((op[3:0]  == 4'b0001) ? w_sub_result_tdata :
            ((op[3:0] == 4'b0010) ? w_mul_result_tdata : ((op[3:0] == 4'b0011) ? w_div_result_tdata :
            ((op[3:0] == 4'b0100) ? w_rec_result_tdata : ((op[3:0] == 4'b0101) ? w_sqrt_result_tdata :
            ((op[3:0] == 4'b0110) ? w_log_result_tdata : ((op[3:0] == 4'b0111) ? w_exp_result_tdata : B)))))));

assign result_ready = (op[3:0] == 4'b0000) ? w_sum_result_ready : ((op[3:0]  == 4'b0001) ? w_sub_result_ready :
                      ((op[3:0] == 4'b0010) ? w_mul_result_ready : ((op[3:0] == 4'b0011) ? w_div_result_ready :
                      ((op[3:0] == 4'b0100) ? w_rec_result_ready : ((op[3:0] == 4'b0101) ? w_sqrt_result_ready :
                      ((op[3:0] == 4'b0110) ? w_log_result_ready : ((op[3:0] == 4'b0111) ? w_exp_result_ready : 0)))))));
```

Rys 3.5.4 *Przypisanie odpowiedniego wyjścia z całej jednostki*

Ostatnim etapem jest wyprowadzenie wyjścia zawerającego odpowiedni wynik na zewnątrz modułu. Na wyjście *Y* trafia wynik odpowiadający pożądanej operacji wybierany na podstawie dolnych 4 bitów sygnału *op.* Zaimplementowanych operacji jest 8, aczkolwiek dodanie większej ilości operacji nie stanowi problemu, wystarczy jednostkę rozbudować wzorując się na poprzednich działach. W przypadku gdy ilość operacji będzie większa niż liczba możliwa do zapisania w *op*, należałoby to wejście rozszerzyć. Nie zawsze natomiast chcemy wykonywać jakiekolwiek operacje arytmetyczne. W przypadku załadowania wartości bezpośredniej do rejestru lub pamięci, wartość ta przechodzi przez jednostkę bez żadnych zmian. Aby uzyskać taki rezultat należy na wejście *op* wpisać wartość która nie odpowiada

żadnej z operacji (rozsądnie jest trzymać się jednej, konkretnej wartości, np. 0x0Fh z uwagi na możliwość dalszej rozbudwy bazy operacji). Wtedy liczba z wejścia *B* zostanie przekierowana bezpośrednio na wyjście *Y*.

3.6. Jednostka sterująca

Po załadowaniu danych oraz instrukcji do pamięci na opadające zbocze sygnału *reset* soft procesor zaczyna wykonywać program. Program jest wykonywany instrukcja po instrukcji, a modułem odpowiedzialnym za interpretację tych instrukcji jest właśnie jednostka sterująca. Zawiera ona indeks aktualnie przetwarzanej instrukcji (*program counter*), który połączony jest z wejściem adresowym pamięci instrukcji. Indeks ten inkrementowany jest po wykonaniu każdej instrukcji. Jednostka sterująca posiada cztery wejścia. Są to sygnał zegarowy, reset, wejście na dane zawierające instrukcję oraz wejście informujące o tym, że jednostka arytmetyczna zakończyła wykonywać operację. Wyjść natomiast jest 17 i zostały one odpowiednio pogrupowane na wyjścia związane z jednostką arytmetyczną, plikiem rejestrów, pamięcią danych oraz pamięcią instrukcji. Jedyne wyjście które nie zalicza się do żadnej kategorii jest sygnał *program_ready* informujący o końcu programu. Z pamięcią instrukcji wiąże się jedno wyjście – *program_counter* stanowiący adres kolejnej instrukcji do załadowania. Z pamięcią danych wiążą się dwa wyjścia – *dmem_addr* oraz *dmem_op*. Pierwsze to wejście adresowe pamięci danych a drugie ro rodzaj operacji wykonywanej na pamięci. Wyjścia związane z jednostką arytmetyczną to *alu_start_operation*, czyli sygnał rozpoczęcia operacji arutymetycznej, *alu_b_sel* – selektor multipleksera którego wyjście trafia na wejście *B* do jednostki, *alu_op* – indeks wykonywanej operacji arytmetycznej i ostatecznie sygnał *alu_b_imm* zawierający wartość bezpośrednią z instrukcji. Reszta wyjść to wejścia wchodzące do pliku rejestrów oraz multipleksera znajdującego się nad nim (rys 3.1.1).

```
always @(posedge clk)
begin
    if(!reset && !r_program_ready)
    begin
        if(r_alu_start_operation == 1'b1)
        begin
            r_alu_start_operation <= 1'b0;
        end
        if(alu_result_ready)
        begin
            waiting_for_alu_result <= 1'b0;
            r_rf_wr <= 1'b1;
        end
        if(!waiting_for_alu_result)
        begin
            pc <= pc + 1;
            r_instruction_data <= instr_data;
            r_alu_start_operation <= ((instr_data[36:33] < 8) ? 1'b1 : 1'b0);
            waiting_for_alu_result <= ((instr_data[36:33] < 8) ? 1'b1 : 1'b0);
            r_rf_wr <= ((instr_data[36:33] < 8) ? 1'b0 : instr_data[51]);
            r_program_ready <= instr_data[63];
        end
    end
    if(reset)
    begin
        pc <= 8'b0;
        r_program_ready <= 1'b0;
    end
end
```

Rys 3.6.1 *Logika jednostki sterującej*

Na rysunku 3.6.1 pokazana została logika zawarta w jednostce sterującej wykonująca się na każde narastające zbocze sygnału zegarowego. Wewnątrz pierwszej instrukcji warunkowej zawarty jest kod wykonujący się podczas trwania programu – jest to część odpowiedzialna za interpretację instrukcji. Sygnał *r_alu_start_operation* ustawiany jest na stan wysoki w momencie gdy interpretowana instrukcja zawiera operację arytmetyczną. Sygnał ten wysyłany jest do jednostki arytmetycznej, która oddaje informację zwrotną w postaci sygnału *alu_result_ready* – który sygnalizuje koniec wykonywania operacji wewnątrz jednoski arytmetycznej. Na ten sygnał zerowany jest sygnał *waiting_for_alu_result* oraz ustawiany sygnał *r_rf_wr* mówiący o zapisie aktualnego wyniku do jednego z rejestrów. Gdy moduł nie czeka na wynik z jednostki sterującej inkrementowany jest indeks instrukcji, do podręcznego rejestru *r_instruction_data* trafia kolejna instrukcja zaciągnięta z pamięci instrukcji. Następnie w zależności od instrukcji moduł przechodzi do stanu czekania na wynik z jednostki arytmetycznej bądź nie. Jeśli pobrana została instrukcja *end* na sygnał *r_program_ready* ustawiany zostaje stan wysoki co świadczy o zakończeniu programu.

```
assign alu_start_operation = r_alu_start_operation;
assign program_counter = pc;
assign program_ready = r_program_ready;

assign alu_b_imm = r_instruction_data[31:0];
assign alu_b_sel = r_instruction_data[32];
assign alu_op = r_instruction_data[37:33];
assign rf_sel_B = r_instruction_data[40:38];
assign rf_sel_A = r_instruction_data[43:41];
assign rf_sel_WR = r_instruction_data[46:44];
assign rf_float_reg_b = r_instruction_data[47];
assign rf_float_reg_a = r_instruction_data[48];
assign rf_float_reg_wr = r_instruction_data[49];
assign rf_precision = r_instruction_data[50];
assign rf_wr = r_rf_wr;
assign rf_in_mux = r_instruction_data[52];
assign dmem_op = r_instruction_data[54:53];
assign dmem_addr = r_instruction_data[62:55];
```

Rys 3.6.2 *Rozdzielenie instrukcji na sygnały*

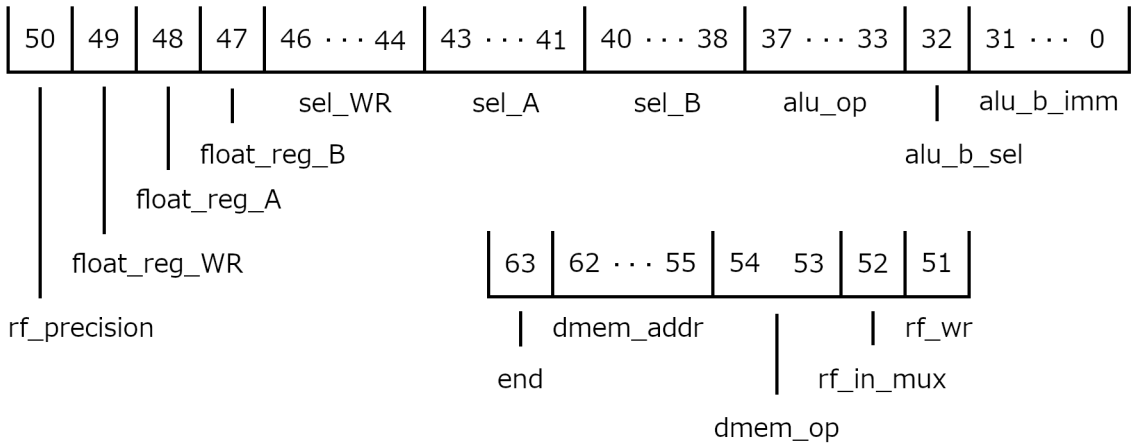
W dalszej części jednostki sterującej wyjścia są przypisywane do odpowiednich segmentów aktualnie przetwarzanej instrukcji. Wyjątek stanowi sygnał *rf_wr*, który przypisywany jest do rejestru roboczego ze względu na latencję jednostki arytmetycznej. Zapis do rejestru włączany jest w momencie otrzymania poprawnego wyniku przetwarzanej operacji. Sygnał *alu_start_operation* również przypisywany jest do rejestru roboczego *r_alu_start_operation*, gdyż jest to jedynie impuls trwający jeden cykl zegarowy. Jest on zerowany przed pobraniem kolejnej instrukcji.

4. Assembler

Aby umożliwić wygodne pisanie instrukcji dla modułu, napisany został program w środowisku MALTAB. Program ten ładuje instrukcje z pliku tekstowego, przetwarza je na ciągi binarne oraz zapisuje do pliku tekstowego. Plik ten następnie jest wykorzystywany w symulacyjnym testowaniu modułu. W symulacyjnym module testowym plik jest wczytywany oraz pamięć instrukcji zapełniana jest jego zawartością.

4.1. Interpretacja instrukcji

Instrukcje trafiające do pamięci instrukcji to 64 bitowe ciągi binarne. Interpretuje je jednostka sterująca – dzieli je na kawałki i wystawia odpowiednie sygnały na odpowiednich liniach. Rysunek 4.1.1 przedstawia nazwy wszystkich bitów instrukcji. Nazwy te odpowiadają sygnałom wewnątrz modułu i zostanę one omówione poniżej.



Rys 4.1.1 *Układ instrukcji (źródło: opracowanie własne)*

Zaczynając od najwyższego bitu – *end* sygnalizuje instrukcję końca programu. Po odczytaniu instrukcji z tym bitem w stanie wysokim jednostka sterująca wystawia sygnał *program_ready* sygnalizujący ukończenie programu. Jeśli instrukcja ta jest wykonywana po instrukcji działania arytmetycznego (co nie powinno zachodzić, ponieważ w zamyśle modułu powinno się wyniki zapisywać w pamięci danych), moduł sygnalizuje o końcu programu w momencie gdy jednostka arytmetyczna zakończy liczyć. *dmem_addr* to 8 bitów przeznaczonych na wybór adresu w pamięci danych. Z komórką pod tym adresem można wykonać działania opisane w rozdziale 3.4, a sygnałem za pomocą którego możemy wybrać te operacje to *dmem_op*. Kolejnym sygnałem jest *rf_in_mux*. Jest to selektor multipleksera,

którego wejściami są wyjście z jednostki arytmetycznej oraz wyjście z pamięci danych. Wyjście multipleksera trafia natomiast na wejście do pliku rejestrów. Sygnał *rf_wr* informuje o tym, czy dane na wejściu pliku rejestrów powinny zostać zapisane w jednym z rejestrów. Jako że moduł obsługuje liczby zmiennoprzecinkowe zarówno pojedynczej jak i podwójnej precyzji, konieczny jest sygnał informujący o bieżącym typie, na którym wykonywane jest działanie. Odpowiada za to bit oznaczony jako *rf_precision*. Następne trzy sygnały mają znaczenie jeśli bieżąca operacja wykonywana jest na liczbach pojedynczej precyzji, innymi słowy dla *rf_precision = 0*. W innym wypadku sygnały te są ignorowane. Wybór konkretnego rejestru 32 bitowego polega na wyborze odpowiedniego rejestru 64 bitowego oraz jego górnej bądź dolnej części. Dla wartości '0' jest to dolny, a dla '1' górny rejestr. Sygnał *sel_WR* wybiera górną bądź dolną część tego rejestru 64 bitowego, do którego chcemy zapisać. Analogicznie postępuje się z wyborem górnej bądź dolnej części rejestrów 64 bitowych, z których chcemy odczytać wartości i wystawić na wyjścia *out_A* oraz *out_B* z pliku rejestrów. Kolejne trzy sygnały *sel_WR, sel_A* oraz *sel_B* to selektory rejestrów 64 bitowych, na których chcemy przeprowadzić operacje kolejno zapisu, odczytu na wyjście *out_A* oraz odczytu na wyjście *out_B*. Następne pięć bitów nazwane jako *alu_op* wybiera operację do przeprowadzenia w jednostce arytmetycznej. Sygnał ten został opisany szczegółowo w rozdziale 3.5. Bit *alu_b_sel* to selektor multipleksera decydujący o tym, czy do jednostki arytmetycznej na wejście *B* trafia wartość bezpośrednia z instrukcji zapisana w 32 bitach *alu_b_imm,* czy wyjście *out_B* z pliku rejestrów.

4.2. Lista instrukcji

Zaimplementowanych zostało 25 instrukcji. 16 instrukcji związanych jest z obliczeniami arytmetycznymi, gdzie 8 instrukcji jest dla liczb pojedynczej precyzji a pozostałe 8 dla liczb podwójnej precyzji. Pozostałe instrukcje to przemieszczanie wartości między rejestrami, ładowanie i zapis do pamięci danych oraz instrukcja końca programu (*end)*. Te instrukcje dla liczb pojedynczej precyzji zostały przedstawione poniżej:

- ➔ *fldf sx, #addr* - załadowanie do rejestru *sx* wartości spod adresu *addr,*
- ➔ *fstf #addr, sx* - zapisanie pod adresem *addr* wartości z rejestru *sx,*
- ➔ *fstfi #addr, imm* - zapisanie pod adresem *addr* wartości bezpośrendniej *imm,*
- ➔ *fmovf sx, sy* - zapisanie w rejestrze *sx* wartości z rejestru *sy,*
- ➔ *fmovfi sx, imm* - zapisanie w rejestrze *sx* wartości bezpośredniej *imm.*

Operacje te dla liczb podwójnej precyzji zostały okrojone do trzech instrukcji:

- ➔ *fldd dx, #addr* - załadowanie do rejestru *dx* wartości spod adresu *addr,*
- ➔ *fstd #addr, dx* - zapisanie pod adresem *addr* wartości z rejestru *dx,*
- ➔ *fmovd dx, dy* - zapisanie w rejestrze *dx* wartości z rejestru *dy.*

Zostało zaimplementowanych 8 operacji arytmetycznych dla liczb pojedynczej oraz podwójnej precyzji, co daje 16 instrukcji assemblera. Dla liczb pojedynczyej precyzji pierwszy człon instrukcji zakończony jest literą *f* (ang. float), a dla podwójnej precyzji literą *d* (ang. double).

- *faddf sx, sy, sz* - zapisanie w rejestrze *sx* sumy *sy* oraz *sz,*
- *fsubf sx, sy, sz* - zapisanie w rejestrze *sx* różnicy *sy* oraz *sz,*
- *fmulf sx, sy, sz* - zapisanie w rejestrze *sx* iloczynu *sy* oraz *sz,*
- *fdivf sx, sy, sz* - zapisanie w rejestrze *sx* ilorazu *sy* oraz *sz,*
- *frecf sx, sy* - zapisanie w rejestrze *sx* odwrotności *sy,*
- *fsqrtf sx, sy* - zapisanie w rejestrze *sx* pierwiastka kwadratowego z *sy,*
- *flogf sx, sy* - zapisanie w rejestrze *sx* logarytmu naturalnego z *sy,*
- *fexpf sx, sy* - zapisanie w rejestrze *sx* ekxponenty z *sy.*

Dla liczb podwójnej precyzji instrukcje różnią się jedynie ostatnią literą pierwszego członu – dla przykładu operacja dodawania dwóch liczb podwójnej precyzji będzie się nazywała *faddd*.

4.3. Implementacja programowa

W środowisku *MATLAB* został napisany program interpretujący instrukcje wymienione w rozdziale 4.2. Implementacja poszczególnych operacji jest analogiczna, instrukcje różnią się jedynie precyzją oraz ilością parametrów, dlatego w dalszej części zostanie omówiona jedynie operacja dodawania. Program składa się z załadowania pliku tekstowego, głównej pętli, w której ładowane są kolejne instrukcje oraz zapis przetworzonych instrukcji.

```
programFile = fopen('zad.asm', 'r');
parsedFile = fopen('../FPU.sim/sim_1/behav/xsim/zad.mc', 'w');
line = fgets(programFile);
while line ~= -1
    instruction = parseLine(line);
    if(instruction ~= "\n")
        fprintf(parsedFile, instruction);
    end
    line = fgets(programFile);
end
fclose(programFile);
fclose(parsedFile);
```

***Rys 4.3.1** Pętla główna programu*

W głównej pętli programu najważniejsza linia to ta, w której wykonywana jest funkcja *parseLine.* Przyjmuje ona jako parametr łańcuch znaków, w postaci instrukcji assemblera, a zwraca łańcuch znaków w postaci ciągu binarnego. Na początku funkcji wydobywane jest pierwsze słowo w linii, a następnie utworzony *switch* w zależności od tego słowa. Na rysunku 4.3.2 przedstawiony jest przypadek dodawania *faddf*.

```
case 'faddf'
    [sx remain] = strtok(remain, ' ,');
    [sy remain] = strtok(remain, ' ,');
    [sz remain] = strtok(remain, ' ,;');
    sz = deblank(sz);
    [sel_A_dx, sel_A_low] = sxTodxAndLow(sy);
    [sel_B_dx, sel_B_low] = sxTodxAndLow(sz);
    [sel_WR_dx, sel_WR_low] = sxTodxAndLow(sx);
    d_mem_op = '00';
    rf_in_mux = '0';
    rf_write = '1';
    rf_precision = '0';
    out = strcat(out, addrToBin('0x00'), d_mem_op, ...
        rf_in_mux, rf_write, rf_precision, ...
        sel_WR_low, sel_A_low, sel_B_low, ...
        sel_WR_dx, sel_A_dx, sel_B_dx, '00000', ...
        '0', immToBin('0x00000000'));
```

Rys 4.3.2 *Interpretacja instrukcji dodawania liczb pojedynczej precyzji*

Na początku wydobywane są potrzebne parametry, których ilość jest zależna od instrukcji. Dla dodawania są to *sx, sy* oraz *sz*. Na podstawie tych parametrów funkcja *sxTodxAndLow* wyznaczana selektory *sel_A, sel_B* i *sel_WR* oraz odpowiadające im górne bądź dolne rejestry 32 bitowe o których mowa w rozdziale 3.2. Kolejne sygnały (*d_mem_op, rf_in_mux, rf_write, rf_precision*) ustawiane są osobno dla każdej operacji. Ostatecznie wyznaczany jest łańcuch znaków *out* zawierający przetworzoną instrukcję. Funkcja *addrToBin* przyjmuje 8 bitowy adres w postaci heksadecymalnej i zwraca go w postaci binarnej. Analogicznie zachowuje się funkcja *immToBin* z tą różnicą, że przyjmuje jako parametr wartość 32 bitową.

4.4. Przykładowy program

Aby zilustrować łatwość obsługi języka napisany został przykładowy program w edytorze plików tekstowych *Notepad++*.

```
fmovfi s0, 0xc0a0a3d7; wpisanie -5.02 do resejtru s0
fldf s1, 0x09; załadowanie 2.58 spod adresu 0x09 do s1
faddf s2, s0, s1; suma s2 = s0 + s1 = -2.44 - wynik z programu
fmulf s3, s0, s2; iloczyn s3 = s0 * s2 = 12.2488 - wynik z programu
fstf 0x0F, s3; zapisanie wyniku z s3 do pamięci danych pod adres 0x0F
fstfi 0x12, 0xc0a0a3d7; wpisanie do pamięci -5.02 pod adres 0x12

fldd d3, 0x00; załadowanie double 2.555973 do d3
fldd d4, 0x02; załadowanie double 4.982 do d4
faddd d5, d3, d4; suma d5 = d3 + d4 = 7.537973 - wynik z programu
fmuld d6, d4, d5; iloczyn d6 = d4 * d5 = 37.554181486000004 - wynik z programu

fstd 0x04, d6; zapisanie wyniku z d6 do pamięci danych pod adres 0x04

end; zakończenie programu
```

Rys 4.4.1 *Implementacja przykładowego programu*

Jest to prosty program wykorzystujący funkcjonalności ładowania oraz zapisywania danych w pamięci. Wykonywane są operacje dodawania oraz mnożenia na liczbach zarówno pojedynczej jak i podwójnej precyzji. O zakończeniu programu informuje instrukcja *end*. Kompilator w środowisku *MATLAB* ignoruje komentarze użytkownika znajdujące się za średnikiem w każdej linii. Ponadto ignorowane są również puste linie nie zawierające żadnych instrukcji. Po wykonaniu skryptu w środowisku *MATLAB* otrzymujemy plik ukazany na rysunku 4.4.2.

```
00000000000010000000000000111111100000101000001010001111010111
00000100100110100000000000011110000000000000000000000000000000
00000000000010001001000000000000000000000000000000000000000000
00000000000010100001000001000100000000000000000000000000000000
00000111101000001000000001011110000000000000000000000000000000
00001001001000000000000000111111100000101000001010001111010111
00000000010111000011000000111110000000000000000000000000000000
00000001010111000100000000111110000000000000000000000000000000
00000000000011000101011001000000000000000000000000000000000000
00000000000011000110100101100100000000000000000000000000000000
00000100110010000000011011111000000000000000000000000000000000
10000000000000000000000000111110000000000000000000000000000000
```

Rys 4.4.2 *Plik wynikowy kompilacji programu z rysunku 4.4.1*

Rezultat kompilacji w postaci pliku tekstowego zapisywany jest w strukturze katalogu projektu całego modułu projetkowanego w *Vivado*. Informacje o korzystaniu z pliku wynikowego przedstawione zostaną w rozdziale 5.

5. Testy

5.1. Rodzaje przeprowadzonych testów

Testowanie oprogramowania jest bardzo istotnym elementem jego tworzenia. Wraz z projektowaniem poszczególnych modułów pisane były testy jednostkowe weryfikujące poprawność działania każdego elementu z osobna. Dzięki takiej metodyce wychwytywane były błędy już w początkowych fazach istnienia soft procesora. Takie podejście zapewnia głębokie zrozumienie działania wszystkich komponentów. Pozwala ono również uniknąć sytuacji, w której trafiamy na błąd programu i analizujemy cały projekt, podczas gdy nieprawidłowość tkwi w nieprzetestowanej jednostce, na przykład multiplekserze, którego selektor wybiera zły sygnał na wyjście. Dzięki temu paradoksalnie wkładając dodatkowy czas na tworzenie środowiska testowego, zyskujemy czas, który zostałby utracony na rzecz szukania błędów. Poza testami jednostkowymi utworzono jeden test integracyjny weryfikujący poprawność działania całego soft procesora. Wspomniane wcześniej testy jednostkowe oraz test integracyjny to testy symulacyjne utworzone jako *testbench* w środowisku *Vivado*. Testy symulacyjne umożliwiają **bezpieczną** weryfikację działania programu. Bezpieczną w takim sensie, że pozwalają wykryć nieprawidłowości, które mogłyby potem kosztować uszkodzeniem docelowego układu rekonfigurowalnego. Jak wiadomo układy te kosztują często nawet po kilka tysięcy złotych, także warto korzystać z funkcjonalności symulacyjnej wbudowanej w środowiska do projektowania układów FPGA. Gdy symulacyjnie przetestowano każdy komponent, soft procesor jako całość oraz wszystkie możliwe operacje w nim zaimplementowane, przystąpiono do testowania oprogramowania w sprzęcie. Wybrano płytkę *PicoZed FMC Carrier Card V2* z zamontowanym układem rekonfigurowalnym *PicoZed 7010* ze względu na jej dostępność u opiekuna pracy. Testowanie sprzętowe oraz w ogóle wgrywanie programu do układu rekonfigurowalnego umożliwiło środowisko *Xilinx Software Development Kit (SDK)*. Testy w tym środowisku pisane były w języku C. Weryfikacja polegała na porozumieniu między komputerem a układem rekonfigurowalnym za pomocą portu szeregowego, co zostało dokładnie opisane w rozdziale 5.3.

5.2. Testy symulacyjne

Wraz z budowaniem poszczególnych komponentów soft procesora pisane były odpowiednie środowiska testowe. W sumie napisanych zostało siedem testów jednostkowych oraz jeden test integracyjny. Symulacyjnie przetestowane były multipleksery zawierające dwa wejścia o szerokościach 64 bitów, rejestry 32 oraz 64 bitowe, plik rejestrów, jednostka arytmetyczna oraz pamięci instrukcji i danych. Test integracyjny objął cały soft procesor

razem z ładowaniem danych do pamięci danych, ładowaniem instrukcji do pamięci instrukcji oraz odczytem wyników z pamięci danych. Testy wszystkich modułów były istotne, natomiast poniżej zostanie omówiony jedynie test integracyjny weryfikujący poprawność działania całego soft procesora.

```
initial
begin
    file1 = $fopen("zad.mc", "rb");
    if(!file1) $display("the file \"zad.mc\" could not be found");
    else begin
        while(!$feof(file1)) begin
            $fscanf(file1, "%b\n", instruction_set[i1]);
            i1 = i1 + 1;
        end
    end

    current_external_instruction = instruction_set[0];

    file2 = $fopen("data_memory.mc", "rb");
    if(!file2) $display("the file \"data_memory.mc\" could not be found");
    else begin
        while(!$feof(file2)) begin
            $fscanf(file2, "%b\n", data_memory_set[i2]);
            i2 = i2 + 1;
        end
    end

    current_external_data = data_memory_set[0];
    #4;
    while(1)
    begin
        #1; clk = !clk;
    end
end
```

Rys 5.2.1 *Sekcja "initial" testu integracyjngo*

Na rysunku 5.2.1 pokazana została inicjalizacja środowiska testowego. W pierwszej częsci z pliku *zad.mc* wczytywane zostają instrukcjie do wcześniej zadeklarowanej tablicy rejestrów o szerokości 64 bitów. Rejestr przechowujący bieżącą instrukcję zapełniany jest pierwszą instrukcją z pliku tekstowego. Następnie analogicznie zapełniane zostają rejestry przechowujące dane, które docelowo mają trafić do pamięci danych. Następnie generowane jest lekkie opóźnienie oraz w nieskończonej pętli sygnał zegarowy.

```
always @(posedge clk)
begin
    cnt <= cnt + 1;
    instr_addr <= instr_addr + 1;
    data_addr <= data_addr + 2;
    current_external_data = data_memory_set[data_addr / 2];
    current_external_instruction = instruction_set[instr_addr];

    if(cnt == (i1 > i2 ? i1 : i2))
    begin
        reset <= 1'b0;
    end
    if(cnt == i2)
    begin
        data_op <= 2'b00;
    end
end
```

Rys 5.2.2 *Przesyłanie instrukcji oraz danych do modułu soft procesora*

Na rysuknu 5.2.2 ukazano zmiany następujące przy każdym narastającym zboczu sygnału zegarowego. Idąc od góry inkrementowana jest zmienna *cnt* licząca zbocza narastające sygnału *clk*. Następnie inkrementowany jest adres instrukcji w celu wysłania kolejnego ciągu bitów do pamięci instrukcji. Ciąg ten zapisywany jest w rejestrze *current_external_instrucion*. Jako że jednocześnie z instrukcjami wgrywane są również dane, na których wykonywane są operacje, rejestr *data_addr* śledzi adres aktualnie wgrywanych danych do pamięci. Dane te przechowywane są w rejestrze *current_external_data*. Komórki w pamięci danych są 32 bitowe, zatem aby wgrywać po kolei instrukcje 64 bitowe należy adres zwiększać o 2. Tablica *data_memory_set* zawiera 64 bitowe rejestry, dlatego z każdym sygnałem zegarowym jej indeks jest zwiększany o 1. W momencie gdy licznik *cnt* doliczy do ilości instrukcji lub ilości danych, w zależności od tego, czego jest więcej następuje zwolnienie sygnału reset – jest to znak zakończenia ładowania instrukcji oraz danych. W chwili gdy licznik *cnt* doliczy do ilości danych zmieniany jest sygnał *data_op* tak, aby zamiast zapisu następował odczyt z pamięci danych.

```
FPU dut(
    .clk(clk),
    .reset(reset),
    .in_instr_addr(instr_addr),
    .in_instr_data(current_external_instruction),

    .in_data_mem_op(data_op),
    .in_data_mem_addr(data_addr),
    .in_data_mem_data(current_external_data),

    .out_data_mem_data(data_mem_out_data),
    .program_ready(program_ready)
);
```

Rys 5.2.3 *Połączenie sygnałów soft procesora w środowisku testowym*

Zinstancjonowano obiekt testowy oraz połączono jego wejścia oraz wyjścia do opdowiednich rejestrów. Sygnały *in_instr_addr* oraz *in_instr_data* to kolejno rejestry zawierające adres aktualnie zapisywanej instrukcji oraz jej zawartość. W momencie gdy *reset* przyjmuje stan niski sygnały te są ignorowane przez moduł. Kolejne cztery sygnały odpowiadają za zapis oraz odczyt wartości z pamięci danych. Podczas stanu wysokiego sygnału *reset* dane są zapisywane, następnie *data_op* jest zmieniane tak, aby realizowany był odczyt liczb pojedynczej precyzji. Sygnał *program_ready* informuje użytkownika o zakończeniu działania programu oraz o gotowych wynikach w pamięci danych. Na rysuknu 5.2.4 ukazana została reakcja na zbocze narastające tego sygnału. Zerowany zostaje rejestr *data_addr*, a *data_op* ustawiane jest na odczyt liczb podwójnej precyzji. Dzięki temu po zakończeniu programu odczytywana będzie cała pamięć danych.

```
always @(posedge program_ready)
begin
    data_addr <= 8'b0;
    data_op <= 2'b10;
end
```

Rys 5.2.4 *Operacje podejmowane po zakończeniu programu*

Z tak przygotowanym środowiskiem testowym przystąpiono do pisania plików zawierających pamięć danych oraz pamięć instrukcji. W programie *MATLAB* napisano krótki skrypt konwertujący liczby zmiennoprzecinkowe pojedynczej oraz podwójnej precyzji do postaci binarnej. Liczby te następnie wpisano do pliku *data_memory.mc*, z którego korzysta środowisko testowe wewnątrz symulacji.

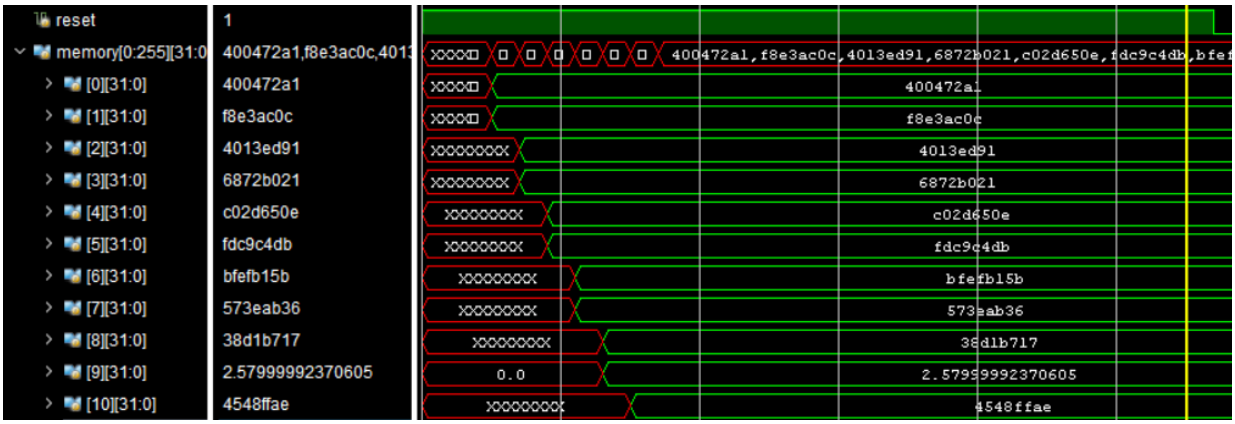
```
%% 64 bits
value = -5.02;
binVal = hexToBinaryVector(num2hex(double(value)), 64);
result = '';
for i=1:length(binVal)
    result = strcat(result, num2str(binVal(i)));
end
result
```

Rys 5.2.5 *Skrypt konwertujący liczby zmiennoprzecinkowe do postaci binarnej*

Do konwersji liczb pojedynczej precyzji napisano analogiczny skrypt, w którym zmienna *value* jest rzutowana na typ liczby pojedynczej precyzji za pomocą funkcji *single()* oraz drugi argument wbudowanej w *MATLAB* funkcji *hexToBinaryVector()* to szerokość 32 bitów. Po załadowaniu liczb do pliku, w przygotowanym wcześniej assemblerze utworzono jeden program testujący całą dostępną funkcjonalność soft procesora. Program został podzielony na dwie części, jedna poświęcona liczbom pojedynczej precyzji, a druga podwójnej. Na początku testowane są funkcje ładowania do rejestrów wartości przekazywanych bezpośrednio w instrukcji, ładowania z pamięci danych oraz przenoszenia wartości między rejestrami. Następnie wykonywanych jest 8 operacji arytmetycznych, a ich wyniki trafiają do kolejnych rejestrów. Ostatecznie wartość z wybranego rejestru zapisywana jest w pamięci danych. W przypadku liczb pojedynczej prezycji przetestowane zostało również zapisywanie w pamięci danych wartości przekazanej bezpośrednio w instrukcji. Po napisaniu programu, w środowiku *MATLAB* dokonano przetworzenia instrukcji do postaci ciągu bitów zapisanych w pliku tekstowym wykorzystywanym w symulacji. Następnie w środowisku *Vivado* zasymulowano całą funkcjonalność soft procesora, od ładowania danych, przez wykonywanie instrukcji, do odczytu pamięci danych.

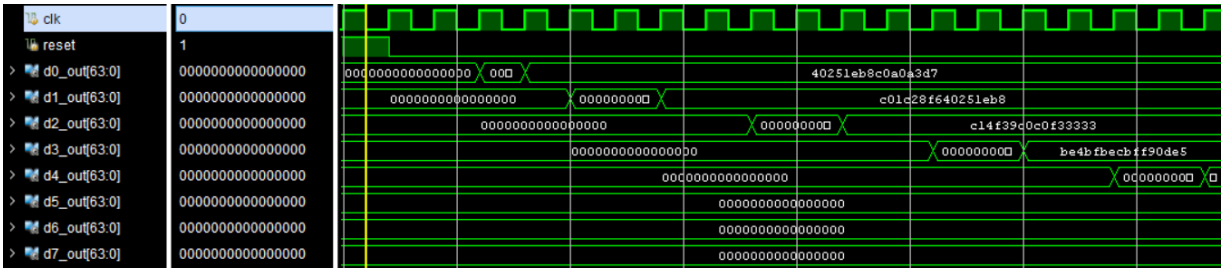
```
fmovfi s0, 0xc0a0a3d7; -5.02
fldf s1, 0x09; 2.58
fmovf s2, s1;

faddf s3, s0, s1;
fsubf s4, s0, s1;
fmulf s5, s0, s1;
fdivf s6, s0, s1;
frecf s7, s0;
fsqrtf s8, s1;
flogf s9, s0;
fexpf s10, s0;

fstfi 0x04, 0xc0a0a3d7; -5.02
fstf 0x05, s10;

fldd d0, 0x00; 2.555973
fldd d1, 0x02; 4.982
fmovd d2,d0;

faddd d3, d0, d1;
fsubd d4, d0, d1;
fmuld d5, d0, d1;
fdivd d6, d0, d1;
frecd d7, d0;
fsqrtd d2, d0;
flogd d3, d0;
fexpd d4, d0;

fstd 0x02, d4;

end;
```

Rys 5.2.6 *Program w jezyku assembler testujący całą funkcjonalność soft procesora*



Rys 5.2.7 *Ładowanie wartości do pamięci danych podczas sygnału reset*

Jak widać na rysunku 5.2.7 dane są ładowane do pamięci wewnętrznej co jeden cykl zegarowy biorąc pod uwagę opóźnienie początkowe ustalone w środowiku testowym. W pliku zawierającym dane pod adres 0x09h umieszczona została liczba pojedynczej precyzji o wartości 2.58, która ze względu na naturę liczb zmiennoprzecinkowych nie jest możliwa do zapisania w tym formacie, dlatego przyjęte zostało jej najlepsze przybliżenie. Pod adres 0x00h wpisano liczbę podwójnej precyzji w zapisie heksadecymalnym 0x400472a1f8e3ac0c, która wynosi 2.555973 w zapisie dziesiętnym. Ze względu na zbyt duży rozmiar pamięci instrukcji (2048 możliwych do zapisania instrukcji) nie dało się ich nanieść na wykres symulacji. Zliczono natomiast ilość narastających zboczy zegarowych do momentu zbocza opadającego sygnału *reset* oraz ilość instrukcji. W obu przypadkach liczba ta wynosiła 26, co potwierdza logikę działania testu, w której sygnał *reset* zostaje zwolniony wraz z ostatnią ładowaną instrukcją. W kolejnym kroku przeanalizowano stany wewnętrze wszystkich rejestrów wraz z kolejnymi taktami zegarowymi.



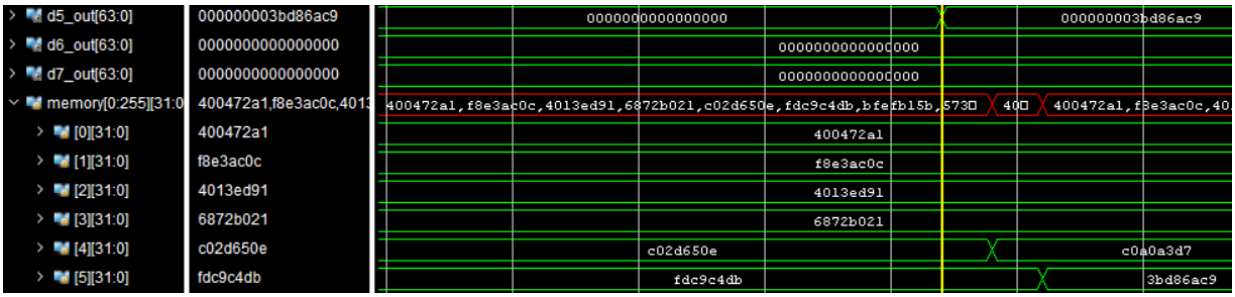
Rys 5.2.8 *Stany wewnętrzne rejestrów podczas symulacji cz. 1*

Po zwolnieniu sygnału *reset* mija jeden cykl zegarowy zanim soft procesor zaczyna interpretować pierwszą instrukcję, jaką jest zapis bezpośredniej wartości -5.02 do rejestru *s0*. Po kolejnym cyklu zegarowym dolna część rejestru *d0* zostaje zapełniona wartością 0xc0a0a3d7h, która w zapisie dziesiętnym wynosi właśnie -5.02. Kolejna instrukcja to załadowanie do rejestru *s1* liczby spod adresu 0x09h w pamięci. W poprzedniej części powiedziane było, że pod ten adres załadowana została liczba 2.58, która w zapisie heksadecymalnym wynosi 0x40251eb8h. Zgadza się to z zawartością rejestru *s1*. W kolejnej instrukcji liczba znajdująca się w rejestrze *s1* kopiowana jest do rejestru *s2* instrukcją *fmovf*. Odczytana wartość podczas symulacji pokrywała się z zawartością rejestru *s1*, co świadczy o poprawnej implementacji instrukcji. Kolejne polecenia to wszystkie operacje arytmetyczne na liczbach pojedynczej precyzji. Tabela 5.2.1 zawiera operacje arytmetyczne oraz zawartości końcowe rejestrów wykorzystywanych do testów. Dla przypomnienia *s0 = -5.02, s1 = 2.58.*

Tabela 5.2.1 *Wyniki działań przeprowadzonych na wartościach z rejestrów s0 i s1*

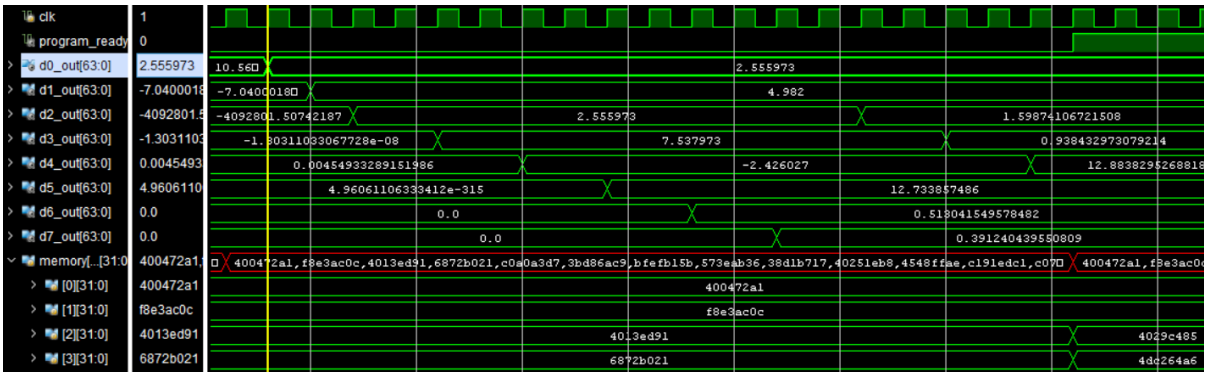
Rejestr	*s3*	*s4*	*s5*	*s6*	*s7*	*s8*	*s9*	*s10*
operacja	$s0+s1$	$s0-s1$	$s0 \cdot s1$	$s0/s1$	$1/s0$	$\sqrt{s1}$	$\ln(s1)$	$e^{(s0)}$
wynik	-2.44	-7.6	-12.9516	-1.9457	-0.1992	1.6062	0.9478	0.0066

Po przeanalizowaniu wyników oraz zrekonstruowaniu działań w środowisku *MATLAB* potwierdzono poprawność otrzymanych wyników obliczeń na liczbach pojedynczej precyzji. Po tych obliczeniach sprawdzono dwie instrukcje zapisu danych w pamięci współdzielonej między soft procesorem a zewnętrzną jednostką CPU.



Rys 5.2.9 *Stany wewnętrzne rejestrów podczas symulacji cz.2*

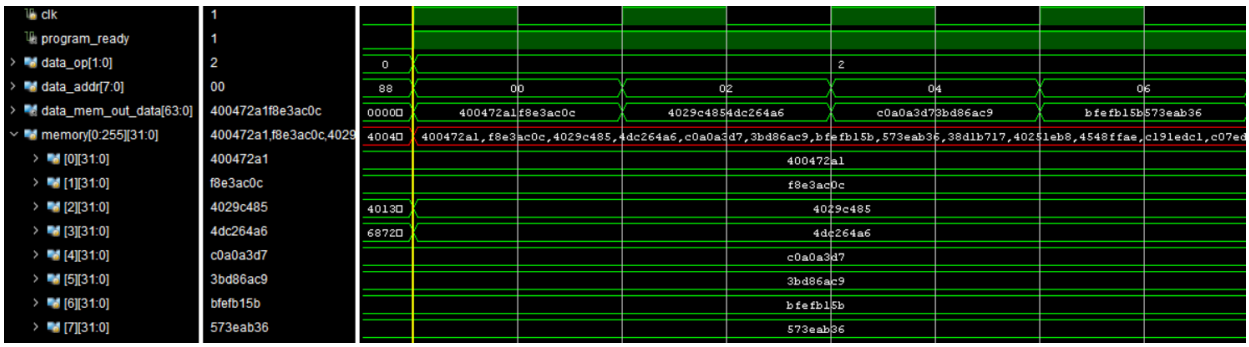
Z przegiegu symulacji pokazanej na rysnuku 5.2.9 wynika, że po operacji w której zostaje zmieniona zawartość rejestru *s10* zmieniona zostaje również dwukrotnie pamięć danych pod adresami 0x04h oraz 0x05h. Pierwsza zmiana to załadowanie bezpośrednio z instrukcji wartości 0xc0a0a3d7h, a druga to kopia rejestru *s10*. W dalszej części programu testowane były funkcjonalności związane z liczbami podwójnej precyzji.



Rys 5.2.10 *Stany wewnętrzne rejestrów podczas symulacji cz. 3*

Na początek do rejestrów *d0* i *d1* ładowane są liczby spod adresów kolejno 0x00h oraz 0x02h. Przed zmianą wyświetlania zawartości rejestrów na zapis decymalny liczby pokrywały się z zawartością pamięci. W kolejnej instrukcji przetestowana została funkcja *fmovd* która przekopiowała zawartość komórki *d0* do *d2*. Wartości pokrywały się. Następna zmiana nastąpiła w rejestrze *d3* po dwóch cyklach zegarowych. Jako że latencja wszytkich modułów *FPU IP Core* została ustawiona na 1 oraz zapis do rejestru trwa jeden takt zegarowy operacje te w sumie trwały dwa cykle zegarowe. W komórce *d3* pojawiła się wartość 7.537973, która odpowiada operacji dodawania pierwszych dwóch rejestrów. Następnie w *d4* pojawił się wynik operacji odejmowania, czyli -2.426027. Rejestr *d5* zapełniony został liczbą 12.733857486, która została otrzymana w skutek operacji mnożenia operandów znajdujących

się w rejestrach *d0* oraz *d1*. Po podzieleniu przez siebie tych wartości otrzymano wynik 0.513041549578482 znajdujący się w rejestrze *d6*. Wartość ta dokładnie opdowiada dzieleniu wykonanym w środowisku *MATLAB*. Do komórki *d7* trafiła odwrotność wartości z *d0*, czyli 0.391240439550809. Pierwiastek kwadroatowy z liczby 2.555793 został następnie umieszczony w rejestrze *d2*, a wyniósł on dokładnie 1.59874106721508. Ostatnimi operacjami arytmetycznymi przeprowadzonymi na liczbach podwójnej precyzji to logarytm oraz eksponenta wartości z rejestru *d0*. Wyniki trafiły do komórek *d3* oraz *d4* i wynosiły one odpowiednio 0.938432973079214 oraz 12.8838295268818. Wszystkie wyniki zgadzały się z rezultatami otrzymanymi w wyniku rekonstrukcji operacji w środowisku *MATLAB*. Ostateczną przetestowaną funkcjonalnością było zapisanie wyniku eksponenty w pamięci pod adres 0x02h. Jak widać na rysunku 5.3.10 komórki 0x02h oraz 0x03h o szerokości 32 bitów zostały zapełnione wartością 0x4029c4854dc264a6, która odpowiada zawartości rejestru *d4*. Po zmianie wyświetlania wartości z rejestru *d4* na zapis heksadecymalny wartości pokrywały się. Operacja zapisu była ostatnią operacją w programie, dlatego sygnał *program_ready* ukazany na ww. rysunku zmienia się. W środowisku testowym reakcja na ten sygnał powodowała zerowanie rejestru *data_addr* inkrementowanego na każde narastające zbocze zegarowe oraz zmianę sygnału *data_op* na 0x02h, co oznacza odczyt z pamięci liczb podwójnej precyzji.



Rys 5.2.11 *Stany wewnętrzne rejestrów podczas symulacji cz. 4*

Na rysunku 5.2.11 pokazana została zmiana sygnałów *data_op* oraz *data_addr* spowodowana zboczem narastającym sygnału *program_ready*. Jak widać w kolejnych cyklach zegara na wyjściu *data_mem_out_data* pojawiają się wartości z pamięci danych *memory*, których adres odpowiada liczbie zapisanej w rejestrze *data_addr*.

5.3. Testy rzeczywiste

Do walidacji poprawności działania modułu wykorzystany został układ *Picozed 7010 FMC Carrier Card V2*. Aby wgrać program do układu FPGA w środowisku *Vivado* utworzono nowy projekt, do którego dodano blok *processing_system* niezbędny do działania

układu i porozumiewania się z własnymi modułami. Do projektu dodano nowy *IP Core* komunikujący się z resztą po magistrali *AXI*. W tym *IP Core* zinstancjonowano soft procesor oraz przypięto rejestry i ścieżki (*ang. wire*) do odpowiednich wejść oraz wyjść. Na etapie implementacji okazało się, że moduł przekracza dostępną ilość zasobów układu. Wykorzystywał on 266% dostępnych tablic LUT oraz dwukrotnie przekroczył ilość mnożarek DSP. Okazało się, że główną przyczyną jest pamięć danych, zatem zredukowano ją do 128 bajtów. Aby lepiej wykorzystywać dostępne zasoby należałoby zorganizować pamięć danych i instrukcji jako blokowa pamięć ram (*Block RAM*). Pamieć instrukcji również zmniejszono tak, aby mieściła 256 instrukcji. Po tych zmianach wykorzystanie LUT nadal było zbyt duże. Sprawdzono, że *FPU IP Core* wykonujący eksponentę na liczbach podwójnej precyzji wykorzystywał ponad 6 tysięcy LUT przy dostępnych 17 600. Zrezygnowanie z tej operacji oraz przeznaczenie mniejszych ilości DSP w poszczególnych modułach *FPU IP Core* (głównie module logarytmującym liczby podwójnej precyzji) implementacja programu przeszła pomyślnie. Z programu wygenerowany został *bitstream*, a następnie projekt został wyeksportowany do *Xilinx Software Development Kit.* Na podstawie wyeksportowanego projektu zaimplementowane zostało środowisko testowe napisane w języku C testujące funkcjonalność soft procesora. Na początku wykorzystano makra preprocesora w taki sposób, aby nadać odpowiednim adresom w pamięci czytelne nazwy odpowiadające wejściom i wyjściom soft procesora (*clk, reset itd.*). W assemblerze zbudowano program testujący funkcjonalność każdej instrukcji, bardzo podobny do programu z testów symulacyjnych. Aby sprawdzić poprawność wyników operacji arytmetycznych zostały one wszystkie zapisane w pamięci danych. Zawartość tej pamięci została porównana z odpowiednimi rejestrami z testów symulacyjnych.

```
reset = 0x01;
in_data_mem_op = 0b11;

sprintf(tab, "reset = 0x%02x, data_mem_op = 0x%02x\n\r", reset, in_data_mem_op);
print(tab);
```

Rys 5.3.1 *Środowisko testowe cz. 1*

Na rysunku 5.3.1 pokazano ustawienie sygnału *reset* oraz określenie operacji wykonywanej na pamięci danych (zapis liczb podwójnej precyzji). Następne dwie linie służyły do walidacji poprawności zapisu liczb do poprzednich sygnałów. Analogiczna walidacja została wykorzystywana w późniejszych częściach programu.

```
print("Loading data and instructions\n\r");
for(uint32_t i = 0; i < instruction_data_size; i++)
{
    clk = 0;
    in_instr_addr = i;
    in_instr_data_high = instruction_data[i] >> 32;
    in_instr_data_low = instruction_data[i];

    sprintf(tab, "\n\rinstruction 0x%02x : 0x%08x%08x",
             in_instr_addr, in_instr_data_high, in_instr_data_low);

    print(tab);

    if(i < memory_data_size)
    {
       in_data_mem_addr = 2 * i;
       in_data_mem_data_high = memory_data[i] >> 32;
       in_data_mem_data_low = memory_data[i];

       sprintf(tab, ", data 0x%02x : 0x%08x%08x", in_data_mem_addr,
                    in_data_mem_data_high, in_data_mem_data_low);

       print(tab);

    }
    clk = 1;
}
```

***Rys 5.3.2** Środowisko testowe cz. 2*

Na rysunku 5.3.2 ukazany został proces ładowania danych oraz instrukcji do odpowiednich pamięci. Należy wspomnieć, że rejestry w wygenerowany module *IP Core* do komunikacji po magistrali *AXI* posiada wewnętrze rejestry *slv_reg*, których szerokość została ustawiona na 32 bity. Ze względu na to konieczne było przypisanie dwóch takich rejestrów do sygnału *in_instr_data*, zawierającego instrukcje oraz do sygnału *in_data_mem_data*, gdyż są one szerokie na 64 bity. W środowisku testowym pod odpowiednie adresy zostały wpisane górne oraz dolne części instrukcji i danych. Po każdym zapełnieniu rejestrów następowało narastające zbocze sygnału *clk* odpowiadające za sygnał zegarowy soft procesora.

```
print("\n\rdata memory before program\n\r");
read_data_memory();

reset = 0x00;

print("Waiting for program to finish\n\r");
uint32_t clocks = 0;
while(program_ready == 0)
{
	clk = 0;
	print("0");
	clk = 1;
	print("1");
	clocks++;
}
sprintf(tab, "\n\rProgram finished in %d cycles!\n\r", clocks);
print(tab);

print("\n\rdata memory after program\n\r");
read_data_memory();
```

Rys 5.3.3 *Środowisko testowe cz. 3*

Przed wykonaniem programu upewniono się, że dane zostały poprawnie załadowane do pamięci za pomocą funkcji *read_data_memory()*. Na tym etapie nie sprawdzono, czy instrukcje zostały załadowane, gdyż interfejs soft procesora na to nie pozwala. Następnie zwolniono sygnał *reset*. W pętli *while* wytwarzano sygnał zegarowy, aby program się wykonywał. Flagą przerywającą czekanie na koniec programu był sygnał *program_ready*. Podczas czekania na wyniki zliczano ilość taktów zegarowych potrzebnych do wykonania programu. Ostatecznym krokiem był ponowny odczyt pamięci danych, aby dowiedzieć się, czy operacje zostały poprawnie przeprowadzone a wyniki poprawnie zapisane.

Z tak przygotowanym środowiskiem testowym za pomocą *Xilinx SDK* wygenerowano plik BOOT.bin, zawierający *FSBL (ang. First Stage Bootloader)*, bitstream wygenerowany w *Vivado* oraz środowisko testowe omawiane w powyższej części rozdziału. Plik ten następnie skopiowano na kartę microSD, którą włożono do układu FPGA. Ustawiono odpowiednie przełączniki tak, aby program ładowany był z karty microSD. Za pomocą przewodu USB-microUSB połączono układ z komputerem, na którym zainstalowane zostały odpowiednie sterowniki do komunikacji po magistrali szeregowej z układem *picozed*. Za pomocą programu *PuTTY* nawiązano połączenie oraz włączono układ.

```
Loading data and instructions

instruction 0x00 : 0x0008001fc0a0a3d7, data 0x00 : 0x400472a1f8e3ac0c
instruction 0x01 : 0x049a001e00000000, data 0x02 : 0x4013ed916872b021
instruction 0x02 : 0x0220801e00000000, data 0x04 : 0xc02d650efdc9c4db
instruction 0x03 : 0x000a900000000000, data 0x06 : 0xbfefb15b573eab36
instruction 0x04 : 0x02a0805e00000000, data 0x08 : 0x38d1b71740251eb8
instruction 0x05 : 0x0008a00200000000, data 0x0a : 0x4548ffaec191edc1
instruction 0x06 : 0x0320009e00000000, data 0x0c : 0xc07ed917ba83126f
instruction 0x07 : 0x000aa00400000000
instruction 0x08 : 0x03a0809e00000000
instruction 0x09 : 0x0008b00600000000
instruction 0x0a : 0x042000de00000000
instruction 0x0b : 0x000a300800000000
instruction 0x0c : 0x04a080de00000000
instruction 0x0d : 0x0009400a00000000
instruction 0x0e : 0x0520011e00000000
instruction 0x0f : 0x000b400c00000000
instruction 0x10 : 0x05a0811e00000000
instruction 0x11 : 0x0008500e00000000
instruction 0x12 : 0x0620015e00000000
instruction 0x13 : 0x005c003e00000000
instruction 0x14 : 0x015c103e00000000
instruction 0x15 : 0x000c203e00000000
instruction 0x16 : 0x086400be00000000
instruction 0x17 : 0x000c306000000000
instruction 0x18 : 0x096400fe00000000
instruction 0x19 : 0x000c406200000000
instruction 0x1a : 0x0a64013e00000000
instruction 0x1b : 0x000c506400000000
instruction 0x1c : 0x0b64017e00000000
instruction 0x1d : 0x000c606600000000
instruction 0x1e : 0x0c6401be00000000
instruction 0x1f : 0x000c702800000000
instruction 0x20 : 0x0d6401fe00000000
instruction 0x21 : 0x000c202a00000000
instruction 0x22 : 0x0e6400be00000000
instruction 0x23 : 0x000c302c00000000
instruction 0x24 : 0x0f6400fe00000000
instruction 0x25 : 0x8000003e00000000
```

Rys 5.3.4 *Instrukcje oraz dane załadowane do soft procesora*

Na rysunku 5.3.4 pokazane zostały instrukcje oraz dane w postaci heksadecymanlej, które zostały załadowane do soft procesora. Wartości instrukcji zgadzały się z zawartością pliku wygenerowanego przez kompilator a dane zgadzały się z zawartością pliku, do którego dane zostały ręcznie wpisane.

```
data memory before program
Reading data memory, data_mem_op = 0x02
Addr 0x00 : 0x400472a1f8e3ac0c,
Addr 0x02 : 0x4013ed916872b021,
Addr 0x04 : 0xc02d650efdc9c4db,
Addr 0x06 : 0xbfefb15b573eab36,
Addr 0x08 : 0x38d1b71740251eb8,
Addr 0x0a : 0x4548ffaec191edc1,
Addr 0x0c : 0xc07ed917ba83126f,
Addr 0x0e : 0x0000000000000000,
Addr 0x10 : 0x0000000000000000,
Addr 0x12 : 0x0000000000000000,
Addr 0x14 : 0x0000000000000000,
Addr 0x16 : 0x0000000000000000,
Addr 0x18 : 0x0000000000000000,
Addr 0x1a : 0x0000000000000000,
Addr 0x1c : 0x0000000000000000,
Addr 0x1e : 0x0000000000000000,
Waiting for program to finish
01010101010101010101010101010101010101010101010101010101
Program finished in 68 cycles!
```

Rys 5.3.5 *Zawartość pamięci danych przed wykonaniem programu*

Rysunek 5.3.5 pokazuje zawartość pamięci danych przed wykonaniem programu. Odczytywane zostają dwie 32 bitowe komórki na raz, o czym decyduje sygnał *data_mem_op*. Z tego powodu adres pamięci zwiększany jest o 2 z każdym odczytem. Po odczytaniu zawartości pamięci wykonywał się program, który trwał przez 68 wygenerowanych cykli zegarowych.

```
data memory after program
Reading data memory, data_mem_op = 0x02
Addr 0x00 : 0x400472a1f8e3ac0c,
Addr 0x02 : 0x4013ed916872b021,
Addr 0x04 : 0x40251eb8c01c28f6,
Addr 0x06 : 0xc0f33333c14f39c0,
Addr 0x08 : 0xbff90de5be4bfbec,
Addr 0x0a : 0x3fcd99333f72a252,
Addr 0x0c : 0x3bd86ac9ba83126f,
Addr 0x0e : 0x0000000000000000,
Addr 0x10 : 0x400472a1f8e3ac0c,
Addr 0x12 : 0x401e26e264e48627,
Addr 0x14 : 0xc0036880d801b436,
Addr 0x16 : 0x402977bc2b1c96ed,
Addr 0x18 : 0x3fe06ad61c9db847,
Addr 0x1a : 0x3fd90a15572f93b3,
Addr 0x1c : 0x3ff9947183675fcd,
Addr 0x1e : 0x3fee07a4961b9feb,
```

Rys 5.3.6 *Zawartość pamięci danych po wykonaniu programu*

Wyniki odczytane z pamięci danych po wykonaniu programu porównano z wynikami otrzymanymi symulacyjnie na tych samych danych.

5.4. Wyniki

Biorąc pod uwagę charakter projektu wyniki otrzymane w skutek testowania poprawności działania soft procesora mogą być jedynie poprawne lub nie. Rezultaty które znalazły się w rejestrach bądź pamięci danych po wykonaniu programu były prawidłowe, co potwierdziła rekonstrukcja operacji arytmetycznych w środowiku *MATLAB*. Wszystkie wartości już na poziomie symulacyjnym zostały przekonwertowane na postać decymalną w celu ułatwienia walidacji.

6. Przykład zastosowania

W dziedzinie automatyki istotną rolę odgrywają regulatory, z tego powodu zdecydowano zaimplementować regulator PID wykorzystujący soft procesor do obliczeń zmiennoprzecinkowych. Jest on jedynie przykładem wykorzystania modułu. Do jego budowy wykorzystano wzór (6.1).

$$u[k]=P\cdot e[k]+I\cdot\sum_{i=0}^{N} e[k-i]+D\cdot(e[k]-e[k-1]) \tag{6.1}$$

Oznaczenia do wzoru 6.1:

- u - wyliczone sterowanie,
- k - krok dyskretyzacji,
- P - współczynnik P regulatora,
- I - współczynnik I regulatora,
- D - współczynnik D regulatora,
- N - ilość zapamiętanych przeszłych pomiarów uchybu ustalonego.

Algorytm postanowiono zrealizować z wykorzystaniem liczb pojedynczej precyzji. Do przechowywania przeszłych pomiarów uchybu ustalonego użyto wewnętrznej pamięci danych soft procesora. Jako że wyjście regulatora to jedna wartość, która ostatecznie musi być zapisana w pamięci danych, uznano że będzie ona zapisywana pod adresem 0x00h. W przygotowanym języku assemblera zaimplementowano algorytm.

```
1  ; s11 = e[k], s12 = e[k - 1], s13 = P, s14 = I, s15 = D
2  fmovfi s11, 0xf8ffffff
3  fmovfi s12, 0xf8ffffff
4  fmovfi s13, 0xf8ffffff
5  fmovfi s14, 0xf8ffffff
6  fmovfi s15, 0xf8ffffff
7  ; store P component and store in s13
8
9  fmulf s0, s13, s11; s0 = Proportional part
```

Rys 6.1 *Implementacja regulatora PID w assemblerze cz. 1*

Na początku do rejestrów załadowane zostały potrzebne parametry – uchyb ustalony, poprzedni uchyb ustalony oraz współczynniki PID. Początkowe pięć instrukcji jest zmieniane z poziomu projektu w środowisku *Xilinx SDK*, o czym mowa w późniejszej części. W linii dziewiątej wyznaczona zostaje część proporcjonalna regulatora.

```
10 ; get I component ans store in s1
11 fldf s1, 0x01; load first value
12 fldf s2, 0x02; load second value
13 faddf s1, s1, s2; accumulate
14 fldf s2, 0x03;
15 faddf s1, s1, s2;
16 fldf s2, 0x04;
17 faddf s1, s1, s2;
18 fldf s2, 0x05;
```

Rys 6.2 *Implementacja regulatora PID w assemblerze cz. 2*

W dalszej części obliczana jest suma przeszłych pomiarów uchybu ustalonego potrzebna do wyliczenia części regulatora odpowiadającej całkowaniu. Na rysunku 6.2 pokazany został jedynie fragment sumy, reszta obliczana jest analogicznie aż do wartości znajdującej się w pamięci pod adresem 0x1Fh.

```
75 fmulf s2, s14, s1; s2 = I * s1
76 ; get D component and store in s4
77
78 fsubf s3, s11, s12;
79 fmulf s4, s15, s3;
80 ;compute u
81
82 faddf s5, s0, s2;
83 faddf s6, s5, s4;
84 fstf 0x00, s6;
```

Rys 6.3 *Implementacja regulatora PID w assemblerze cz. 3*

Po wykonanej sumie obliczono człon całki regulatora. Następnie na podstawie aktualnego uchybu ustalonego oraz jego poprzedniej wartości obliczana jest część różniczkująca regulatora. Ostateczną częścią było zsumowanie wszystkich komponentów składających się na wynikowe sterowanie. Rezultat został zapisany pod adresem 0x00h. Gotowy program skompilowano do instrukcji w postaci ciągów binarnych.

W środowisku *Xilinx SDK* utworzono nowy projekt skierowany na platformę sprzętową zawierającą moduł soft procesora. Dyskretny regulator zrealizowano w postaci funkcji w języku C zwracającej zmienną 32 bitową oraz przyjmującej również zminną 32 bitową jako parametr. Do wcześniej utworzonej tablicy wpisano instrukcje otrzymane w poprzedniej części.

```
uint32_t PID(uint32_t e)
{
    reset = 0x01;
    clk = 1;
    clk = 0;

    // zapis uchybu ustalonego w wewnętrznej pamięci danych
    static uint32_t address = 1;
    static uint32_t e_prev = e;

    in_data_mem_addr = address;
    in_data_mem_op = 0b01; // zapis liczb pojedynczej precyzji
    in_data_mem_data_low = e;

    clk = 1;
    clk = 0;

    address = (address % 31) + 1; // zakres (01 - 1F);

    in_data_mem_op = 0b11; // zapis instrukcji

    const uint32_t P = 0x3f800000; // 1.0
    const uint32_t I = 0x3f800000; // 1.0
    const uint32_t D = 0;
```

Rys 6.4 *Implementacja funkcji wykonującej algorytm regulatora PID cz. 1*

Funkcja realizująca program regulatora w układzie FPGA wygląda następująco. Ustawiany jest sygnał *reset*. Następnie pod adres oznaczony zmienną *address* wpisywany jest uchyb ustalony. Zmienna ta jest inkrementowana z każdym wywołaniem funkcji oraz dzielona modulo 31 tak, aby jej zakres wynosił od 0x01h do 0x1Fh. W dalszej części ustalane są parametry P, I oraz D regulatora.

```
// podstawienie parametrów do instukcji
pid_instruction_data[0] = (pid_instruction_data[0] & 0xFFFFFFFF00000000) | e;
pid_instruction_data[1] = (pid_instruction_data[1] & 0xFFFFFFFF00000000) | e_prev;
pid_instruction_data[2] = (pid_instruction_data[2] & 0xFFFFFFFF00000000) | P;
pid_instruction_data[3] = (pid_instruction_data[3] & 0xFFFFFFFF00000000) | I;
pid_instruction_data[4] = (pid_instruction_data[4] & 0xFFFFFFFF00000000) | D;

// informacja o poprzednim uchybie ustalonym do części różniczkującej
e_prev = e;

for(uint32_t i = 0; i < pid_instruction_data_size; i++)
{
    clk = 0;
    in_instr_addr = i;
    in_instr_data_high = pid_instruction_data[i] >> 32;
    in_instr_data_low = pid_instruction_data[i];
    clk = 1;
}
clk = 0;
```

Rys 6.5 *Implementacja funkcji wykonującej algorytm regulatora PID cz. 2*

W dalszej części realizowana jest zmiana instrukcji assemblera w taki sposób, aby do rejestrów trafiły odpowiednie wartości. Następnie wszystkie instrukcje wygenerowane za pomocą kompilatora ładowane są do pamięci instrukcji soft procesora.

```
    reset = 0x00;

    uint32_t clocks = 0;
    while(program_ready == 0)
    {
        clk = 0;
        clk = 1;
        clocks++;
    }
    sprintf(tab, "\n\rPID finished in %d cycles!\n\r", clocks);
    print(tab);

    clk = 0;
    in_data_mem_op = 0x00; // odczyt sterowania spod adresu 0x00
    in_data_mem_addr = 0x00;
    clk = 1;
    uint32_t u = out_data_mem_data_low;

    return u;
}
```

Rys 6.6 *Implementacja funkcji wykonującej algorytm regulatora PID cz. 3*

Po zwolnieniu sygnału reset funkcja czeka w pętli *while* na wynik programu stanowiąc sygnał zegarowy soft procesora. Po wykonaniu algorytmu regulatora, sterowanie znajdujące się pod adresem 0x00h w pamięci danych jest zwracane z funkcji. Zaimplementowany regulator, którego część całkująca wykorzystuje 31 przeszłych pomiarów uchybu ustalonego wylicza sterowanie w 147 cyklów zegarowych.

Algorytm regulatora PID to jeden z wielu algorytmów możliwych do zaimplementowania na soft procesorze. Udostępnia on funkcjonalność dodawania, odejmowania, mnożenia, dzielenia, odwrotności, pierwiastka kwadratowego, logarytmu naturalnego oraz eksponenty. Każdy algorytm, którego operacje zawierają się w tym zbiorze da się zaimplementować na zbudowanym module w precyzji zmiennoprzecinkowej.

7. Podsumowanie

Tematem projektu dyplomowego była implementacja modułu wspierającego obliczenia zmiennoprzecinkowe w układzie rekonfigurowalnym FPGA. Moduł zaprojektowano w programie *Vivado 2017.4* za pomocą języka opisu sprzętu *verilog*. Zaczęto od schematu blokowego całego modułu zawierającego plik rejestrów, pamięci danych oraz instrukcji, jednostki arytmetycznej, jednostki sterującej oraz multiplekserów. Zaimplementowano każdy moduł z osobna, aby następnie zinstancjonować je oraz połączyć i jednym głównym module, stanowiącym soft procesor. Na każdym kroku dokładnie sprawdzano funkcjonalność każdej części z osobna, by jak najwcześniej wykryć wszelkie nieprawidłowości związane z błędnym działaniem komponentów. Po skompletowaniu wszystkich modułów w jedną całość napisano symulacyjne środowisko testowe, w którym przetestowano całą funkcjonalność soft procesora, od instrukcji ładowania danych z pamięci, przez wszystkie oracje arytmetyczne do zapisu danych z powrotem w pamięci. Wszystko to na liczbach zarówno pojedynczej jak i podwójnej precyzji. Następnym krokiem było napisanie prostego kompilatora, dzięki któremu można było pisać instrukcje w postaci bardziej przyjaznej dla człowieka. Kompilator napisany w środowisku *MATLAB* zamieniał instrukcje w postaci podobnej do assemblera na ciągi binarne, co umożliwiło w miarę szybkie pisanie programów. Plik tekstowy zawierający instrukcje został wykorzystany w symulacyjnym środowisku testowym, co potwierdziło poprawne działanie kompilatora. Wszystkie otrzymane wyniki operacji arytmetycznych zostały na tym etapie zweryfikowane poprzez pomocnicze środowsko testowe konwertujące liczby z zapisu heksadecymalnego na decymalny oraz na odwrót. Po przetestowaniu symulacyjnym soft procesora oraz kompilatora przystąpiono do wgrania programu na układ rekonfigurowalny. Wybrano układ *Picozed 7010 FMC Carrier Card V2*. W środowisku *Vivado* na etapie implementacji programu okazało się, że program przekracza dostępną ilość zasobów. Wykorzystywał on zbyt wiele tablic LUT oraz mnożarek DSP. Po zmniejszemiu pamięci danych, ograniczeniu mnożarek DSP dla kilku operacji oraz zrezygnowanie z operacji eksponenty na liczbach podwójnej precyzji program zmieścił się w dostępnm układzie FPGA. Obostrzenia te nie byłby by konieczne dla innych układów z rodziny *picozed 7000*, np. Dla *picozed 7020*, który zawiera dwa razy więcej mnożarek DSP oraz ponad 3 razy więcej tablic LUT [10]. Korzystną optymalizacją byłoby wykorzystanie pamięci dwuportowej *Block RAM* dostępnej w układzie FPGA, dzięki temu rezygnacja z operacji eksponenty nie byłaby potrzebna, a pamięć danych oraz instrukcji wzrosła by diametralnie. Po wykonanej implementacji przystąpiono do utworzenia kolejnego projektu w *Vivado*, zawierającego system *picozed 7010.* Do tego projektu wytworzono *IP Core* komunikujący się z tym systemem po magistrali *AXI*. W tym *IP Core* zinstancjonowano soft procesor oraz podpięto jego wejścia oraz wyjścia do konkretnych 32 bitowych rejestrów *slv_reg,* dzięki którym umożliwiona została jego obsługa. Z tego projektu następnie

wygenerowano bitstream oraz wyeksportowano go, aby na jego podstawie utworzyć środowisko testowe w programie *Xilinx SDK* w języku C. W tym środowisku testowym zadeklarowane oraz zdefiniowane zostały tablice zawierające pamięć danych oraz pamięć instrukcji. Do tej drugiej wykorzystano zaimlementowany wcześniej kompilator. Napisano procedurę zapełnienia obu pamięci wewnątrz soft procesora. Nasępnie w pętli czekano na wyniki programu generując sygnał zegarowy. Po wykonanym programie odczytano pamięć danych, aby sprawdzić, czy zgadzają się z wynikami symulacji zweryfikowanymi poprzednio. Program został skompilowany oraz wygenerowany został plik BOOT.bin konieczny do załadowania programu do układu FPGA. Plik ten został przeniesiony na kartę microSD, którą włożono do wejścia w układzie. Komputer z zainstalowanym programem do obsługi portu szeregowego połączono kablem USB do portu microUSB znajdującego się w układzie FPGA. Po otworzeniu portu szeregowego podłączono zasilanie do układu rekonfigurowalnego oraz ustawiono bootowanie z poziomu karty microSD. W programie umieszczono kod odpowiedzialny za wysyłanie pożądanych informacji na port szeregowy, dzięki czemu możliwa była weryfikacja działania programu. Wyniki okazały się być poprawne, zarówno te otrzymane w skutek operacji na liczbach pojedynczej jak i podwójnej precyzji. Odczyt poprawnych danych na poziomie programu w C świadczył o tym, że pozostałe instrukcje (odczyt oraz zapis w pamięci danych soft procesora) również wykonały się poprawnie.

Projekt soft procesora ma wiele możliwości przyszłego rozwoju. Koniecznością jest reorganizacja pamięci danych oraz pamięci instrukcji. Zamiast przechowywać dane w wewnętrznych rejestrach opartych fizycznie o tablice LUT, należałoby zaimplementować je w formie pamięci *Block RAM.* Dzięki temu przywrócona zostałaby operacja eksponenty na liczbach podwójnej precyzji oraz powiększone by były pamięci danych oraz instrukcji. Możliwa jest również optymalizacja pod kątem długości instrukcji – ciąg binarny po odpowiednich operacjach mógłby ulec skróceniu. Przeniesienie soft procesora na bardziej pojemny układ umożliwiłoby dodanie większej ilości operacji arytmetycznych dostępnych w module *FPU IP Core*, takich jak wartość bezwzględna czy odwrotny pierwiastek kwadratowy [11].

Bibliografia

1) https://www.analog.com/en/education/education-library/articles/fixed-point-vs-floating-point-dsp.html#
2) Mateusz Komorkiewicz, Tomasz Kryjak. Systemy Rekonfigurowalne, skrypt do ćwiczeń laboratoryjnych. Published by AGH, 2014.
3) https://www.asawicki.info/Download/Productions/Lectures/Adam%20Sawicki%20-%20Pulapki%20liczb%20zmiennoprzecinkowych.pdf
4) https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/bg-floating-point-fpga.pdf
5) https://pl.wikipedia.org/wiki/IEEE_754
6) https://pl.wikipedia.org/wiki/Liczba_zmiennoprzecinkowa
7) https://www.embedded.com/implementing-floating-point-algorithms-in-fpgas-or-asics/
8) https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/po/bg-floating-point-fpga.pdf
9) https://www.sciencedirect.com/topics/engineering/register-bank
10) http://linuxgizmos.com/linux-ready-modules-support-range-of-xilinx-fpgas/
11) https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf