Architektura Komputerów 2

Liczby zdenormalizowane

czwartek nieparzysty, 18:55

Michał Sieroń
256 259

Paweł Różański
252 772

prowadzący
dr inż. Piotr Patronik

Informatyka Techniczna, Wydział Elektroniki

11 czerwca 2021

Spis treści

1 Wstęp

Zadanie projektowe polegało na analizie zawartości artykułu i implementacji przedstawionych układów w języku *Verilog*. Z powodu ograniczonego czasu na wykonanie projektu możliwe było zaimplemenetowanie jedynie układów *A1* oraz *M* przedstawionych w artykule [2].

Następnie zaimplementowano odpowiadajace im układy zgodne ze standardem *IEEE-754* [3] [1]. Dokonano również porównania błędów wynikajacych z użycia danej reprezentacji liczby zmiennoprzecinkowej.

2 Opis koncepcji zapisu i formatu

Koncepcja formatu liczb zmiennoporzecinkowych, zaproponowanego w artykule [2] wzięła się z obserwacji, że standard *IEEE-754* nie został stworzony z myślą o systemach wbudowanych. Eliminacja logiki normalizującej powinna obniżyć koszt produkcji układu, a wpływ na precyzję obliczeń nie powinien mieć znaczenia w docelowych zastosowaniach. Normalizacja liczby jest sktukiem używania ukrytej jedynki w liczbach znormalizowanych. Sprawia to, że pojawia się wyjątek, który trzeba obsłużyć w sprzęcie. Proponowany format pozbywa się ukrytej jedynki, kosztem jednego z bitów mantysy. Konsekwencją tego jest zmniejszona precyzja liczb.

3 Opis metody

Wszystkie układy zostały zaimplementowane w języku opisu sprzętu *Verilog*. Jednak druga część projektu, która je ze sobą łączy i porównuje z wartością referencyjną, została napisana w języku *Python*. Dla zadanej ilości przypadków testowych generowaliśmy tyle samo par liczb typu `float`. W języku *Python*, typ `float` odpowiada liczbie zmiennoprzecinkowej o podwójnej precyzji. Wobec tego konieczne było przekonwertowanie wygenerowanych liczb na liczbę zmiennoprzecinkową o pojedynczej precyzji. W tym celu napisaliśmy funkcję `py2float` w języku *C*, która zamienia wartość typu `double` na `float`. Tak otrzymane wartości były następnie zamieniane na ich szesnastkową reprezentację. W tym celu musieliśmy uprzednio otrzymać bajtową reprezentację danej liczby, która z kolei była zamieniana na liczbę typu `int`, z której w końcu mogliśmy otrzymać reprezentację w systemie szesnastkowym.

Konwersję z liczby znormalizowanej na zdenormalizowaną zaimplementowaliśmy w tym samym skrypcie. W ten sam sposób co wcześniej, otrzymywaliśmy zapis szesnastkowy liczby lecz denormalizacja liczby wymaga operacji bitowych. Przez fakt użycia *Pythona* konieczne było do tego zamienienie liczby na listę bitów (w tym przypadku liczb całkowitych typu `int` o wartościach 0 lub 1). Tak otrzymana lista była następnie odpowiednio dzielona na części znaku, wykładnika i mantysy. Mantysa była przesuwana o jedną pozycję w prawo, a wykładnik zwiększany o jeden. Tak zmodyfikowaną reprezentację bitową zamienialiśmy z powrotem na zapis szesnastkowy.

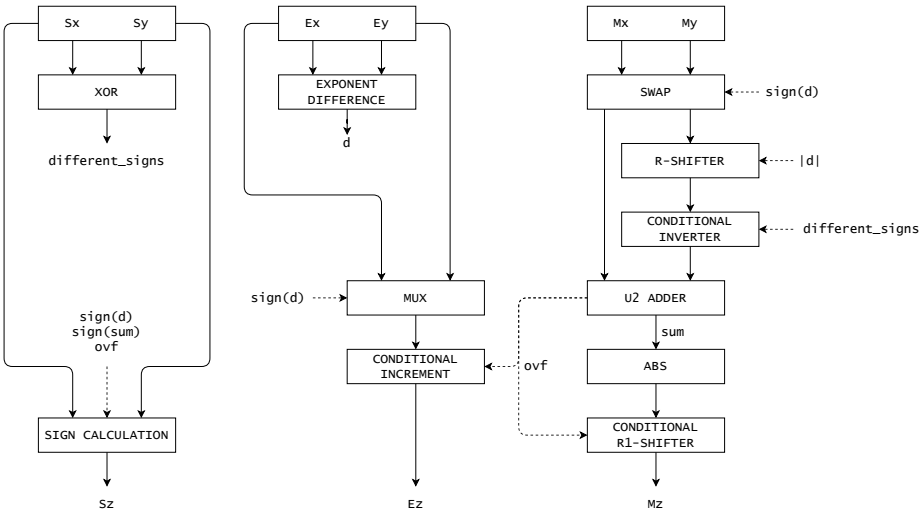
Wykorzystując wygenerowane pary liczb tworzyliśmy pliki *Veriloga* wykorzystujące napisane przez nas moduły opisane w sekcji 4. Utworzone pliki były następnie uruchamiane, a wyniki zapisywane w pliku *csv* do dalszego przetwarzania.

4 Opis implementacji

4.1 Układy dodawania

Implementacja układu dodawania liczb zmiennoprzecinkowych została wykonana na podstawie opisu oraz modelu układu z artykułu [2]. Według nazewnictwa z artykułu układ dodawania, który odwzorowaliśmy, to *A1*. Jest to najprostsza wersja dodawania dwóch liczb zdenormalizowanych.

Implementację układu zaczęliśmy od dokładnego przeczytania opisu przedstawionego w artykule, a następnie stworzeniu podstawowych bloków do obliczeń zawartych na diagramie 1.

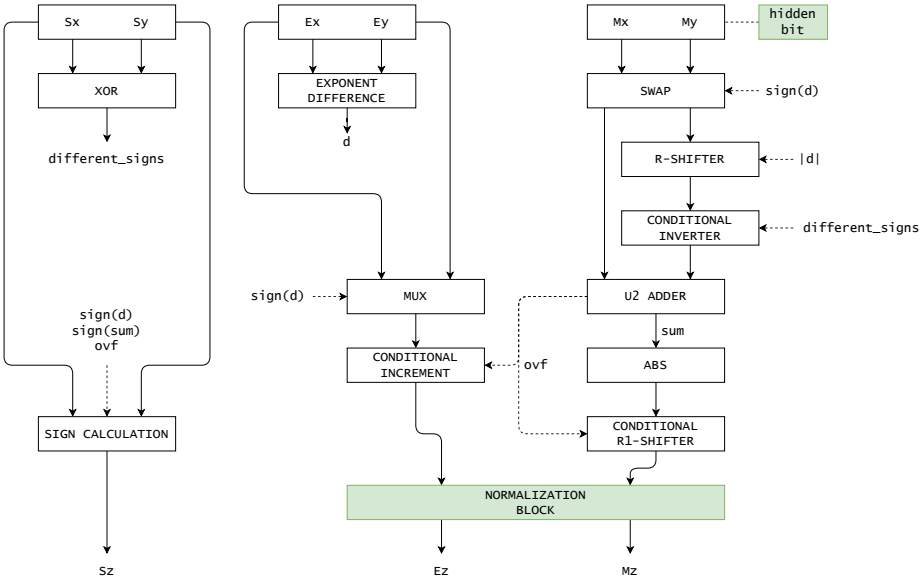


Rysunek 1: Schemat układu dodawania - zdenormalizowane

Na początku układ oblicza różnicę wykładników, żeby następnie wyrównać drugą z nich do tej samej wartości wykładnika. Liczba z mniejszym wykładnikiem jest przesuwana w prawo o ilość bitów równą wartości różnicy wykładników. Jeżeli liczby mają różne znaki, to wyrównywana liczba jest dodatkowo negowana. Umożliwia to sumowanie mantys w U2 wykorzystując obliczony wcześniej bit `different_signs` jako przeniesienie wejściowe. Wewnątrz sumatora wykrywane jest przepełnienie i po obliczeniu wartości bezwzględnej wyniku, wykorzystywane do ewentualnego przesunięcia go w prawo o jeden bit. W takim przypadku zwiększany jest też wykładnik wyjściowy. Wyznaczenie znaku

w przypadku liczb zdenormalizowanych nie jest łatwe. Wynika to z faktu, że nawet gdy wykładnik jednej z liczb jest większy od drugiej, to różnica w mantysach może być jeszcze większa. Wobec tego, konieczne jest wykorzystanie znaków różnicy wykładników, sumy mantys oraz bitu `ovf` informującym o przepełnieniu.

Zaimplementowany przez nas układ dodawania liczb znormalizowanych oparty jest o wersję zdenormalizowaną. Różnice zaznaczyliśmy na rysunku 2 na zielono.

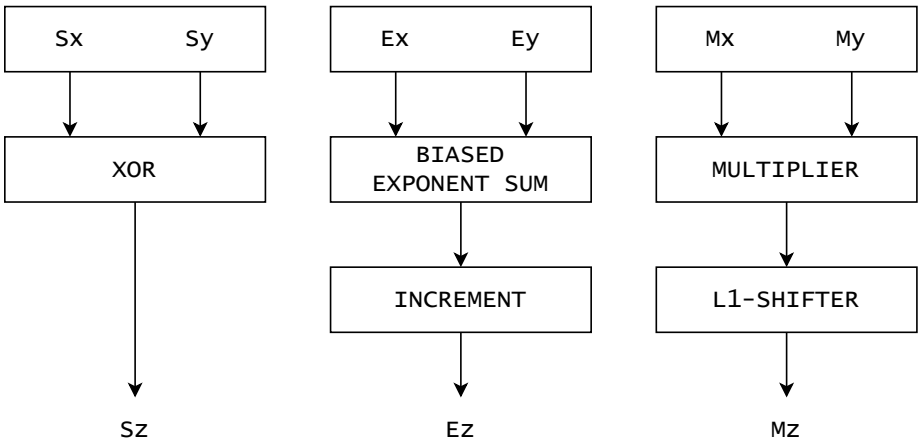


Rysunek 2: Schemat układu dodawania - *IEEE-754*

Pierwszą zmianą jest warunkowe dodanie ukrytej jedynki. Drugą zaznaczoną zmianą jest blok normalizacyjny. Mantysa jest przesuwana w lewo dopóki na pozycji ukrytego bitu nie pojawi się 1. Natomiast inną zmianą, która nie jest uwzględniona na schemacie, jest szerokość bitowa sumy mantys. W wersji liczb znormalizowanych ma ona 48 bitów.

4.2 Układy mnożenia

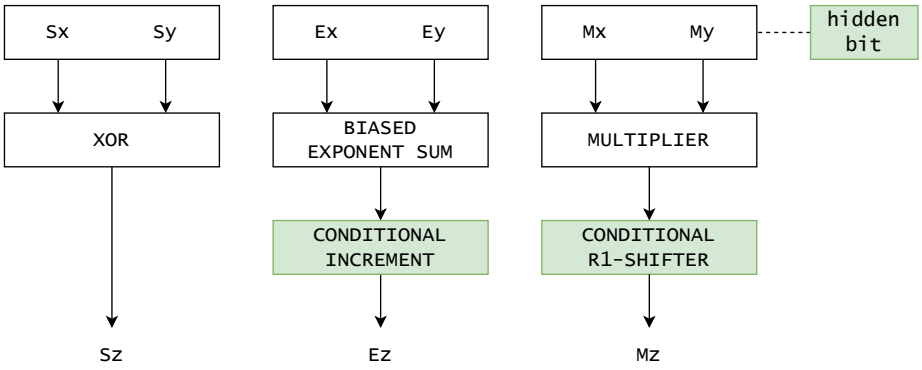
Kolejnym układem, który zaimplementowaliśmy, był układ mnożący, który podobnie jak układ dodawania pochodzi z artykułu [2]. Autorzy nadali mu nazwę *M*. Jest to pierwsza wersja zaproponowanej przez autorów implementacji układu mnożenia liczb zdenormalizowanych.



Rysunek 3: Schemat układu mnożenia - zdenormalizowane

W przeciwieństwie do sumatora, wyznaczenie znaku jest prostą operacją `XOR`. Wykładnik jest sumą wykładników wejściowych z odjętym obciążeniem. Autorzy przyjęli założenie, że operacja mnożenia mantys, zawsze zakończy się przepełnieniem. Wobec tego wykładnik wyjściowy jest zawsze zwiększany o jeden, a mantysa przesuwana w prawo. W przypadku braku wystąpienia przepełnienia, tracony jest jeden bit precyzji.

Zmiany względem układu mnożącego liczby zgodne ze standardem *IEEE-754* są w tym przypadku niewielkie.



Rysunek 4: Schemat układu mnożenia - *IEEE-754*

Ponieważ wynik mnożenia dwóch licz znormalizowanych gwarantuje pojawienie się jedynki na jednym z dwóch najstarszych bitów wyniku, normalizacja sprowadza się do warunkowego przesunięcia w prawo w zależności od wystąpienia przepełnienia.

5 Narzędzia

Do implementacji układów dodawania i mnożenia użyliśmy języka *Verilog*. W *C* napisaliśmy funkcje konwertujące wartości typu `double` na `float` konieczne do porówania wyników. *Python* służył jako język, z którego wywoływane były wszystkie polecenia kompilujące wcześniej wspomniane funkcje w *C*, generujące oraz uruchamiające pliki testowe *Veriloga*. Przy użyciu *Pythona* obliczaliśmy również błędy obliczeniowe wynikające z formatu zdenormalizowanych liczb zmiennoprzecinkowych oraz wykresy je prezentujące. Do tworzenia wykresów posłużyliśmy się biblioteką *Matplotlib*. Narzędzie *Icarus Verilog* posłużyło nam do kompilacji i uruchamiania modułów napisanych w języku *Verilog*. Natomiast do sprawdzenia poprawności wyników poszczególnych bloków i debugowania programu używaliśmy programu *GTKWave*. Do utworzenia diagramów została użyta aplikacja `diagrams.net`. Całość kodu była tworzona w programie *Visual Studio Code*. Testowanie i uruchamianie miało miejsce w systemie Ubuntu na maszynie wirtualnej *WSL 2*.

6 Opis sposobu testowania

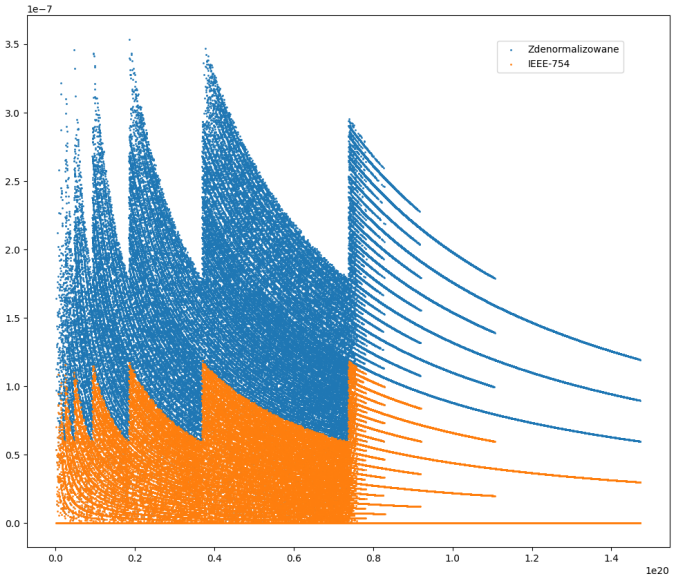
Stworzone układy sumatora i mnożenia liczb zdenormalizowanych zostały przetestowane pod względem różnic pomiędzy wynikami w swoich odpowiednikach w implementacji *IEEE-754*. Testowanie rozpoczeliśmy od wygenerowania liczb pseudolosowych w zakresie, w którym zadbaliśmy o to, aby nie doszło do przepełnienia wykładnika. Wygenerowana liczba utworzona została w języku *Python*, więc była podwójnej precyzji. Przy pomocy zaimplementowanej przez nas funkcji, zmieniliśmy ją na liczbę pojedynczej precyzji.

Kolejnym krokiem w testowaniu było wykonanie odpowiednich dodawań lub mnożeń w zależności od testowanych układów. Program następnie startował symulator uruchamiający kolejne układy. Pierwszym z nich był zaimplementowany przez nas układ działający na liczbach zdenormalizowanych. Drugim był układ częściowo zgodny ze standardem *IEEE-754*. Do tego obliczaliśmy wynik operacji na liczbach podwójnej precyzji, którego używaliśmy jako wartości referencyjnej do porównywania wyników. Tak otrzymane liczby były zapisywane do plików *CSV*.

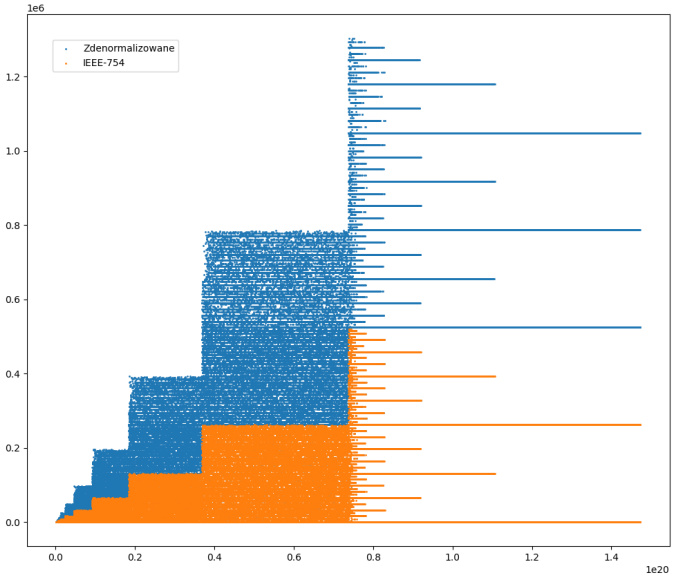
Testowanie przeprowadzilismy dla następujących ilości wyników 100, 1 000, 10 000, 100 000, 1 000 000, 10 000 000. Uznaliśmy, że wykresy najlepiej obrazowały otrzymane wyniki przy 1 000 000 punktów. Możliwe jest wtedy zaobserwowanie szczegółów pozwalających na analizę wyników. Przy większych ilościach punktów, wykresy stawały się nieczytelne.

Następnym krokiem było porównanie poszczególnych wartości liczb zdenormalizowanych i odpowiadającym im liczb referencyjnych. Dla każdego wyniku obliczyliśmy błąd względny używając wyniku liczb podwójnej precyzji jako wartości referencyjnej rysunki 5, 7, 9. Dodatkowo utworzyliśmy wykresy prezentujące błąd bezwzględny w ULP - rysunki 6, 8, 10.

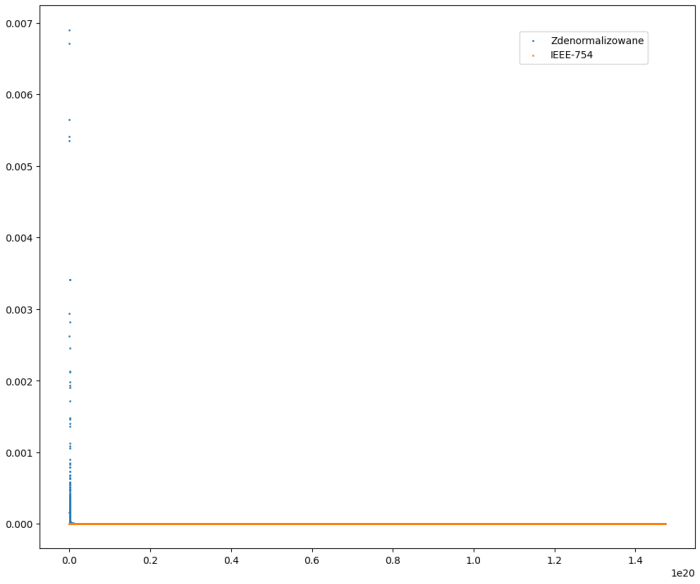
7 Wyniki pomiarów



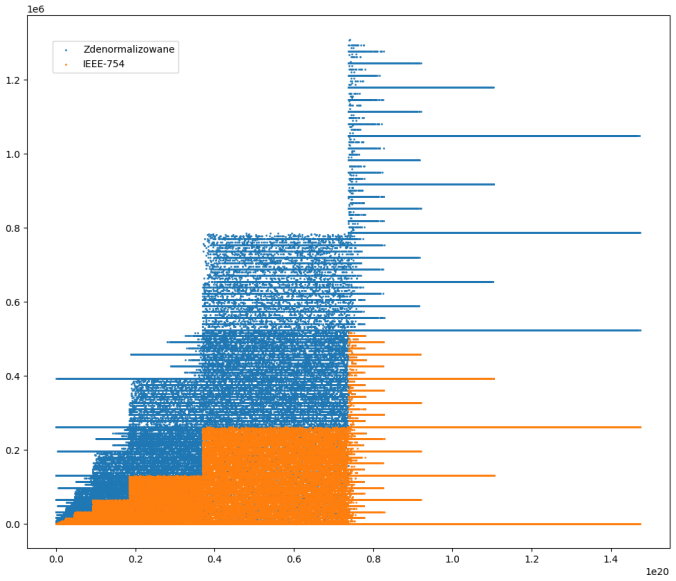
Rysunek 5: Błąd względny - dodawanie



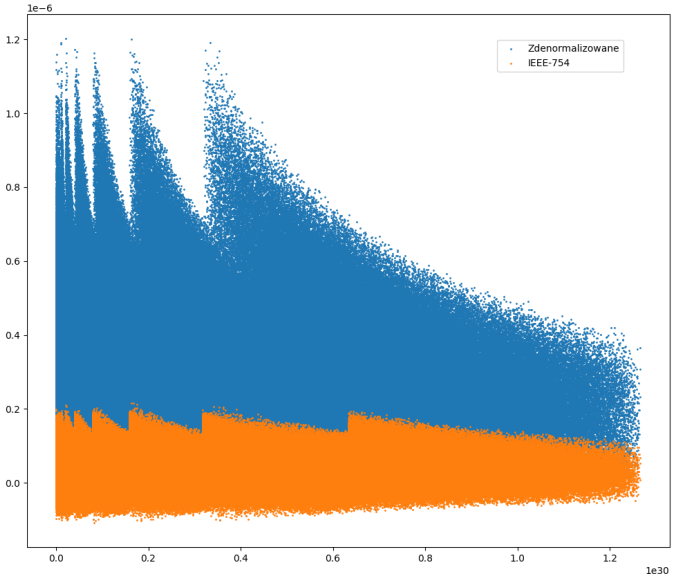
Rysunek 6: Błąd bezwzględny - dodawanie



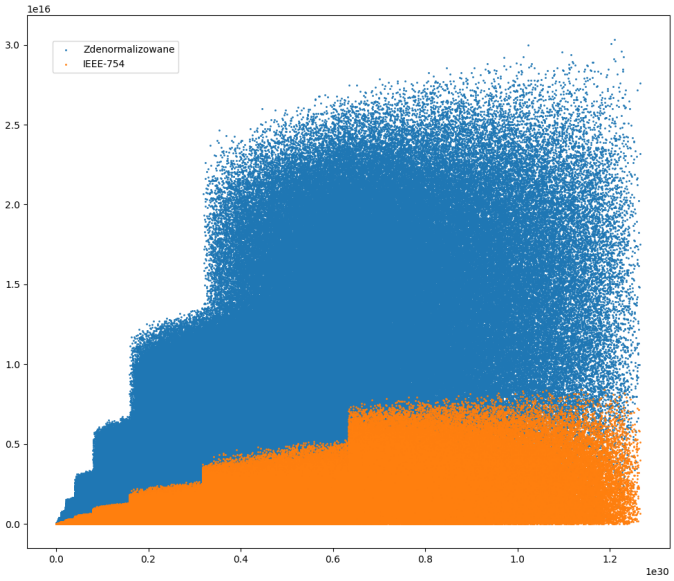
Rysunek 7: Błąd względny - odejmowanie



Rysunek 8: Błąd bezwzględny - odejmowanie



Rysunek 9: Błąd względny - mnożenie



Rysunek 10: Błąd bezwzględny - mnożenie

8 Wnioski

Podstawową i najbardziej zauważalną różnicą jest wydzielenie się dwóch grup punktów przedstawiających wyniki dla liczb zdenormalizowanych i znormalizowanych. W pomiarach widzimy różnicę w utracie dokładności poprzez przesunięcie wszystkich bitów w prawo i pozbycie się ukrytego bitu w liczbach zdenormalizowanych. Różnice w działaniu układów sprawiają, że implementacja dla liczb zdenormalizowanych jest prostsza, zużywa mniej komponentów, a przez to jest mniej energochłonna.

Łatwo zaobserwować linie powstałe na wykresie. Ich obecność wynika ze stałej liczby bitów reprezentacji liczby. Ponieważ zbiór liczb możliwych do przedstawienia w każdym z badanych zapisów jest skończony, to wyniki są zaokrąglane, a widoczne linie są tego wynikiem.

Literatura

[1] A. J. Al-Khalili. Floating point adders and multipliers.

[2] S. Gonzalez-Navarro, J. Hormigo. Normalizing or not normalizing? an open question for floating-point arithmetic in embedded systems. 2017.

[3] P. Srivastava, E. Chung, S. Ozana. Asynchronous floating-point adders and communication protocols: A survey. 2020.