NEURO-SYMBOLIC AI: KOOPERACJA LOGIKI Z SIECIAMI NEURONOWYMI

Index: 285358
www.programistamag.pl
Magazyn programistów i liderów zespołów IT

# programista

4/2026 (125)
Cena 33.90 zł (w tym VAT 8%)

# ESP-NOW

## Protokół komunikacyjny dla drona

JUŻ W EMPIKU

ISSN 2084-9400 04

- Komputer niebinarny. Czy to jest możliwe?
- Programista wspomagany sprzętowo
- Czy można zastąpić płatne AI lokalnymi?
- Własny bot na Discordzie
- ECS w silnikach gier i symulacji
- Strefa CTF

PERFECTION IN PROTECTION LICENSING SECURITY
WIBU SYSTEMS

# Alternatywne formaty liczb zmiennoprzecinkowych

W numerze 5/2024 (115) „Programisty” ukazał się artykuł „Tajemnice liczb zmiennoprzecinkowych”, w którym omówione zostały przede wszystkim liczby zgodne z formatem IEEE 754. Choć format ten jest najbardziej rozpowszechniony i najczęściej używany, nie jest oczywiście jedynym. W niniejszym artykule chciałbym przybliżyć alternatywy dla niego, których rozwój jest ściśle związany z obliczeniami wykonywanymi w algorytmach uczenia maszynowego.

## WSTĘP

Dla porządku przypomnijmy krótko, w jaki sposób reprezentowane są liczby zmiennoprzecinkowe w pamięci komputera – czyli w postaci binarnej. Z grubsza reprezentacja dzieli się na dwie części o określonej, stałej długości: mantysę (część ułamkową – ang. *significand* lub *fraction*) i wykładnik (albo cechę – ang. *exponent*). Wykładnik to potęga dwójki, przez którą pomnożyć należy mantysę, aby otrzymać kodowaną liczbę zmiennoprzecinkową. Liczba bitów kodujących mantysę i wykładnik różni się w zależności od zastosowanego formatu, podobnie jak forma ich przechowywania – np. w najpopularniejszym standardzie IEEE 754 wykładnik przechowywany jest w formie przesuniętej (ang. *biased*), a manysa – znormalizowanej (po szczegółowe wyjaśnienia odsyłam do wspomnianego już artykułu z numeru 5/2024). Dla liczb ze znakiem mamy też bit znaku, część formatów wyróżnia też „wartości specjalne”, np. nieskończoność.

Pomysł, by liczby przedstawiać w postaci oddzielnie zapisywanego wykładnika i mantysy (a dodatkowo jeszcze *bazy* – czyli liczby, którą potęgujemy; w przypadku komputerów jest to zwykle dwójka, w powszechnie stosowanej notacji naukowej – dziesiątka), nie jest, delikatnie mówiąc, nowy. Pierwszy raz zastosował go Archimedes w dziele „Psammites”, traktującym o liczeniu ziaren piasku potrzebnych do wypełnienia całego wszechświata (spoiler alert: $80 \cdot 10^{15}$). Prawie dwa tysiące lat później koncepcja ta została zastosowana w arytmetyce zmiennoprzecinkowej przez hiszpańskiego polimatę Leonardo Torresa Quevedo, który w 1914 roku przedstawił ją w teorii, a w 1920 roku w praktyce, demonstrując swój elektromechaniczny arytmometr (maszyna była nawet wyposażona w klawiaturę oraz umożliwiała zdalny dostęp po drutach elektrycznych). Pierwsza implementacja na działającym komputerze jest dziełem niemieckiego inżyniera Konrada Zuse, który używał 22-bitowych liczb zmiennoprzecinkowych w swoich zbudowanych w latach 1938-1941 komputerach V1, V2 i V3 (przemianowanych potem, z oczywistych przyczyn, na Z1, Z2 i Z3.) *Notabene* Z3 był pierwszą w historii maszyną kompletną w sensie Turinga, czyli zdolną do wykonania każdego algorytmu.

Zanim doczekaliśmy się standaryzacji, istniało wiele różnych, z zasady niekompatybilnych, formatów liczb zmiennoprzecinkowych, które były wprowadzane wraz z rozwojem poszczególnych architektur komputerów. Popularność danego formatu zależała w dużej mierze od popularności sprzętu, który ten format wspierał. I tak od 1964 roku IBM stosował format System/360, który oferował liczby 32- i 64-bitowe, i używał heksadecymalnej bazy, co dawało duży zakres, ale mniejszą precyzję, niż w systemach o bazie binarnej. Mniej więcej w tym samym czasie CDC (Control Data Corporation, firma, która swego czasu produkowała najszybsze superkomputery na świecie) wprowadziła własny, wielce osobliwy format 60-bitowy o bazie 2, a DEC (Digital Equipment Corporation, w owym czasie również ogromny gracz na rynku komputerów) 32- i 64-bitowe, również o podstawie binarnej. Taka sytuacja trwała do połowy lat ‘80, kiedy to opublikowano standard IEEE 754. Prace nad nim trwały od końca lat ‘70, a głównym jego architektem był kanadyjski matematyk William Kahan. Zupełnie przypadkowo był on również głównym architektem koprocesora arytmetycznego 8087, stworzonego przez Intel w 1980 roku, więc komputery osobiste wspierały standard jeszcze przed jego oficjalną publikacją. Kiedy się to stało, w 1985 roku, CDC było już w tarapatach, DEC wpadł w nie kilka lat później, podobnie jak IBM. Silna pozycja rynkowa Intela i zawirowania na rynku komputerów pomogły w upowszechnieniu „jedynego słusznego” formatu.

Zacznijmy od stwierdzenia oczywistości: IEEE 754 jest i w przewidywalnej przyszłości będzie najpowszechniejszym formatem zapisu liczb zmiennoprzecinkowych. Standaryzacja, powszechne wsparcie sprzętowe, dojrzały ekosystem oprogramowania ze wsparciem w ogromnej większości języków programowania, kompilatorów, bibliotek numerycznych i aplikacji naukowych zoptymalizowanych pod kątem tego formatu powodują, że dominuje on od dziesięcioleci. Ogromna baza istniejącego sprzętu, oprogramowania, wiedzy i doświadczenia powodują, że zmiana na jakikolwiek nowy standard wiązałaby się z gigantycznymi kosztami i wysiłkiem. Standard IEEE 754 to wybór domyślny – ale nie jedyny.

## LICZBY ZMIENNOPRZECINKOWE W AI

Dawno, dawno temu, na początku lat 2000, ktoś wpadł na pomysł, żeby wykorzystać kartę graficzną do czegoś innego, niż liczenie jakichś tam shaderów, wokseli i innych poligonów. Idea ta nazywa się teraz GPGPU (ang. *general-purpose computing on graphics processing units*), czyli obliczeniami ogólnego zastosowania na układach graficznych. Rzeczone układy mają ze swojej natury architekturę równoległą, z dziesiątkami (obecnie dziesiątkami tysięcy) rdzeni, znakomicie nadających się do rozwiązywania różnych problemów inżynieryjnych, wymagających np. operacji na macierzach (kto nie słyszał o metodzie elementów skończonych, niechaj pierwszy rzuci

dwuteownikiem stalowym). W 2007 roku NVIDIA odpowiedziała na rosnące zapotrzebowanie rynku wprowadzeniem platformy CUDA, zestawu API i narzędzi, które znacznie ułatwiły GPGPU, i niedługo potem ktoś zauważył, że na GPU świetnie trenuje się sieci neuronowe. W 2012 roku dwóch studentów z Kanady wygrało prestiżowy konkurs rozpoznawania obrazów [1] siecią wytrenowaną na kartach graficznych. Od tego czasu według jednych „puchnie bańka", a według innych „toczy się supercykl" sztucznej inteligencji. Dlaczego o tym piszę? Bo potrzebne są do tego formaty zmiennoprzecinkowe inne niż IEEE 754.

Sieci neuronowe wykazują pewną odporność na szum, błędy i zaokrąglenia w obliczeniach. Wynika to z jednej strony z ich nadmiarowości, a z drugiej ze zdolności do uczenia się na podstawie statystycznych wzorców, a nie absolutnej precyzji każdej pojedynczej operacji wewnątrz sieci. Zbudowany (czy raczej wyuczony) na sieci neuronowej model będzie działał poprawnie, nawet gdy część parametrów będzie „trochę" niedokładna. Ponieważ w uczeniu maszynowym często okazuje się, że można osiągnąć „wystarczająco dobrą" dokładność odpowiedzi mniejszym nakładem kosztów, opracowano wiele metod optymalizacji modeli. Wszystkie mają na celu znalezienie równowagi między wydajnością (szybkością działania), zapotrzebowaniem na zasoby (pamięć i moc obliczeniową, a pośrednio energię, potrzebną do wykonania obliczeń i odprowadzenia wytworzonego przez nie ciepła) a dokładnością. W zależności od przyjętego celu optymalizacji może okazać się, że satysfakcjonująca dokładność jest bardziej wartościowa, niż marginalny wzrost uzyskany dużym kosztem. Część tych metod oparta jest na idei kwantyzacji, czyli procesu przekształcania wartości o dużej precyzji na wartości o precyzji niższej. Robi się to albo po zakończeniu treningu (ang. *Post-Training Quantization*, PQT), albo w jego trakcie (ang. *Quatnization-Aware Training*, QAT). W praktyce (i wielkim uproszczeniu) polega to na zmapowaniu 32-bitowych zmiennoprzecinkowych wartości parametrów wewnątrz sieci na wartości o niższej precyzji.

IEEE 754 proponuje co prawda format 16-bitowy liczby (ang. *half-precision*, czyli półprecyzja, w odróżnieniu od pełnej, 32-bitowej precyzji; formaty te skrótowo nazywa się FP16 i FP32), ale na wykładnik przeznaczono w nim tylko 5 bitów, co przekłada się na ograniczony zakres dynamiczny (czyli stosunek między największą a najmniejszą wartością, jaką można zapisać za pomocą danej reprezentacji; szczegóły dotyczące struktury i zakresu omawianych reprezentacji przedstawiono w Tabeli 1). Wartości podlegające przetwarzaniu (celowo nie wchodzę tu w szczegóły) w sieciach neuronowych mogą zmieniać się w bardzo szerokim zakresie i gdy używamy formatu o zbyt małym zakresie, możemy napotkać problem z utratą informacji spowodowaną niedomiarem (ang. *underflow* – liczby stają się zbyt małe, by je reprezentować) lub nadmiarem (ang. *overflow*, analogicznie). Format FP32 jest dobrym, bezpiecznym punktem wyjścia – niestety, pamięciożernym i wolnym. FP16 jest szybszy i oszczędny, ale często wymaga zastosowania dodatkowych technik, aby zapobiec problemom z przekroczeniem zakresu.

Około 2017 roku inżynierowie z zespołu badawczego zajmującego się w ramach Google sztuczną inteligencją (Google Brain) zaproponowali format bfloat16 (*Brain Floating Point*), w którym wykładnik ma 8 bitów (jak w FP32), ale manysa tylko 7. Dzięki temu cała liczba zajmuje tylko 16 bitów, oferując jednocześnie taki sam zakres dynamiczny, co dłuższy odpowiednik z IEEE 754. Format ten jest wspierany sprzętowo przez oferowane przez Google układy TPU (Tensor Processing Unit) od wersji v2. W nieco innym kierunku poszła NVIDIA. W 2020 roku, wraz z architekturą Ampere, wprowadzono format TF32 (*TensorFloat32*). Jest to przeznaczony do obliczeń format wewnętrzny akceleratorów AI Tensor Core – kod jest pisany z wykorzystaniem FP32, z którego wykładnik (8 bitów) jest używany bez zmian, natomiast manysa jest obcinana do 10 bitów. Powstaje nieco dziwaczny, 19-bitowy (z bitem znaku) twór, który jednak daje przyspieszenie obliczeń porównywalne z FP16, ale z zakresem FP32.

Od wprowadzenia swojej architektury Hopper w 2022 roku NVIDIA oferuje jeszcze bardziej agresywną optymalizację – ośmiobitowy format FP8 [2]. Oferowany jest on w dwóch wariantach, różniących się podziałem bitów między wykładnik (4 lub 5 bitów) a mantysę (odpowiednio 3 lub 2; tradycyjnie występuje też bit znaku), a co za tym idzie zakresem dynamicznym i precyzją. Dla rozróżnienia nazywa się je FP8-E5M2 i FP8-E4M3. Redukcja zużycia pamięci pozwala na użycie większych modeli AI i wdrażanie ich na urządzeniach z ograniczoną pamięcią (na przykład na urządzeniach brzegowych IoT). Dodatkowo zwiększa się szybkość obliczeń, bo mamy mniej danych do przenoszenia między pamięcią a jednostkami obliczeniowymi, a co za tym idzie zużywamy mniej energii i emitujemy mniej ciepła. Oprócz NVIDII oba warianty FP8 są wspierane we wprowadzonej w 2021 roku architekturze ARMv9-A, a konkretnie w będącym jej częścią rozszerzeniu instrukcji wektorowych SVE2 (Scalable Vector Extension 2), architekturze Intel Gaudi 2 (2022 r.) oraz AMD CDNA 3 (2023 r.). Google również aktywnie bada i wykorzystuje 8-bitowe formaty zmiennoprzecinkowe, chociaż nie ogłoszono oficjalnie wprowadzenia odpowiednika bfloat16. Formaty te są częścią standardu Open Compute Project OFP8 [3].

Formaty zmiennoprzecinkowe o reprezentacji krótszej niż 16 bitów są czasami nazywane *minifloats*. We wspomnianej już architekturze Hopper NVIDIA wprowadziła również wsparcie dla FP4-E2M1, czyli czterobitowego formatu z dwoma bitami wykładnika i jednym mantysy. Jest to najmniejszy format zmiennoprzecinkowy, który można zbudować „zgodnie z zasadami" IEEE 754, tzn. z bitem znaku, znakowanym zerem i nieskończonością oraz wartością NaN (w swojej implementacji NVIDIA pomija wartości i ±∞NaN, dzięki czemu można reprezentować liczby od -6 do 6, zamiast od -3 do 3) [4]. Oczywiście oferowane przez ten format precyzja i zakres są bardzo ograniczone, ale jego głównym zadaniem jest minimalizacja zużycia pamięci i zwiększenie efektywności obliczeń. Minifoaty są tak małe, że zamiast wykonywać na nich „prawdziwe" obliczenia, można użyć zapisanej w pamięci tabeli przeglądowej (ang. *lookup table*).

Rozwiązaniem problemu z niewielkim zakresem dynamicznym jest zastosowanie techniki znanej co najmniej od lat '70 ubiegłego wieku – *block floating point* (blokowa arytmetyka zmiennoprzecinkowa). Pierwotna idea pozwalała na programową implementację arytmetyki zmiennoprzecinkowej na architekturach, które nie wspierały jej sprzętowo. W podejściu tym pojedynczy, wspólny wykładnik jest używany przez zestaw mantys. Na potrzeby uczenia maszynowego wyspecyfikowano bazujący na tej idei zestaw formatów nazwanych MX (*Microscaling*) [5]. W MX minifloaty (FP8, FP4 oraz sześciobitowy FP6 – dwa ostatnie zdefiniowane bez wartości specjalnych; dodatkowo standard przewiduje użycie całkowitoliczbowego INT8)

przechowywane są w grupach po 32 elementy, do których przypisany jest wspólny, ośmiobitowy wykładnik (ze zdefiniowaną wartością specjalną NaN, aplikowaną do całego bloku). W praktyce, oprócz wspomnianych wcześniej technik optymalizacji typu QAT i PQT, do obliczeń w modelach AI używana jest też arytmetyka zmiennoprzecinkowa o zmiennej precyzji (ang. *mixed precision arithmetics*), gdzie w pojedynczej operacji używane są liczby o różnej długości (np. pomnożenie dwóch liczb ośmiobitowych daje szesnastobitową) – na przykład popularna biblioteka PyTorch oferuje API do automatycznego stosowania jej w treningu modeli AI [6].

Dla porządku powinienem wspomnieć jeszcze o używanych przez firmę Tesla formatach przeznaczonych dla architektury D1 komputera Tesla Dojo. Służy on rzekomo (ani układy D1, ani urządzenia na nich oparte nie są dostępne komercyjnie) do szkolenia sieci neuronowych w kontekście autonomicznej jazdy. Formaty te to CFloat16 i CFloat8 – pierwszy ma 10 bitów w wykładniku i 5 w mantysie, natomiast drugi jest odpowiednikiem standardowych FP8 [7]. Litera „C" w nazwie pochodzi od *configurable* – elementem konfigurowanym jest przesunięcie (*bias*) wykładnika (jest to liczba odejmowana od zapisanej wartości wykładnika po to, żeby można było zakodować zarówno wartości dodatnie, jak i ujemne). We wszystkich wymienionych tu wcześniej formatach jest ono stałe (np. w formacie FP32 wynosi 127). Trudno powiedzieć, jakie korzyści to przynosi, bowiem szczegóły implementacyjne nie są znane. Być może zaproponowanie własnych formatów to objaw syndromu *not invented here*, na który cierpi Tesla, a może przygotowanie do trollingu patentowego (który ma miejsce również w dziedzinie formatów danych – Google został pozwany za użycie konkretnego zakresu dynamicznego w bfloat16 i w ramach ugody wypłacił powodowi nieujawnioną sumę [8]). W każdym razie konfigurowalność przesunięcia ma zapewnić spełnienie wymagań co do zakresu różnych parametrów bez zwiększania liczby bitów wykładnika.

## TAPERED FLOATING POINT I POSIT

W IEEE 754 i pokrewnych formatach struktura reprezentacji jest stała – znak, wykładnik i mantysa mają w danej reprezentacji zawsze tyle samo bitów. Powoduje to, że mają w przybliżeniu stałą precyzję względną, czyli procentową różnicę między kolejnymi możliwymi do reprezentowania wartościami. Oznacza to jednak, że różnica bezwzględna rośnie wraz z wartością liczby. Odległość między dwoma kolejnym liczbami zmiennoprzecinkowymi nazywa się ULP (ang. *unit of least precision* lub *unit in the last place* – jednostka najmniejszej precyzji lub jednostka ostatniego miejsca). Przyjrzyjmy się teraz, jak zmienia się ona dla liczb różnej wielkości w reprezentacji. Dla uproszczenia przykładu użyjemy reprezentacji o bazie dziesiętnej. Załóżmy, że w naszej reprezentacji liczba ma postać $d.dd\times10^w$, gdzie $w$ to wykładnik, a mantysa $d.dd$ ma trzy cyfry znaczące.

1. Liczby bliskie 1
   - pierwsza liczba $l_1 = 1.00\times10^0 = 1.00$
   - kolejna większa liczba, którą można reprezentować $l_2 = 1.01\times10^0 = 1.01$
   - różnica bezwzględna $l_1 - l_2 = 0.01$
   - różnica względna $(l_1 - l_2)/ l_1 = 0.01$
2. Liczby bliskie 100
   - pierwsza liczba $l_1 = 1.00\times10^2 = 100$
   - kolejna większa liczba, którą można reprezentować $l_2 = 1.01\times10^w = 101$
   - różnica bezwzględna $l_1 - l_2 = 1$
   - różnica względna $(l_1 - l_2)/ l_1 = 0.01$
3. Liczby bliskie 0.01
   - pierwsza liczba $l_1 = 1.00\times10^{-2} = 0.01$
   - kolejna większa liczba, którą można reprezentować $l_2 = 1.01\times10^{-2} = 0.01$
   - różnica bezwzględna $l_1 - l_2 = 0.0001$
   - różnica względna $(l_1 - l_2)/ l_1 = 0.01$

Jak widać, im większa jest wartość, tym większa ULP, a co za tym idzie mniejsza precyzja. Istnieją jednak reprezentacje, w których liczba bitów przeznaczonych na mantysę (a tym samym precyzja) zmienia się w zależności od wielkości liczby, bez zmiany długości samej reprezentacji (jak to ma miejsce w arytmetyce zmiennoprzecinkowej o zmiennej precyzji). Są to liczby zmiennoprzecinkowe o zawężającej się precyzji (ang. *tapered floating point*). Pojęcie zaproponowano

| Format | Wykładnik (bity) | Mantysa (bity jawne) | Przybliżony zakres dynamiczny (podstawa dziesiętna) | Przybliżony zakres dynamiczny (podstawa binarna) |
|---|---|---|---|---|
| FP32 | 8 | 23 | od $\pm1.18\times10^{-38}$<br>do $\pm3.40\times10^{38}$ | od $\pm2^{-126}$<br>do $\pm1.99\times2^{127}$ |
| FP16 | 5 | 10 | od $\pm6.10\times10^{-5}$<br>do $\pm6.55\times10^{4}$ | od $\pm2^{-14}$<br>do $\pm1.99\times2^{15}$ |
| bfloat16 | 8 | 7 | od $\pm1.18\times10^{-38}$<br>do $\pm3.40\times10^{38}$ | od $\pm2^{-126}$<br>do $\pm1.99\times2^{127}$ |
| tensorfloat32 | 8 | 9 | od $\pm1.18\times10^{-38}$<br>do $\pm3.40\times10^{38}$ | od $\pm2^{-126}$<br>do $\pm1.99\times2^{127}$ |
| FP8-E5M2 | 5 | 2 | od $\pm6.10\times10^{-5}$<br>do $\pm5.73\times10^{4}$ | od $\pm2^{-14}$<br>do $\pm1.75\times2^{15}$ |
| FP8-E4M3 | 4 | 3 | od $\pm1.56\times10^{-2}$<br>do $\pm2.40\times10^{2}$ | od $\pm2^{-6}$<br>do $\pm1.875\times2^{7}$ |

*Tabela 1. Struktura i przybliżone zakresy dynamiczne w różnych formatach liczb zmiennoprzecinkowych. Zakres podano dla liczb znormalizowanych ze znakiem*

w króciutkim (niecałe 2 strony) artykule z 1971 roku [9] i niezbyt dużo się w tym obszarze działo, nawet w rozważaniach teoretycznych aż do roku 2013 i prezentacji niejakiego Gustafsona [10].

John Leroy Gustafson jest znany przede wszystkim z nazwanego jego nazwiskiem prawa dotyczącego przetwarzania równoległego [11]. Notabene współautor tegoż prawa, Edwin H. Barsis, nie załapał się nawet na notkę w Wikipedii, choć osiągnięcia naukowe na polu przetwarzania równoległego ma niemałe, a i życiorys dość ciekawy – z wykształcenia jest fizykiem jądrowym i zawodowo zajmował się produkcją bomb atomowych. We wspomnianej prezentacji Gustafson zaproponował koncepcję Unum (od *universal numbers*), system liczbowy bazujący na idei *tapered floating point*. Podstawowymi założeniami były zmienne długości pól dla mantysy i wykładnika (a więc i zmienna precyzja) i specjalny *u-bit*, który wskazywał, czy dany unum to dokładna liczba, czy też przedział między kolejnymi dokładnymi unumami. Do obliczeń na unumach miała być stosowana arytmetyka przedziałowa, dział matematyki, którego pierwotnym zastosowaniem była ścisła kontrola błędów zaokrągleń w obliczeniach numerycznych na liczbach rzeczywistych. Pierwotna idea, nazywana teraz „Type I Unum", była niepraktyczna (złożona w implementacji i kosztowna obliczeniowo), podobnie jak zaproponowany 3 lata później „Type II". Dopiero w 2017 roku Unum Type III, znany obecnie jako *Posit*, osiągnął na tyle dużą dojrzałość, że można było myśleć o jego implementacji sprzętowej. Wydany 5 lat później standard przemysłowy [12] definiuje konkretne typy danych (8-,16- i 32-bitowy). Ogólna struktura to 1 bit znaku, zmienna liczba bitów reżimu (ang. *regime* – co najmniej 2 bity; w uproszczeniu reżim określa skalę liczby, co umożliwia manipulację jej zakresem), dwa bity na wykładnik i zmienna liczba bitów mantysy (im dłuższy reżim, tym krótsza mantysa i odwrotnie).

Obliczenie wartość liczby Posit wygląda podobnie do IEEE 747, przy czym wykonywane jest jeszcze mnożenie wykładnika przez czynnik zależny od reżimu. Pokażę to na przykładzie posit16, zainteresowanych przypadkiem ogólnym (posit*n*) odsyłam do standardu.

Liczba Posit składa się z następujących elementów:

1. Bit znaku *s* (0 oznacza liczbę dodatnią)
2. Reżim *r* – sekwencja *k* identycznych bitów (samych 0 lub 1), zakończona bitem o przeciwnej wartości (długość reżimu to k+1). Jeśli reżim zaczyna się od 1, to r = k, w przeciwnym wypadku r = -k
3. Wykładnik *e* – następująca po reżimie sekwencja o stałej długości *es*, jej wartość jest interpretowana jako liczba całkowita bez znaku; do obliczenia wartości liczby używa się wartości $d = 2^{es}$
4. Mantysa – pozostałe bity, wartość mantysy *m* jest interpretowana jako część ułamkowa 0.xxx…, gdzie xxx… to bity mantysy.

Wartości specjalne to 0 (gdy wszystkie bity są zerowe) i NaR (*Not a Real*, odpowiednik NaN w IEEE 754 – gdy bit znaku wynosi 1, a wszystkie pozostałe bity są zerowe).

Wartość posit *P* jest obliczana wg wzoru:

$$P = (-1)^s \times 2^{(r \times d + e)} \times (1 + f)$$

Załóżmy teraz, że mamy liczbę posit16 (es=2, czyli d=4) o reprezentacji 0111010010000000. Zatem:

- pierwszy bit (bit znaku) to 0 – liczba dodatnia
- reżim ma postać 1110, kolejny bit to 0, zatem reżim to 111; liczba identycznych bitów k=3, czyli wartość reżimu r=k=3
- kolejne dwa bity to wykładnik, w typ przypadku 10, czyli e = 2
- pozostałe 11 bitów to mantysa, jej wartość to 1.01000000000, czyli 1.25

/* REKLAMA */

Podstawiając do wzoru, mamy:

$P = (-1)^0 \times 2^{(3x4+2)} \times 1.25 = 20480$

Kluczowym elementem sprzętowej implementacji Posit jest *quire* – bardzo duży akumulator (wzn. rejestr) stałoprzecinkowy, którego rozmiar jest znacznie większy niż rozmiar pojedynczej liczby Posit (np. dla operacjach na posit32 wymagany jest quire o długości 512 bitów). Jego podstawową funkcją jest dokładne sumowanie iloczynów (np. a*b + c*d + e*f itd.) bez wprowadzania błędów zaokrągleń na każdym etapie dodawania – zamiast zaokrąglać każdy wynik mnożenia do Posit przed dodaniem go do sumy, iloczyny są obliczane z pełną precyzją i dodawane bezpośrednio do quire. Dopiero końcowy wynik z quire jest zaokrąglany do Posit. W przypadku IEEE 754 każdy kolejny iloczyn jest zaokrąglany do najbliższej liczby zmiennoprzecinkowej, każde zaokrąglenie wprowadza mały błąd, a błędy kumulują się przy sumowaniu. Implementacja qiure jest złożona i kosztowna obliczeniowo, ale dzięki zwiększeniu dokładności zastosowanie Positów może prowadzić do zysków wydajnościowych na bardziej ogólnym poziomie (np. przez użycie mniejszych formatów danych zachowując wymaganą precyzję czy wyeliminowanie konieczności stosowania algorytmów kompensacji błędów zaokrągleń). Implementacja sprzętowego wsparcia dla formatu Posit (na razie w postaci FPGA) jest przedmiotem zainteresowania wielu grup badawczych, a dotychczasowe rezultaty wydają się obiecujące [13-15].

## PODSUMOWANIE

Dominacja standardu IEEE 754 na polu formatów zmiennoprzecinkowych jest niepodważalna. Praktycznie żaden producent procesorów ogólnego zastosowania nie oferuje wsparcia dla czegokolwiek innego. Powody są prozaiczne: nieprzeliczone miliardy linii kodu opierają się na tym standardzie, sprzęt jest zoptymalizowany pod jego kątem, a jego zmiana wymagałaby ogromnych inwestycji w sprzęt i oprogramowanie. Dopiero burzliwy rozwój w dziedzinie sztucznej inteligencji w ostatniej dekadzie wymusił na producentach sprzętu poszukiwanie alternatywnych reprezentacji. Nadal jednak są to formaty oparte na filozofii dominującego standardu – stałej liczbie bitów w poszczególnych częściach reprezentacji. Potrzeba optymalizacji może jednak wymusić odejście od tego paradygmatu i pójście inną drogą – formatu z niejako „wbudowaną" cechą zmiennej precyzji. Czas pokaże, czy formatem tym będzie Posit (lub jego następca). Wyniki dotychczasowych badań pozwalają przypuszczać, że wsparcie sprzętowe dla tego formatu zacznie pojawiać się na rynku – początkowo w specjalistycznych akceleratorach dla sztucznej inteligencji i uczenia maszynowego, gdzie trzeba pogodzić dokładność, szybkość działania i efektywność energetyczną. Bez wsparcia największych graczy na rynku „uniwersalne liczby" pozostaną jednak tym, czym były do niedawna – akademicką ciekawostką bez szerszego zastosowania w praktyce.

### Bibliografia


[1] https://en.wikipedia.org/wiki/AlexNet
[2] https://developer.NVIDIA.com/blog/floating-point-8-an-introduction-to-efficient-lower-precision-ai-training/
[3] https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1
[4] https://developer.NVIDIA.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/
[5] https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf
[6] https://residentmario.github.io/pytorch-training-performance-guide/mixed-precision.html
[7] https://cdn.motor1.com/pdf-files/535242876-tesla-dojo-technology.pdf
[8] https://www.reuters.com/technology/google-settles-ai-related-chip-patent-lawsuit-that-sought-167-bln-2024-01-24/
[9] R. Morris, „Tapered Floating Point: A New Floating-Point Representation", IEEE Transactions on Computers, vol. C-20, no. 12, 1971 https://ieeexplore.ieee.org/document/1671767
[10] http://www.johngustafson.net/presentations/Right-SizingPrecision1.pdf
[11] https://pl.wikipedia.org/wiki/Prawo_Gustafsona
[12] https://posithub.org/docs/posit_standard-2.pdf
[13] M. s. Alapati, R. Swain, A. K. Panigrahy, „Hardware Implementation of Posit Numeration System using FPGA for Signal Processing Applications," 7th International Conference on Trends in Electronics and Informatics, 2023 https://ieeexplore.ieee.org/document/10125920
[14] Naohito Nakasato, Yuki Murakami, Fumiya Kono, Maho Nakata. „Evaluation of POSIT Arithmetic with Accelerators", International Conference on High Performance Computing in Asia-Pacific Region, 2024 https://dl.acm.org/doi/10.1145/3635035.3635046
[15] A.Raveendran, V.Desalphine „A Posit Arithmetic Unit Enabled RISC-V Processor", RISC-V Summit, 2024 https://www.youtube.com/watch?v=PmnQ3YT-mng

**MACIEJ NORBERCIAK**

maciej.norberciak@nokia.com


Od początku kariery w przemyśle zajmuje się systemami wbudowanymi w telekomunikacji, od ponad dekady jako architekt w Nokii. Doktor nauk technicznych w dziedzinie sztucznej inteligencji. Mentor, mówca, trener. W chwilach wolnych od bycia tatą lubi się uczyć nowych rzeczy, głównie języków obcych, jeździć na rowerze i pracować w ogrodzie.



Zamów Programistę w prenumeracie >>