Posit-Based Adder and Multiplier Design for MAC Unit

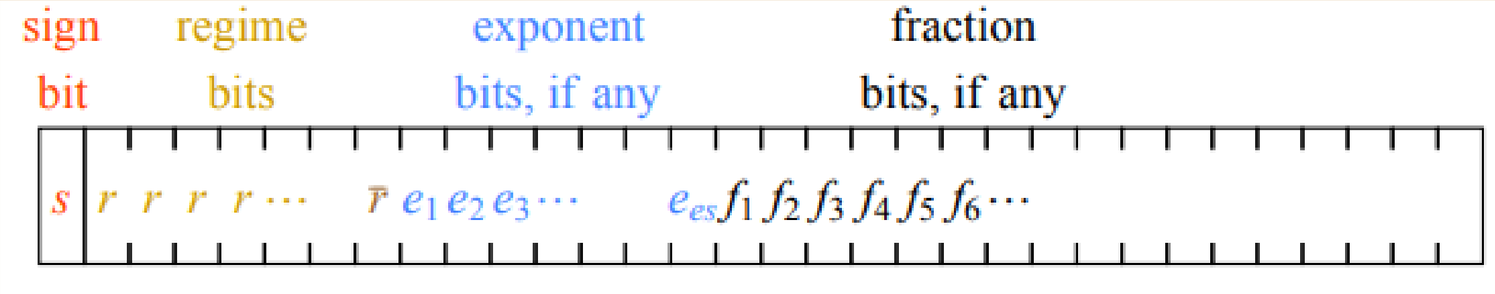
CONTENTS

Introduction of MAC Unit

- MAC - Multiplication & Accumulation Unit
- AI/ML algorithmic architectures have many variants including Deep Neural Networks (DNNs), which are popular due to their diverse applications in various fields viz. scientific, engineering, agriculture, healthcare etc.
- DNN consists of one input layer, one output layer, and multiple hidden layers. Each DNN layer performs matrix (vector) multiplication and accumulation, and a layer's output is input to the subsequent layers within the network.
- The MAC unit is present in each neuron of the hidden layers to perform the multiplication & accumulation operations.
- In general, the arithmetic representations for various numbers in neural network computations are IEEE-754 float, fixed-point, bfloat16 or Posit.

POSIT ARITHMETIC

- Posit is the latest development in Universal Number (unum) system under type-3 unum. Posit is claimed as a possible substitute for floating point (FP) number system.
- The structure of an N-bit Posit representation with ES exponent bits as follows-



- The Sign bit in posit is 0 for positive numbers and 1 for negative numbers. For the case of negative number, first take 2's complement before decoding regime, exponent and mantissa bits.
- There are only two exception cases: zero and infinity.
- Zero - (000...000) representation
- Infinity - (1000...000) representation.

- For all other cases, the value x of a posit is given by -

$$x = (-1)^{MSB} \times useed^{k} \times 2^{exp} \times (1 + \sum_{i=1}^{fn-1} b_{fn-1-i} 2^{-i})$$

- The regime indicates a scale factor of useed^k where useed = 2^(2^ES) and ES is the exponent size. The numerical value of k is determined by the run length of 0 or 1 bits in the string of regime bits.

$$\begin{aligned}
&\text{With ES=2: } 0_0001_11_1 = +(2^4)^{-3} \times 2^3 \times (1 + \tfrac{1.0}{2}), \text{ and} \\
&11101011 \rightarrow 0_001_01_01 = -(2^4)^{-2} \times 2^1 \times (1 + \tfrac{1.0}{4}), \\
&\text{With ES=3: } 0_110_101_1 = +(2^8)^1 \times 2^5 \times (1 + \tfrac{1.0}{2}), \text{ and} \\
&10001111 \rightarrow 0_1110_001 = -(2^8)^2 \times 2^1 \times (1 + 0.0).
\end{aligned}$$

WHY POSIT?

- MAC Units Designs will differ in terms of Approximation, Architecture, Arithmetic & Algorithms.
- The selection of number representation scheme and the associated numeric computations play a significant role in efficient AI/ML hardware architecture design and the accuracy of machine learning inference.
- For DNN to have better inference accuracy, IEEE-754 floating-point representations are mostly preferred with a precision of 16-bit (half) or 32-bit (single) or bfloat16. However, the hardware design comes with higher resource utilization and more power consumption.
- For an efficient hardware design, fixed-point representation is used with Q4.4 or Q8.8 schemes.
- Recently, a new number system, i.e., Posit, has found a place with an ML inference accuracy level similar to floating-point but requiring fewer hardware resources like fixed-point arithmetic.

POSIT ADDER ALGORITHM (Currently working)

Algorithm 1 Regime, Exponent, Fraction Bits Extraction

Posit Size: N; Exponent Size: es
OutputTwos $[N-2:0]$ ▷ Input Size
Regime $[\log_2(N):0]$ ▷ Input Size
FractionBits $[N+1-es:0]$ ▷ Output Size
ExpBits$[es-1:0]$ ▷ Exponent Size
Atest ← InputA$[N-1]$
& ∼|InputA$[N-2:0]$ ▷ **Checking Special Case**
OutputTwos ← $\{[N]\{$InputA$[N-1]\}\}$
XOR InputA$[N-1:0]$ + InputA$[N-1]$ ▷ **2's Complement of Input**
InvertInput ← $\{[N-1]\{$OutputTwos $[N-2]\ \}\}$
XOR OutputTwos $[N-2:0]$
ZC ← Leading Zero Detector (InvertInput)
temp$[N-4:0]$ ← OutputTwos $[N-4:0] \ll (\text{ZC}-1)$
FractionBitsTemp ← $\{2'b01, \text{temp}, 3'b0\}$
ExpBits ← temp$[N-4:N-3-es]$
Regime ← OutputTwos $[N-1]$? $(\text{ZC}-1)$: –ZC
FractionBits ← ((InputA $[N-1]$ XOR InputB $[N-1]$) & InputA/B [MSB])
? –FractionBitsTemp : FractionBitsTemp

Algorithm 2 Calculating Scaling Factor

ShiftRegimeA ← RegimeA $\ll$ *es*
ShiftRegimeB ← RegimeB $\ll$ *es*
SFA ← ShiftRegimeA + ExpBitsA
SFB ← ShiftRegimeB + ExpBitsB
ValuetoShift ← |SFA – SFB|
if InputA> InputB **then**
    GreaterFractionBits ← FractionBitsA
    SmallerFractionBits ← FractionBitsB
    GreatestScalingFactor ← SFA
else
    GreaterFractionBits ← FractionBitsB
    SmallerFractionBits ← FractionBitsA
    GreatestScalingFactor ← SFB
end if

POSIT ADDER ALGORITHM (Currently working)

Algorithm 3 Adding Significand

SignificandAdd ← (GreaterFractionBits + (SmallerFractionBits ≫ ValuetoShift))
AnsIsZero ← ∼| SignificandAdd
AddOperation ← InputA$[N-1]$ OR InputB$[N-1]$

if SignificandAdd[MSB] = 0 & AddOperation=0/1 **then**
    ToNormalizedFraction ← SignificandAdd
    Shift = 0
else if SignificandAdd[MSB] =1 & AddOperation=0 **then**
    ToNormalizedFraction ← SignificandAdd ≫ 1
    Shift = 1
else if SignificandAdd[MSB] =1 & AddOperation=1 **then**
    ToNormalizedFraction ← –SignificandAdd
    ZC1 ←Leading Zero Detector(ToNormalizedFraction)
    NormalizedFraction ← ToNormalizedFraction ≪ ZC1
end if
Adjusted Scaling factor ← GreatestScalingFactor+Shift−ZC1

Algorithm 4 Decoding Regime and Exponent Value, Encoding and Rounding

Absolute SF ← Adjusted Scaling Factor [MSB] ? –Adjusted Scaling Factor : Adjusted Scaling Factor
if Adjusted Scaling Factor[MSB] = 0 **then**
    ExpBits ← Absolute SF $[es-1:0]$
    RegimeAns ← Absolute SF ≫ es
else
    ExpBits ← Adjusted Scaling Factor $[es-1:0]$
    RegimeAns1 ← Adjusted Scaling Factor ≫ es
    RegimeAns ← –RegimeAns1
end if
TempAns1 ← {2'b10,ExpBits, NormalizedFraction, {[maxregime]{1'b0 } } } ▷ **Packing Stage 1**
TempAns2 ← {2'b01,ExpBits, NormalizedFraction, {[maxregime]{1'b0 } }
if RegimeAns={log(N){1'b1}} **then**
    Shiftnegexp ← RegimeAns
else
    Shiftnegexp ← RegimeAns−1
end if
if Adjusted Scaling Factor[MSB] = 0 **then**
    TempAns ← TempAns1 ≫ RegimeAns
else
    TempAns ← TempAns2 ≫ Shiftnegexp
end if
Guard bit ← Extract from TempAns ▷ **Packing Stage 2**
Round ← Extract from TempAns
Sticky ← Extract from TempAns
LSB ← Extract from TempAns
checkround ← ((LSB & Guard) | (Guard & Round)) | (Guard & Round | Sticky)
IntermediateAns ← {1'b0,TempAns}+checkround
if AnsIsZero= 0 **then**
    FinalAns ← 0
else if take2sans=1 **then**
    FinalAns ← –IntermediateAns
else
    FinalAns ← IntermediateAns
end if
AdditionResult ← Special ? inf : FinalAns

NEXT STEPS

- Need to Understand the algorithm of Posit Multiplier
- Implementation of Posit adder & Posit multiplier using Verilog HDL
- To implement Posit Adder the following modules has to design -
- Top-module which takes N (posit word size) and es (posit exponent size), Posit data extract sub-module, Dynamic right shifter sub-module, Dynamic left shifter sub-module, Leading-One-Detector sub-module, Leading-Zero-Detector sub-module, Two N-bit integer adder sub-module, Two N-bit integer subtract sub-module, Integer Adder sub-module for mantissa overflow addition, Test-bench module.
- Similarly, Posit Multiplier implementation

REFERENCES

- Manish Kumar Jaiswal and Hayden K.-H. So, "Architecture Generator for Type-3 Unum Posit Adder/Subtractor", IEEE International Symposium on Circuits and Systems (ISCAS 2018), pp. 1-5, Florence, Italy, May 2018.
- Manish Kumar Jaiswal and Hayden K.-H. So, "Universal Number Posit Arithmetic Generator on FPGA", Design Automation and Test (DATE 2018), pp. 1159-1162, Dresden, Germany, Mar 2018.
- https://ieeexplore.ieee.org/document/10089906 - By Gopal Sir
- https://github.com/manish-kj/Posit-HDL-Arithmetic - For coding Reference

THANK YOU