Row Matrix multiplication with fixed point and parallelism in a FPGA

Emilio Tonix Gleason

21/Abril/2022

Contents

1 Introduction

Fixed point arithmetic is lighter than floating point and it is used when a processor doesn't have a **FPU** or on embbeded systems when resources are limited only depending on **CPU**. We call fixed point as **FXP**. Its notation is called Q format and it is described as following $QI.F$ where $I = integer$ bits and $F = fractional$ bits resolution i.e signed Q2.3
$\underbrace{1}_{sign}\underbrace{0}_{I}.\underbrace{010}_{F}$
Adding and multiplying will be the operations taken for this document due to matrix multiplication only used this two operations.

1.1 Additon

When we add two fixed numbers we shall align the fixed point arithmetic to make the operation.

Algorithm 1 Sum two numbers with diffent QI.F

```
Require: $QA.B \quad QC.D$
  $Q_{res}I \leftarrow max(A, C)$
  $Q_{res}F \leftarrow max(B, D)$
                                          ▷ Adjusts integer part only if needed
  if $A > C$ then
      Concatenate Left C Sign $A - C$ times
  else if $A < C$ then
      Concatenate Let A Sign $C - A$ times
  end if                                  ▷ Adjusts fractional part only if needed
  if $B > D$ then
      Data« $B - D$ times                 ▷ Right logic shift
  else if $B < D$ then
      Data« $D - B$ times
  end if
  Res = $QA.B + QC.D$
```

Other example. Calculate a+b, if a=10.11 and b=100.001 are two signed numbers, respectively, in Q2.2 and Q3.3 formats. Assume that the adder can process numbers in Q4.3 format.

$$\begin{array}{ccccccccccc} & 1 & 1 & 1 & 0 & . & 1 & 1 & & -1.25 \\ + & 1 & 1 & 0 & 0 & . & 0 & 0 & 1 & -3.875 \\ \hline 1 & 1 & 1 & 0 & 1 & . & 1 & 1 & 1 & -5.125 \end{array}$$

1.2 Multiplication

In this operation we need don't need a compensation of the point. Value grows as the sum of both integer and fractional bit size i.e Q2.2 x Q3.3 = Q5.5 . However the if the multiplication is signed we shall ignore last bit because sign is repeated.

Assume that A=101.001 and B=100.010 are two numbers in Q3.3 format. Assume that A is a signed number but B is unsigned. When we do partials sums we must do padding to the left part in ordert to preserve the sign.

$$\begin{array}{cccccccccccc}
 & & & & & & 1 & 0 & 1 & 0 & 0 & 1 \\
 & & & & & \text{x} & 1 & 0 & 0 & 0 & 1 & 0 \\
\hline
 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 \\
 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 0 & 1 & 0 & 0 & 1 & \\
 & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & & \\
 & \mathbf{0} & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & & & \\
 & \mathbf{0} & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
+ & 1 & 0 & 1 & 0 & 0 & 1 & & & & & \\
\hline
\mathbf{1} & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0
\end{array}$$

Finally data is res = 11001.110010

2 Data generation

2.1 Python data generation

A FXP database was created in python3 with the FXP library fxpmath, numpy, and pandas. Pandas is used for generate a database in a csv file.

```python
import numpy as np
from fxpmath import Fxp
import pandas as pd
```

Pandas data frame has 1000 samples, however size is 2000 because data is saved as one row for floating data and second row for FXP data. Data is saved in this way in order to reused data base in future for compare optimal value with HW calculated.

```python
df = pd.DataFrame(index=np.arange(REALIZATIONS*2), columns=np.arange(MATRIX_SIZE+ROW_SIZE+RESULT_SIZE))
```

We create random matrices and vectors to produce AB=C where A is the matrix and B is the row vector with a resulting vector C. Take acount that matrix dimensions could be reshaped to make easier its storage. If A -> 8x8 this could be storage as 1x64

0	Flatten Matrix[64]	Vector[8]	OptimalMul[8]
1	FXP Matrix[64]	FXP Vect[8]	Empty

We do a for loop of 1000 realizations to test the FPGA hardware in future.

```python
for n in range (0,REALIZATIONS):
    #Flatten matrix 2D in a 1D array
    rand_arry = np.random.uniform(-1.0,1,MATRIX_SIZE)
    #reshaping used to get optimial multiplication
    matrix    = np.reshape(rand_arry,(ROW_SIZE,ROW_SIZE))
    #Vector Generation
    vector    = np.random.uniform(-1.0,1,ROW_SIZE)
    #optimal value,
    #FXP will aproach to this value using only 16 bits
    f_result = matrix@vector.T

    #FXP generation truncation is done by the internal
    #library with the QIF required.
    z1 = Fxp(rand_arry, signed=True, dtype='Q2.14')
    z2 = Fxp(vector, signed=True, dtype='Q2.14')
```

For this project we are limited to used 16 bits for our data so we assume that all of the data is generated in signed Q2.14 because maximun integer range is from -1 to 1. **Note that Q2.14 has only on integer bit resolution because the other bit is the sign**.
Finally storage is done in the pandas data frame and saved in a csv.

```python
    #Save data in pandas data frame
    df.loc[n*2,0:63]    = rand_arry
    df.loc[n*2,64:71]   = vector
    df.loc[n*2,72:79]   = f_result

    for m in range(0,64):
        df.loc[n*2+1,m]  = z1[m].hex()
    for m in range(0,8):
        df.loc[n*2+1,m+64]  = z2[m].hex()

df.to_csv('data.csv')
```

3 Multiplication Algorithm

3.1 Matrix definition

Given a matrix $A_{m\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$ and vector $B_{nx1} = \begin{bmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{m1} \end{bmatrix}$ the product AB is equal vector $C_{mx1} = \begin{bmatrix} c_{11} \\ c_{21} \\ \vdots \\ c_{m1} \end{bmatrix}$ where $c_{11} = \sum_{i=1}^{m} a_{1i} \times b_{i1}$.

For this project we will deal with an square matrix where $m = n$. The product dim are C_{nx1} and first element is $c_{11} = \sum_{i=1}^{n} a_{1i} \times b_{i1}$. Notice that we could split in elementwise product $\odot$ and then apply the sumation the result. $\theta = A[0] \odot B^T$, $c_{11} = \sum_{i=1}^{n} \theta_i$.

3.2 Parallel multiplication with FXP

Columns for the matrix and rows from the vector are completly independent so we can parallelise the multiplication in hardware to get θ. In the case for $n = 8$, we need 8 parallel multpliers.

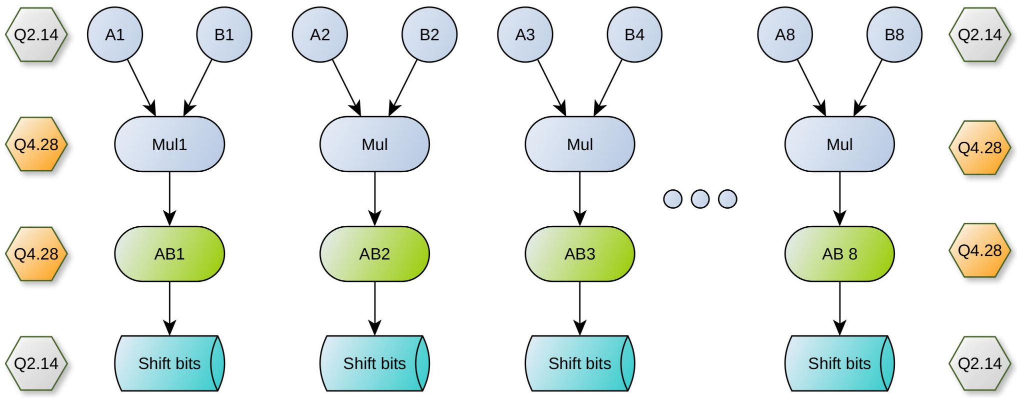

Figure 1: Parallel multiplier idea

As we are working with FXP Q2.14 data multiplication will expand data resoluton however we could only get the bits we are insterested in and keep the resolution of the original data. We could do this because maximum values of our data which is normalized is from -1 to -1 so any multiplication don't exceed the Q resolution. To get back the resolution we ignore the last fractional bits and the last integer bits including the redudant sign.

```
        AB[i]  = A[i]*B[i];
        C[i]   =(AB[i][(2*`WORD_SIZE)-3:14]);// discard last bit and ignore 16
```

3.3 Sumation part with divide and conquer

As multiplicaction may not cause us overflow or underflow sumation can. This mean that we may haven't enought integer part resolution so if ther is an overflow in the sumation we shall detect it and adjust the QI.F resolution keeping as maximum the 16 bits for resolution. To resolve this issue and do the sumation at the time there is a module called FXPcheck. Here is the algorithm of the FXP check which is implementend in verilog too.

Algorithm 2 SUM and FXP check

```
Require: $DataA \quad DataB \quad QI_in \quad QF_in$
  Tp = DataA+DataB

  SignA = DataA[len]
  SignB = DataB[len]
  SignTp = Tp[len]

  // Overflow only is possible if two values have the same sign
  $Overflow \leftarrow (SignA \oplus SignB)?0 : SignA \oplus SignTp$

  // if the integer sign bit is equal to the next bit
  // this means that bit n-2 is irrelevant for integer part
  $Underflow \leftarrow \ (SignTp \oplus Tp[len-2]) AND (QIin! = 1)$

  if Overflow then
      QF-1 , QI+1
      Res = signA,DataA[len-1:1]+signB,DataB[len:1]
  else if Underflow then
      QF+1 , QI-1
      Res = Tp[len-2:0],0;
  else
      Res = Tp;
  end if
```

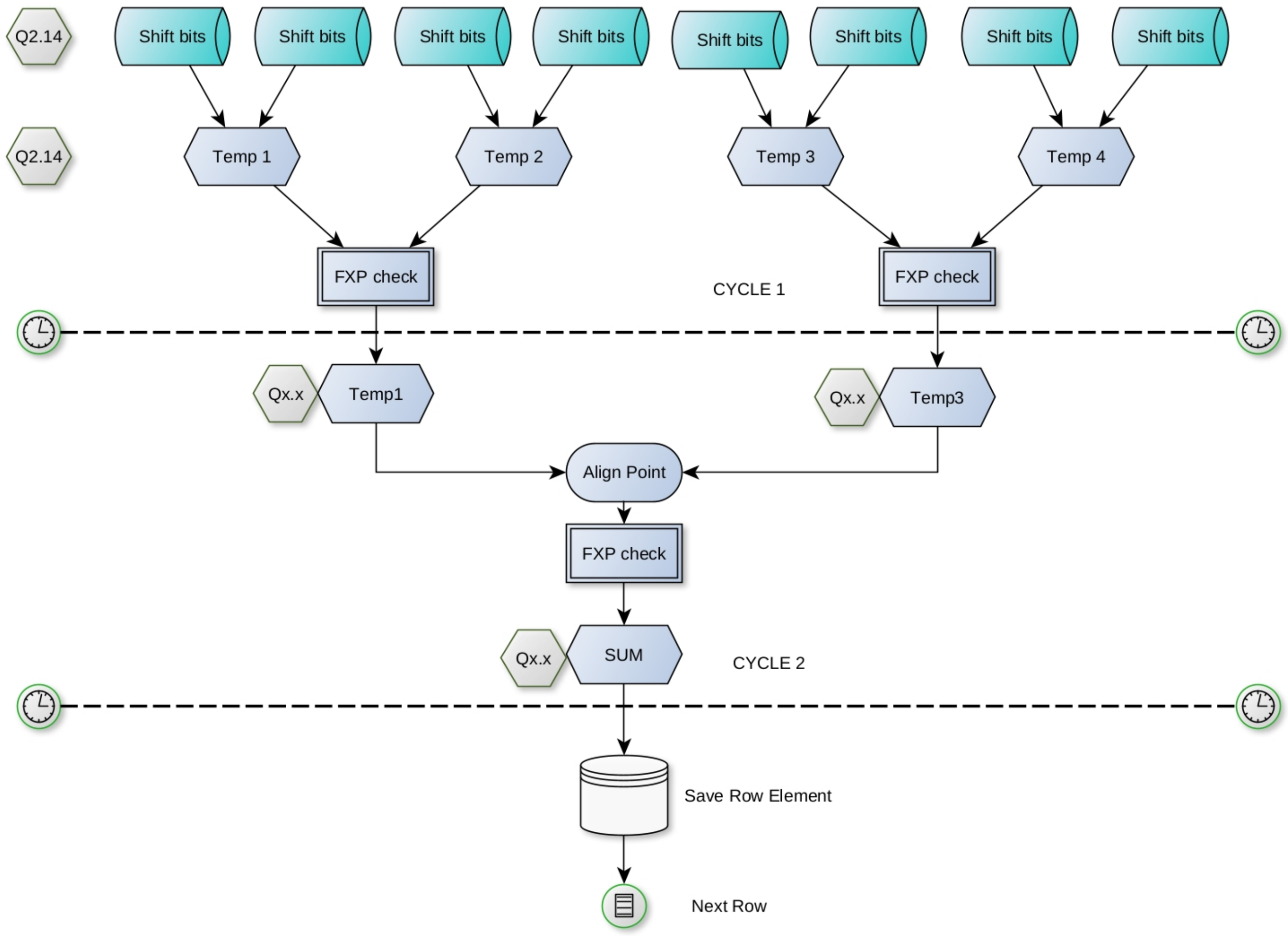

Figure 2: Sum staging architecture after muliplication

3.4 Performance

The FXP check is done 3 times after elementwise product. Instead of doing 8 secuencial sums that may take 8 CPU cycles per row product we only take 3 clock cycles for the multiplication and sum of all elements. Intead of doing 64 cycles for the whole matrix and vector mult we only take 24 cycles. Aproach of the algorithm is $\mathbf{O(\log(N))}$ and this is a good one because there is a balance between reducing execution time but also not too use much hardware.

4 Implementation

4.1 Insert data

Data is saved in a RAM module which has an important feature to reduce reading time. Writing data takes 64 cycles to reduce pin input footprint. However internally RAM data is read in parallel so the entire row is pass to the multiplication procedure. Read takes 3 clock cycles. One for setting the addres, second for clearing flags and send message to a FSM and finnaly for internaly storage read value.

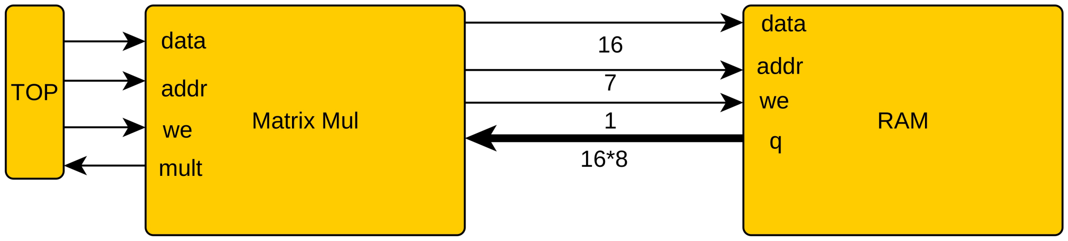

Figure 3: General Architecture

Parallel reading of the data

```
        {A[0],A[1],A[2],A[3],A[4],A[5],A[6],A[7]} = data_rd;
```

4.2 State Machine

Finally to get the resulting vector of the multplication we do a state machine for each row.

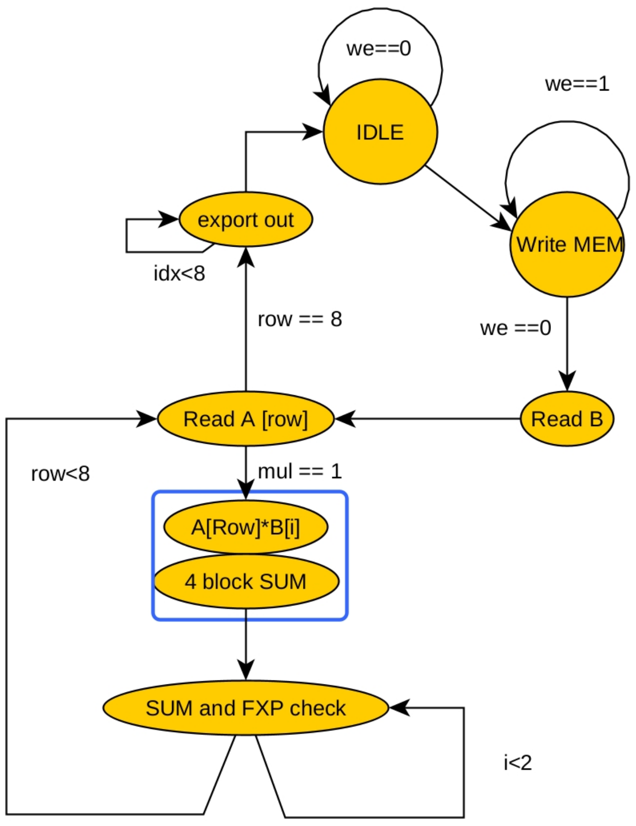

Figure 4: General state machine

All states except the write mem and export rows take 3 cycles. B is the vector and A is the row for the matrix read in parallel. We export rows of the vector in a serial fashion which takes 8 cycles in total.

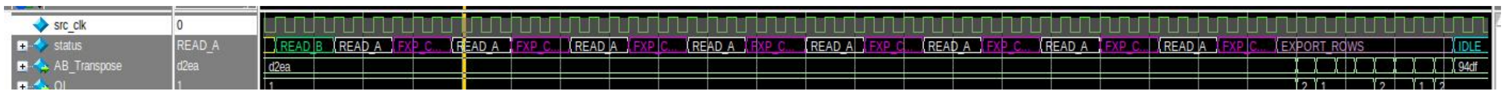

Figure 5: Complete matrix vector multiplication

5 Validation

5.1 Testbench export

To export the data given by the simulation I create an always sensitive to any change of AB_Transpose which is the resulting signal of the multiplicaton. The exported csv has one data value and its resulting QI and QF format.

```
    fd_wr = $fopen("../../PythonScripts/res.csv", "w");
    for(i=0;i<7;i=i+1) begin
        $fwrite(fd_wr,"D_%d ,QI_%d ,QF_%d ,",i,i,i);
    end
    $fwrite(fd_wr,"D_%d ,QI_%d ,QF_%d\n",i,i,i);
```

Vector number	**Data**	**QI**	**QF**	x8

Testbench signal result with the always listening to any change in signal. Data is expressed in hexadecimal format to make a shorter representation.

```
always @(AB_Transpose) begin
    if(hold_repeat == 0)
        if(data_cnt !=7)
        begin
            $fwrite(fd_wr,"%X ,%d ,%d ,",AB_Transpose,QI,QF);
            data_cnt = data_cnt+1;
        end
        else begin
            $fwrite(fd_wr,"%X ,%d ,%d \n",AB_Transpose,QI,QF);
            data_cnt = 0;
            hold_repeat = 1;
        end
    else begin
        hold_repeat = 0;
    end
```

5.2 Python read data and SQNR

The csv file is read in python as a pandas data frame and I take the data from the hex value and QI.F, with this data I called the FXP lib constructor and build my data generated by the simulation on the other hand I read also the data previously generated as the optimal.

```python
#ideal database generated with floating point
df_model = pd.read_csv('data.csv',index_col=False)
df_model = df_model.iloc[: , 1:] # remove row index
df_res   = pd.read_csv('res.csv')# FXP model generated by the simulation
opt = [] # data taken from fp data
estimate = [] # data taken from fxp simulation
```

Storage of the FXP data calling the constructor from data frame. Then when we finish to save all values in a vector we convert it into numpy. Numpy convertion will help us to calculate the power of the signal.

```python
for i in range(0,999):
```

```python
    #FXP
    fxp_vector = []
    #step by 3. HexData, QI, QF
    for n in range(0,24,3):
        #data camputured in the n column
        DataHex = "0x"+str(df_res.iloc[i][n])
        QI      = str(df_res.iloc[i][n+1])
        QF      = str(df_res.iloc[i][n+2])
        Qstr    = "Q"+QI+"."+QF # QI:F resolution format
        z1 = Fxp(DataHex, signed=True, dtype=Qstr)
        fxp_vector.append(z1)

    FPGA_res = np.asarray(fxp_vector)
```

Power of the signal is given by the following formula $\sum |x(n)|^2 \frac{1}{N}$ which can be rewrite in linear notation as $XX^T \frac{1}{N}$. In the code shown bellow is the power calculation.

```python
    opt.append(math.sqrt((optimal_res@optimal_res.T)/8))
    estimate.append(math.sqrt((FPGA_res@FPGA_res.T)/8))
```

Finally we create a vector of each data where we calculate the power and then we procced to average all samples and make a relation between them inside a logartmic function $SQNR = log(\frac{P_{estim}}{P_{ideal}})$

```python
opt_np = np.asarray(opt)
res_np = np.asarray(estimate)
P_opt  = np.average(opt_np)
P_res  = np.average(res_np)
SQNR   = 10*math.log(P_res/P_opt)
print("SQNR: ",end='')
print(SQNR,end='')
print(" dB")
```

Result
SQNR: 0.0010208068556479268 dB

6 Bibliography

https://github.com/francof2a/fxpmath#copy
https://www.allaboutcircuits.com/technical-articles/multiplication-examples-using-the-fixed-point-representation/
https://www.gaussianwaves.com/gaussianwaves/wp-content/uploads/2020/06/Power-and-Energy-of-a-signal-different-ways.png
https://handwiki.org/wiki/Hadamard_product_(matrices)