Politecnico di Torino

Block floating point for FPGAs

Master's degree in Electrical Engineering

Candidate: SHANCHENG LI

Supervisors: Prof. LUCIANO LAVAGNO
Prof. BRIGNONE GIOVANNI
Prof. MUHAMMAD USMAN JAMAL

July 2024

Abstract

Field-Programmable Gate Arrays (FPGAs) typically use fixed-point processors when performing floating-point operations, as this approach can effectively reduce hardware resource consumption and improve computational speed. FPGAs can also perform floating-point operations directly by using IP cores, but this method usually consumes more hardware resources and may reduce computational speed. Therefore, other algorithms can be used to optimize FPGA's computation of floating-point numbers, improving computational accuracy while reducing resource consumption. Block Floating Point (BFP) has a wide range of applications in FPGA design and gets a balance between resource consumption and accuracy. BFP is an algorithm used in digital signal processing. It is a method to optimize floating-point operations in computers, which can reduce memory and processor load while improving accuracy. This paper uses the BFP algorithm to write an operation library based on the C++ language, simulating and synthesis on Vitis HLS, performs related operations on the computation, such as initialization of integers, initialization of double-precision types, normalization, denormalization, overflow check, addition, subtraction, multiplication, division, dot product, etc. Before performing calculations, according to the theory of Block Floating Point (BFP), it is necessary to normalize the data being processed so that the data can obtain the same exponent. Before normalization, it is also necessary to calculate the headroom of each data point. After obtaining the normalized data, related operations can be performed. The key point is to judge the validity of the result, that is, to prevent possible overflow and underflow. Specifically, when performing dot product calculations, the operation library will first normalize the matrices uniformly, obtain the exponent of each individual matrix, and then perform the related matrix calculations.

Contents

1 Introduction

1.1 Block floating point

The relative advantages of fixed-point and floating-point implementations are well-established in computational systems. However, it is feasible to approximate the expansive dynamic range typically associated with floating-point arithmetic on fixed-point processors. This can be achieved through the implementation of floating-point emulation software. Such emulation, while effective, is computationally expensive, as it necessitates the manipulation of all arithmetic operations to simulate floating-point mathematics on a fixed-point architecture. The utilization of this software emulation is only justifiable when a minor fraction of the overall computational load requires an extended dynamic range. Given these constraints, there is a clear need for a more efficient method to achieve floating-point-like dynamic range on fixed-point processors. This approach should offer a balance between performance and the expanded range capabilities, providing a cost-effective alternative to full software emulation. The development of such a solution would bridge the gap between the limitations of fixed-point systems and the requirements of applications demanding higher precision and broader dynamic range. The block floating point algorithm is an extension of the block automatic gain control (AGC) concept. While block AGC is limited to scaling values at the input stage of the Fast Fourier Transform (FFT) and solely adjusts input signal power, the block floating point algorithm offers a more comprehensive approach. It monitors signal strength across multiple stages, thereby providing a more complex scaling strategy and enhanced dynamic range. This discussion focuses on the block floating-point algorithm as a method of floating-point emulation. The algorithm's primary advantage stems from its block-based operational approach, utilizing a shared exponent. In this system, each value within a block is represented by two components: a mantissa and a common exponent. The common exponent is stored as a separate data word, which results in a more efficient hardware implementation compared to conventional floating-point architectures. This method offers a balance between the precision of floating-point arithmetic and the efficiency of fixed-point processing. By sharing an exponent across a block of values, it reduces the storage and computational overhead associated with individual exponents, while still providing a significant increase in dynamic range compared to pure fixed-point systems. This approach is particularly beneficial in applications requiring extended dynamic range without the full complexity of traditional floating-point hardware. The common exponent

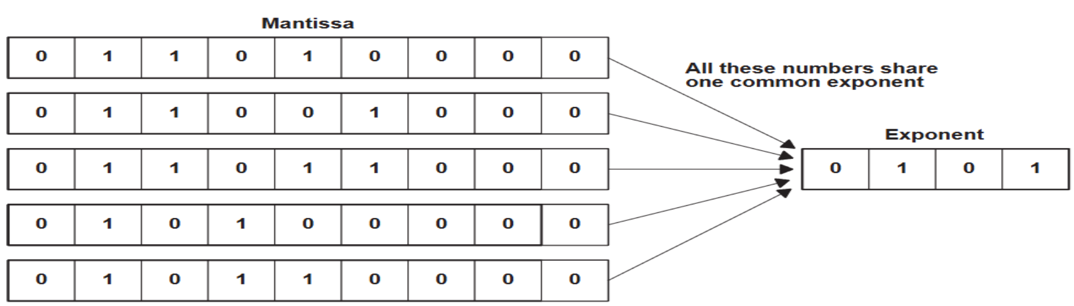


Figure 1: Diagram of Block Floating-Point Representation

in the block floating-point algorithm is determined by the data element with the greatest magnitude within the block. To calculate this exponent, it is necessary to confirm the number of leading bits, which is equivalent to the number of left shifts required to normalize the largest data element to the processor's dynamic range. Some Digital Signal Processing (DSP) processors are equipped with specialized instructions, such as exponent detection and normalization, to facilitate this process efficiently. The algorithm's flexibility is evident in its handling of diverse data scenarios. For blocks comprising predominantly small values, a large common exponent can be employed to shift these values leftward, thereby expanding the effective dynamic range. Conversely, for blocks containing larger values, a smaller common exponent is applied. Regardless of the scenario, once the common exponent is determined, all data elements in the block undergo a uniform left shift by this amount, optimizing the utilization of the available dynamic range. The block floating-point representation offers distinct advantages over both fixed-point and conventional floating-point formats. By scaling each value up by the common exponent, it enhances the dynamic range of data elements compared to a fixed-point implementation. Simultaneously, the use of a single common exponent for all data values within a block preserves the precision characteristic of fixed-point processors. This dual benefit renders the block floating-point algorithm more economically viable than traditional floating-point implementations.

This approach strikes a balance between the extended dynamic range typically associated with floating-point arithmetic and the computational efficiency of fixed-point processing. It provides a cost-effective solution for applications requiring increased dynamic range without the full overhead of conventional floating-point hardware, making it particularly suitable for certain DSP applications where both precision and range are critical.

1.2 Code operations

receive_int: receive integer.
receive_double: receive double.
bfp_init: nomalize.
bfp_headroom: save the result from headroom calculation operation to block floating point
headroom_calc: calculate the headroom with different modes. Because of different result data widths, mode 0 is for normal mode, mode 1 is for addition and subtraction operation, mode 2 is for multiply, and mode 3 is for generating dot_product.
check_overflow: check whether the data width is below the set value.
denomalize: used by addition and subtraction operations.
renomalize: renomalize block floating point.
renomalize_dot_product: same as bfp_init but with a different structure this operation is for dot production.
add: addition operation.
subtract: subtraction operation
multiply: multiply operation.
dot_product: dot production operation
to_double_array: transfer block floating point to floating point.

2 Simulations with base calulations

2.1 ADD

2.1.1 Code implements

Listing 1: ADD Code

```
static bfp_t add(const bfp_t& a, const bfp_t& b) {
                unsigned i;
                exp_t exp_val,interm_exp,a_shr,b_shr;
                bfp_t c,d,result,result_temp;
            ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
                 interm[num_elements];
                //ap_fixed<W+1,I+1> interm[num_elements];
```

```
            result.length = num_elements;
             if(a.exp<b.exp){//new bfp's shared exp is equal to max(a.exp,b.exp)+1
                 interm_exp=b.exp+1;
             }
             else if(a.exp>b.exp){
                 interm_exp=a.exp+1;
             }
             else{
                 interm_exp=0;
             }
             a_shr=interm_exp-a.exp;//get the shift right number for denomalizing
             b_shr=interm_exp-b.exp;
            denomalize(c,a,a_shr);//return c,d with same exp, so they could be added together
            denomalize(d,b,b_shr);
             for(i = 0; i < num_elements; ++i) {
              interm[i] = c.data[i] + d.data[i];
             }
             exp_val=check_overflow(1,interm);//check ovf if it needs to renomalize
             if(exp_val==0){//no ovf
                 for(i = 0; i < num_elements; ++i) {
                  result_temp.data[i] =interm[i];//send data to result bfp
                 }
                    renomalize(result_temp);//find the largest number to comfirm the exp, and make sure the largest number' MSB is 1 behind the sign bit
                    result.exp=result_temp.exp+interm_exp;
                 for(i = 0; i < num_elements; ++i) {
                      result.data[i] =result_temp.data[i];
             }
             }
             else{//ovf, but it could only be 1 more bit than the original bit width in add and sub operations
              for(i = 0; i < num_elements; ++i) {
                  result.data[i] =interm[i]>>exp_val;
                  result.exp=1+interm_exp;
                 }
             }
            result.length = num_elements;
            return result;
    }

    friend bfp_t operator+(const bfp_t& a, const bfp_t& b) {
        return add(a, b);
    }
```

2.1.2 Testbench codes

Listing 2: ADD Testbench

```
std::cout << "ADD␣result␣C␣computered␣by␣bfp:\n";
bfp.to_double_array_1(result,c_from_bfp);
for (int i = 0; i < ARRAY_SIZE; ++i) {
    std::cout << std::setw(10) << std::fixed <<c_from_bfp[i] << "␣";
}
std::cout << std::endl;
```

```
std::cout << "different␣between␣fp␣and␣bfp:\n";
for (int i = 0; i < MATRIX_SIZE; ++i) {
    diff_add[i]=c_from_bfp[i]-c[i];
    if(abs(diff_add[i])>0.0001){
    std::cout << std::setw(10) << std::fixed <<diff_add[i] << "␣";
    }
}
```

2.1.3 Comparison of results

ADD result C computed by fp	ADD result C computed by bfp	different between fp and bfp
120.322275	120.322250	0.000025
11.230811	11.230789	0.000022
118.906217	118.906189	0.000028
98.220771	98.220749	0.000022
-77.425459	-77.425476	0.000018
-45.442061	-45.442078	0.000017
-32.514420	-32.514435	0.000015
-69.948424	-69.948441	0.000017
-145.524461	-145.524490	0.000030
-3.808710	-3.808731	0.000021
-28.009888	-28.009903	0.000015
4.089480	4.089462	0.000018
-57.667776	-57.667786	0.000010
-115.286721	-115.286728	0.000007
4.040651	4.040634	0.000016
153.324992	153.324982	0.000010
-56.331065	-56.331085	0.000020
178.240303	178.240295	0.000007
-102.645955	-102.645966	0.000011
152.562029	152.562012	0.000017
106.039613	106.039597	0.000016
17.835017	17.834991	0.000025
-22.724082	-22.724106	0.000024
-20.099490	-20.099503	0.000012
32.929472	32.929459	0.000013
...		
-14.825892	-14.825897	0.000005
-108.645894	-108.645905	0.000011
85.048982	85.048965	0.000017
-111.825922	-111.825928	0.000005
-36.530656	-36.530670	0.000014
-28.363903	-28.363922	0.000019
-19.214454	-19.214462	0.000009
45.185705	45.185699	0.000006
-1.910459	-1.910477	0.000018
36.024049	36.024033	0.000016
27.411725	27.411713	0.000013

Max difference: 0.000030
Average difference: 0.000016

2.2 SUB

2.2.1 Code implements

Listing 3: SUB Codes

```
static bfp_t subtract(const bfp_t& a, const bfp_t& b) {
      unsigned i;
      exp_t exp_val,interm_exp,a_shr,b_shr;
      bfp_t c,d,result,result_temp;
      ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
          interm[num_elements];
      //ap_fixed<W+1,I+1> interm[num_elements];
       if(a.exp<b.exp){
           interm_exp=b.exp+1;
       }
       else if(a.exp>b.exp){
           interm_exp=a.exp+1;
       }
       else{
           interm_exp=0;
       }
       a_shr=interm_exp-a.exp;
       b_shr=interm_exp-b.exp;
          denomalize(c,a,a_shr);
          denomalize(d,b,b_shr);
       for(i = 0; i < num_elements; ++i) {
        interm[i] = c.data[i] - d.data[i];// different from add here
       }
       exp_val=check_overflow(1,interm);
       if(exp_val==0){
           for(i = 0; i < num_elements; ++i) {
            result_temp.data[i] =interm[i];
           }
              renomalize(result_temp);
              result.exp=result_temp.exp+interm_exp;
           for(i = 0; i < num_elements; ++i) {
                result.data[i] =result_temp.data[i];
       }
       }
       else{
        for(i = 0; i < num_elements; ++i) {
            result.data[i] =interm[i]>>exp_val;
            result.exp=1+interm_exp;
           }
       }
    result.length = num_elements;
    return result;
   }

   friend bfp_t operator-(const bfp_t& a, const bfp_t& b) {
      return subtract(a, b);
   }
```

2.2.2 Testbench codes

Listing 4: SUB Testbench

```
std::cout << "SUB␣result␣C␣computered␣by␣fp:\n";
for (int i = 0; i < ARRAY_SIZE; ++i) {
    c[i]=a[i]-b[i];
    std::cout << std::setw(10) << std::fixed <<c[i] << "␣";
}
std::cout << std::endl;
result=A_double-B_double;
std::cout << "SUB␣result␣C␣computered␣by␣bfp:\n";
bfp.to_double_array_1(result,c_from_bfp);
for (int i = 0; i < ARRAY_SIZE; ++i) {
    std::cout << std::setw(10) << std::fixed <<c_from_bfp[i] << "␣";
}
std::cout << std::endl;
std::cout << "different␣between␣fp␣and␣bfp:\n";
for (int i = 0; i < MATRIX_SIZE; ++i) {
    diff_add[i]=c_from_bfp[i]-c[i];
    if(abs(diff_add[i])>0.0001){
    std::cout << std::setw(10) << std::fixed <<diff_add[i] << "␣";
    }
}
```

2.2.3 Comparison of results

SUB result C computed by fp	SUB result C computed by bfp	different between fp and bfp
32.642598	32.642593	0.000004
1.324503	1.324509	0.000005
66.090884	66.090881	0.000003
88.241218	88.241226	0.000008
-14.294870	-14.294861	0.000009
-38.282418	-38.282410	0.000009
-52.027955	-52.027954	0.000001
-112.240974	-112.240982	0.000008
48.518326	48.518326	0.000001
-1.495407	-1.495407	0.000000
133.829768	133.829758	0.000010
125.888852	125.888840	0.000012
-106.076235	-106.076233	0.000002
-57.686087	-57.686081	0.000006
184.130375	184.130386	0.000011
-35.798212	-35.798218	0.000006
-91.775262	-91.775269	0.000007
21.027253	21.027252	0.000001
55.055391	55.055389	0.000002
14.612262	14.612274	0.000012
14.343699	14.343704	0.000005
-9.942930	-9.942932	0.000002
...		
-11.713004	-11.713013	0.000009
-35.663930	-35.663925	0.000005
55.946532	55.946533	0.000002
-24.268319	-24.268311	0.000008
70.827357	70.827362	0.000005
-92.684713	-92.684708	0.000005
-22.913297	-22.913300	0.000003
-3.576769	-3.576767	0.000002
153.245643	153.245651	0.000008
135.953856	135.953857	0.000001

Max difference: 0.000014

Average difference: 0.000006

2.3 MUL

2.3.1 Code implements

Listing 5: MUL

```
static bfp_t multiply(const bfp_t& a, const bfp_t& b) {
        unsigned i;
        exp_t exp_val,interm_exp;
        bfp_t c,d,result;
```

```
        ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
            interm[num_elements];
        //ap_fixed<2*W,2*I> interm[num_elements];
        interm_exp=a.exp+b.exp;//new shared exp
        for (i = 0; i < num_elements; ++i) {
            interm[i] = a.data[i] * b.data[i];
        }

         exp_val=check_overflow(2,interm);
         if(exp_val==0){
          for(i = 0; i < num_elements; ++i) {
           result.data[i] =interm[i];
          }
             renomalize(result);
             result.exp=result.exp+interm_exp;

      }
      else{
       for(i = 0; i < num_elements; ++i) {
         result.data[i] =interm[i]>>exp_val;
          }
         result.exp=interm_exp+exp_val;
      }
         result.length = num_elements;
        return result;

    }
  friend bfp_t operator*(const bfp_t& a, const bfp_t& b) {
        return multiply(a, b);
    }
```

2.3.2 Testbench codes

The testbench calculations for multiplication and dot product have been placed in the matrix simulation computation, where you can view the relevant matrix content.

2.4 DOT-MUL

2.4.1 Code implements

Listing 6: DOT-MUL

```
static bfp_dot_product_t dot_product(const bfp_t& a, const bfp_t& b) {
      unsigned i;
      bfp_dot_product_t result;
      exp_t exp_val=0,interm_exp;
      bfp_t c,d;
      //constexpr unsigned int I_test_value_dot = compute_ITest();
      ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> interm[1];
      //ap_fixed<2*W+I_test_value,2*I+I_test_value> interm_test=0;
      interm_exp=a.exp+b.exp;
      for (i = 0; i < num_elements; ++i) {
         interm[0] += a.data[i] * b.data[i];
         //interm_test=interm>>16;
         //std::cout << interm_test << " ";
      }
      exp_val=check_overflow(3,interm);
      if(exp_val==0){
```

```
            result.data =interm[0];
            renomalize_dot_product(result);
            result.exp=result.exp+interm_exp;
        }
        else{
              result.data =interm[0]>>exp_val;
              result.exp=interm_exp+exp_val;
        }
          return result;
        }
```

2.4.2 Testbench codes

The testbench calculations for multiplication and dot product have been placed in the matrix simulation computation, where you can view the relevant matrix content.

3 Simulations with special operations

3.1 Matrix-multiply

3.1.1 Testbench codes

Listing 7: MatriTx-multiply Testbench

```
int main() {
  std::srand(static_cast<unsigned int>(std::time(nullptr)));

  double Amatrix[MATRIX_SIZE][MATRIX_SIZE];
    double Bmatrix[MATRIX_SIZE][MATRIX_SIZE];

    generateRandomMatrix(Amatrix);
    generateRandomMatrix(Bmatrix);
    std::cout << "\nAmatrix:" << std::endl;
    printMatrix(Amatrix);

    std::cout << "\nBmatrix:" << std::endl;
    printMatrix(Bmatrix);
    double input_A[MATRIX_SIZE], input_B[MATRIX_SIZE];
double Cmatrix[MATRIX_SIZE][MATRIX_SIZE];
    BfpMatrix::bfp_t bfpa;
    BfpMatrix::bfp_t bfpb;
    BfpMatrix::bfp_dot_product_t C_matrix_data;
    double array[MATRIX_SIZE][MATRIX_SIZE]={0};
    double result_array=0;
    BfpMatrix bfp_matrix_multiply;
    double input_A[MATRIX_SIZE]={0}, input_B[MATRIX_SIZE]={0};
    for(int i = 0; i < MATRIX_SIZE; ++i) {
        for(int k = 0; k < MATRIX_SIZE; ++k) {
            input_A[k]= Amatrix[i][k];
        }
        for(int j = 0; j < MATRIX_SIZE; ++j) {
            for(int k = 0; k < MATRIX_SIZE; ++k) {
                input_B[k] = Bmatrix[k][j];
            }
            BfpMatrix::bfp_t bfpa=init(input_A);
            BfpMatrix::bfp_t bfpb=init(input_B);
            C_matrix_data=matrix_multiply(bfpa,bfpb);
```

```
            //std::cout << "result_array1111: " << result_array<< std::endl;
            bfp_matrix_multiply.to_double_array(C_matrix_data.data,C_matrix_data.exp,
                result_array);
            array[i][j]=result_array;
        }
```

3.1.2 Comparison of results

result computed by fp	result computed by bfp	different between fp and bfp
-666.020287	-666.021628	0.001342
-844.510901	-844.509803	0.001098
-7601.749292	-7601.749722	0.000429
5466.590415	5466.590553	0.000139
-10451.839316	-10451.837296	0.002020
-1920.304333	-1920.305948	0.001615
2807.310007	2807.311750	0.001744
5534.871628	5534.870266	0.001362
2814.237011	2814.236042	0.000969
-5758.867764	-5758.867874	0.000110
-6487.584146	-6487.587711	0.003566
-70.393144	-70.393507	0.000363
13115.717777	13115.716293	0.001483
1110.525810	1110.526584	0.000774
10602.955273	10602.955276	0.000003
-115.260597	-115.264430	0.003833
1020.595892	1020.595206	0.000686
639.677915	639.677373	0.000542
-7083.496123	-7083.495144	0.000979
10663.290535	10663.290321	0.000214
-3446.257972	-3446.260942	0.002971
3835.628962	3835.629539	0.000577
2821.907032	2821.906139	0.000893
-4982.870782	-4982.870094	0.000688
5512.296034	5512.295147	0.000887

Max difference: 0.003833

Average difference: 0.001171

3.2 FIR

3.2.1 Testbench codes

Listing 8: Global setting

```
    double coeffs[4] = {0.1, 0.2, 0.3, 0.4};
    double output1[10];
    max_diff = 0.0;
    sum_diff = 0.0;
    BfpFir::bfp_dot_product_t output_data;
    BfpFir::bfp_t receive_h_data,receive_x_data,bfp_h[10],bfp_x[10];
BfpFir fir_bfp;
```

```
    for (int i = 0; i < 10; ++i) {
        int index = 0;
        for (int j = 0; j < 4; ++j) {
            if (i - j >= 0) {
                receive_h[index] = coeffs[j];
                receive_x[index] = input[i - j];
                //std::cout << "receive_h[index]: " << receive_h[index] <<
                    std::endl;
                //std::cout << "receive_x[index]: " << receive_x[index] <<
                    std::endl;
                index++;
            }
        }
        BfpFir::bfp_t bfpa=init_fir(receive_h);
        BfpFir::bfp_t bfpb=init_fir(receive_x);

        output_data=FIR(bfpa,bfpb);
        fir_bfp.to_double_array(output_data.data,output_data.exp, output0);
        output1[i] = output0;

    }
```

3.2.2 Comparison of results

result computed by fp	result computed by bfp	different between fp and bfp
0.099609	0.100000	0.000391
0.408554	0.400000	0.008554
1.017273	1.000000	0.017273
2.035522	2.000000	0.035522
3.034546	3.000000	0.034546
4.033447	4.000000	0.033447
5.032471	5.000000	0.032471
6.070068	6.000000	0.070068
7.069092	7.000000	0.069092
8.067871	8.000000	0.067871

Max difference: 0.070068

Average difference: 0.036924

4 Syntheise with matrix multiply and FIR

4.1 Global setting

Listing 9: Global setting

```

#define MATRIX_SIZE 7
#define ARRAY_SIZE 50

using BfpMatrix = bfp_fpga<MATRIX_SIZE, 16, 32, 16>;
using BfpFir = bfp_fpga<4,5,13,5>;
```

4.2 Matrix-multiply

4.2.1 Codes

Listing 10: Matrix-multiply

```

BfpMatrix::bfp_t init(double a[]) {
    BfpMatrix bfp_matrix_multiply;
    BfpMatrix::bfp_t A_matrix_data,bfp_a;
    bfp_matrix_multiply.receive_double(A_matrix_data, a);
    bfp_matrix_multiply.bfp_init(bfp_a, A_matrix_data);
    return bfp_a;
}
BfpMatrix::bfp_dot_product_t matrix_multiply(BfpMatrix::bfp_t
   bfp_a,BfpMatrix::bfp_t bfp_b){
    BfpMatrix::bfp_dot_product_t C_matrix_data;
    BfpMatrix bfp_matrix_multiply;
    C_matrix_data = bfp_matrix_multiply.dot_product(bfp_a,bfp_b);
    return C_matrix_data;
    //bfp_matrix_multiply.to_double_array(C_matrix_data.data,C_matrix_data.exp,
       array);
}
```

4.3 FIR

4.3.1 Codes

Listing 11: FIR

```
BfpFir::bfp_t init_fir(double a[]) {
    BfpFir bfp_fir;
    BfpFir::bfp_t A_fir_data,bfp_a;
    bfp_fir.receive_double(A_fir_data, a);
    bfp_fir.bfp_init(bfp_a, A_fir_data);
    return bfp_a;
}
BfpFir::bfp_dot_product_t FIR(BfpFir::bfp_t bfp_a,BfpFir::bfp_t bfp_b) {
    BfpFir fir_bfp;
    BfpFir::bfp_dot_product_t output_data;
    output_data =fir_bfp.dot_product(bfp_a,bfp_b);
    return output_data;
    //fir_bfp.to_double_array(output_data.data,output_data.exp, array);
}
```

4.4 Synthesis summary

The comparison of results from figure2 and figure3 reveals that the resource consumption of matrix multiplication, calculated by the BFP and FP kernels respectively, shows that ”Latency”,”Interval”,”DSP” and ”FF” of using the BFP kernel is obviously less than FP. However, the difference in the number of LUTs is not very large, because the loops 116 and 108 in the BFP code are used to calculate the leading zeros, and the bfp accepts arbitrary bit width data, if only the standard data bit width, such as 16 bits, 32 bits, 64 bits, etc., can simplify the code to avoid using while loops and reduce resource usage. Similarly, the comparison of figure4 and figure5 yields the same result.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matrix_multiply				-	-	-	-	-	-	no	0	3	1372	2259	0
dot_product				-	-	-	-	-	-	no	0	3	885	1992	0
dot_product_Pipeline_VITIS_LOOP_335_1				-	9	90.000	-	9	-	no	0	3	73	150	0
VITIS_LOOP_335_1				-	7	70.000	2	1	7	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_116_3				-	-	-	-	-	-	no	0	0	102	97	0
VITIS_LOOP_116_3				-	-	-	1	1	-	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_108_2				-	-	-	-	-	-	no	0	0	101	97	0
VITIS_LOOP_108_2				-	-	-	1	1	-	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_116_32				-	-	-	-	-	-	no	0	0	102	97	0
VITIS_LOOP_116_3				-	-	-	1	1	-	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_108_21				-	-	-	-	-	-	no	0	0	83	97	0
VITIS_LOOP_108_2				-	-	-	1	1	-	yes	-	-	-	-	-

Figure 2: martix mul by bfp

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matrix_multiply_fp				-	35	350.000	-	36	-	no	0	22	2675	2272	0
dot_product				-	35	350.000	-	4	-	yes	0	22	2639	2107	0

Figure 3: martix mul by fp

Performance & Resource Estimates

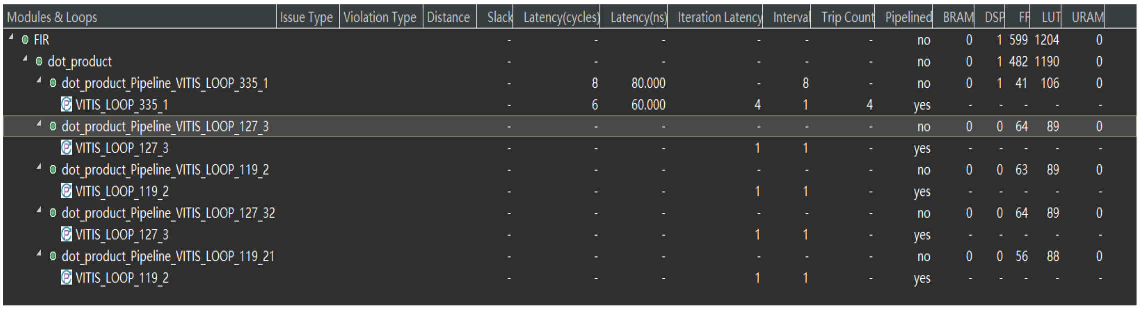
Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
fir_filter				-	23	230.000	-	24	-	no	0	11	1635	1215	0
dot_product				-	23	230.000	-	4	-	yes	0	11	1611	1096	0

Figure 5: fir by fp

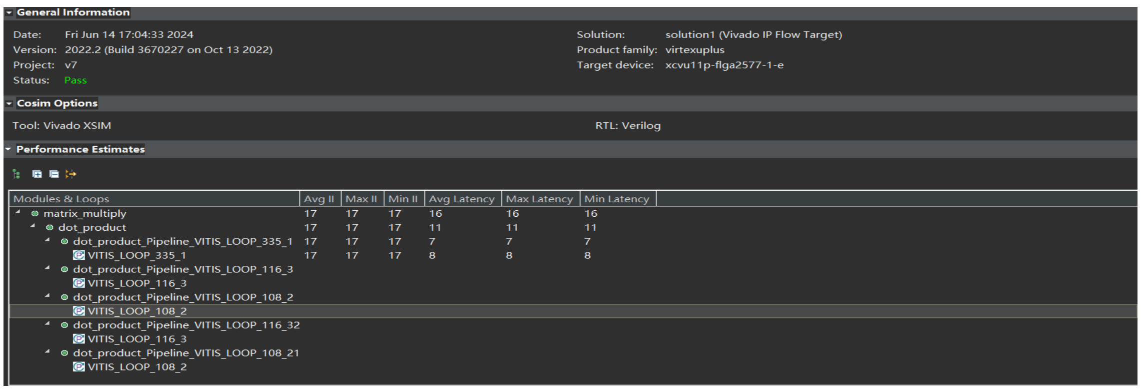
5 Cosimulation

The results show that II(Iteration Interval) and latency of BFP almost half less than FP.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
FIR				-	-	-	-	-	-	no	0	1	599	1204	0
dot_product				-	-	-	-	-	-	no	0	1	482	1190	0
dot_product_Pipeline_VITIS_LOOP_335_1				-	8	80.000	-	8	-	no	0	1	41	106	0
VITIS_LOOP_335_1				-	6	60.000	4	1	4	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_127_3				-	-	-	-	-	-	no	0	0	64	89	0
VITIS_LOOP_127_3				-	-	-	1	1	-	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_119_2				-	-	-	-	-	-	no	0	0	63	89	0
VITIS_LOOP_119_2				-	-	-	1	1	-	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_127_32				-	-	-	-	-	-	no	0	0	64	89	0
VITIS_LOOP_127_3				-	-	-	1	1	-	yes	-	-	-	-	-
dot_product_Pipeline_VITIS_LOOP_119_21				-	-	-	-	-	-	no	0	0	56	88	0
VITIS_LOOP_119_2				-	-	-	1	1	-	yes	-	-	-	-	-

Figure 4: fir by bfp



General Information

Date: Fri Jun 14 17:04:33 2024
Version: 2022.2 (Build 3670227 on Oct 13 2022)
Project: v7
Status: Pass

Solution: solution1 (Vivado IP Flow Target)
Product family: virtexuplus
Target device: xcvu11p-flga2577-1-e

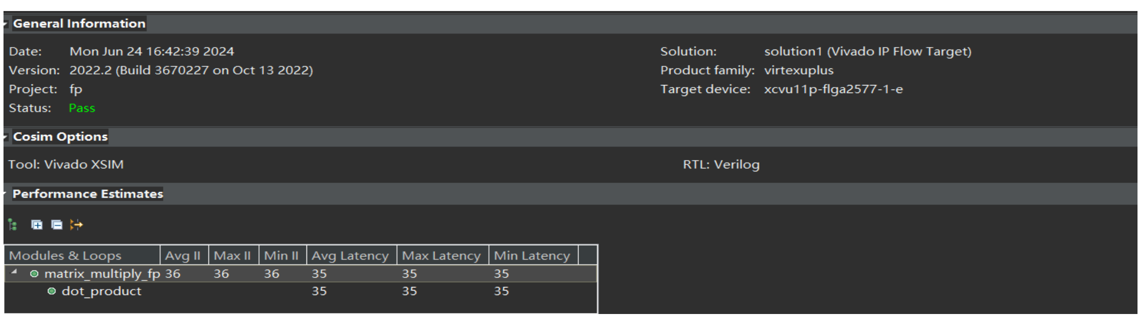
Cosim Options

Tool: Vivado XSIM
RTL: Verilog

Performance Estimates

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
matrix_multiply	17	17	17	16	16	16
dot_product	17	17	17	11	11	11
dot_product_Pipeline_VITIS_LOOP_335_1	17	17	17	7	7	7
VITIS_LOOP_335_1	17	17	17	8	8	8
dot_product_Pipeline_VITIS_LOOP_116_3						
VITIS_LOOP_116_3						
dot_product_Pipeline_VITIS_LOOP_108_2						
VITIS_LOOP_108_2						
dot_product_Pipeline_VITIS_LOOP_116_32						
VITIS_LOOP_116_3						
dot_product_Pipeline_VITIS_LOOP_108_21						
VITIS_LOOP_108_2						

Figure 6: matrix mul by bfp cosimulation result



General Information

Date: Mon Jun 24 16:42:39 2024
Version: 2022.2 (Build 3670227 on Oct 13 2022)
Project: fp
Status: Pass

Solution: solution1 (Vivado IP Flow Target)
Product family: virtexuplus
Target device: xcvu11p-flga2577-1-e

Cosim Options

Tool: Vivado XSIM
RTL: Verilog

Performance Estimates

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
matrix_multiply_fp	36	36	36	35	35	35
dot_product				35	35	35

Figure 7: matrix mul by fp cosimulation result

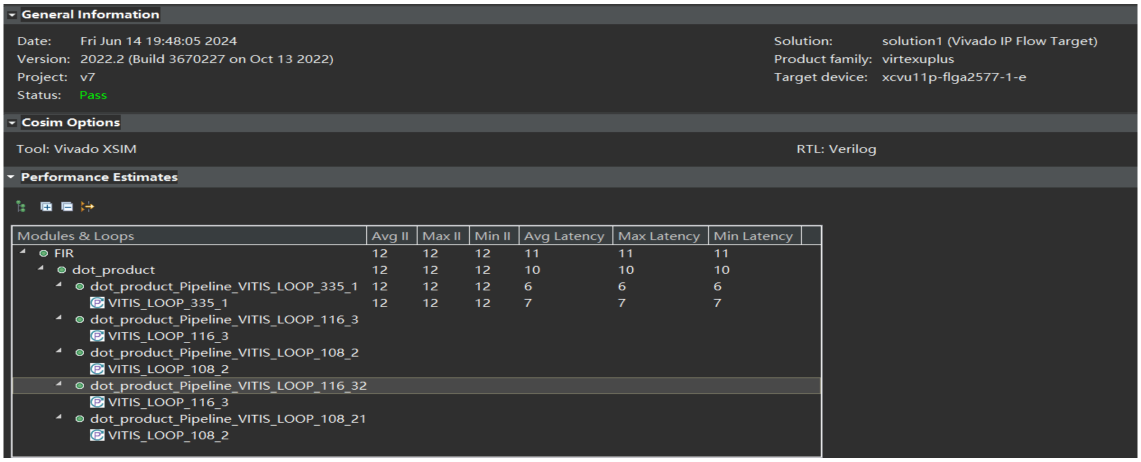


Figure 8: fir by bfp cosimulation result

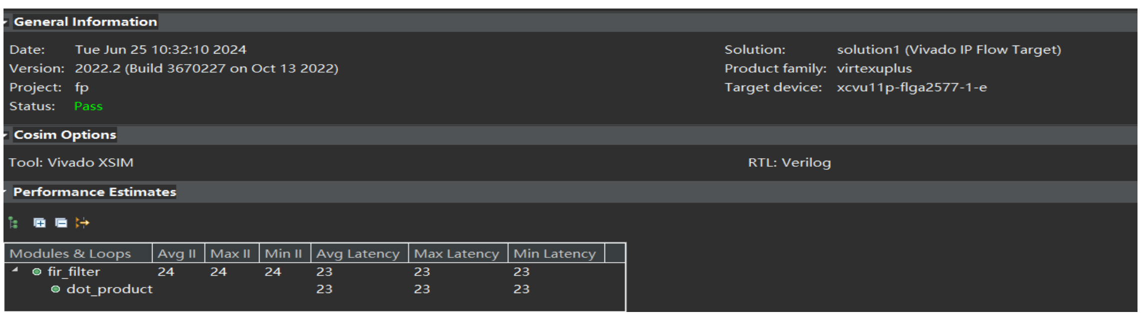


Figure 9: fir by fp cosimulation result

6 APPENDIX A

Listing 12: bfp_fpga.cpp

```
#pragma once
#include <hls_math.h>
#include <ap_int.h>
#include <ap_fixed.h>
//#include "matrix_multiply.h"


template <unsigned num_elements, unsigned exp_t_width, unsigned W,unsigned I>
class bfp_fpga {
private:
public:
    static constexpr unsigned custom_log2(unsigned n) {
        int log = 0;
        while (n >>= 1) ++log;
        return log;
    }
```

```
    static constexpr unsigned I_test_value_dot = custom_log2(num_elements)+1;
    using mantissa_t = ap_fixed<W,I>;
    using exp_t = ap_int<exp_t_width>;
    using headroom_t = ap_int<exp_t_width>;



struct bfp_t {
        mantissa_t  data[num_elements];
        exp_t exp;
        headroom_t hr;
        unsigned length;
        bfp_t() : exp(0), hr(0), length(0){}
        //LHS and RHS by using []
        mantissa_t& operator[](unsigned index) {
            return data[index];
        }

        const mantissa_t& operator[](unsigned index) const {
            return data[index];
        }
        //copy constructor
        bfp_t(const bfp_t& other)
            : exp(other.exp), hr(other.hr), length(other.length) {
            for (unsigned i = 0; i < num_elements; ++i) {
                data[i] = other.data[i];
            }
        }
    };

   struct bfp_dot_product_t {//new struct for the result of dop-mul
            mantissa_t data;
            exp_t exp;
            bfp_dot_product_t() : data(0), exp(0) {}
        };

   static void receive_int(bfp_t& a,const int input_data[num_elements])
        {
        for (int i = 0; i < num_elements; i++) {
            a.data[i]=input_data[i];
        }
        bfp_headroom(a);//pre-calculate the headroom of received array,so
            shared exp can be confirmed
    }

   static void receive_double(bfp_t& a,const double input_data[])
        {
        for (int i = 0; i < num_elements; i++) {
            a.data[i]=input_data[i];
        }
        bfp_headroom(a);
    }

   static void bfp_init( bfp_t& a,const bfp_t& b)//nomalize
    {
        for (int i = 0; i < num_elements; i++) {
            a.data[i] = b[i]<<b.hr;// every element shift left to ensure the
                largest one's MSB is 1 behind the sign bit
```

```
        }
      a.length = num_elements;
      a.exp = -b.hr;
  }

  static headroom_t bfp_headroom(bfp_t& a)
  {
      ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
          conv_a_data[num_elements];
      for(int i = 0; i < num_elements; ++i) {
          conv_a_data[i] =a.data[i];
              }
      a.hr = headroom_calc(0,conv_a_data);
      return a.hr;
  }
  static unsigned clz(ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
      x) {
      if (x == 0) return 0;
      unsigned int n = 0;
      if ((x & 0xFFFF0000) == 0) { n += 16; x <<= 16; }
      if ((x & 0xFF000000) == 0) { n += 8; x <<= 8; }
      if ((x & 0xF0000000) == 0) { n += 4; x <<= 4; }
      if ((x & 0xC0000000) == 0) { n += 2; x <<= 2; }
      if ((x & 0x80000000) == 0) { n += 1; }
      return n;
  }
  static headroom_t headroom_calc(const int mode_num,const
      ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> v[]){
      ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> largest=0;
      ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> abs_val=0;

      if(mode_num!=3){
          for(int k = 0; k < num_elements; k++){
              abs_val = (v[k] < 0) ?
                  ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>(-v[k])
                  : v[k];
              if(abs_val > largest){
              largest = abs_val;
              }
          }
      }
      else{
      largest = (v[0] < 0) ?
          ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>(-v[0]) : v[0];
      }
      unsigned headroom = 0;
      unsigned headroom_result = 0;
       if (largest == 0) {
           headroom_result =2*I+I_test_value_dot-1;
        }
        else if(largest>=1){
            while (largest >= 1)
            {
                largest >>= 1;
                headroom++;
            }
            headroom_result =2*I-headroom-1+I_test_value_dot;
        }
        else{
```

```
            while (largest < 1 )
            {
                largest <<= 1;
                headroom++;
            }
            headroom_result =2*I+headroom-2+I_test_value_dot;
        }

        return headroom_result - (mode_num == 0 ? I + I_test_value_dot :
            (mode_num == 1 ? I + I_test_value_dot - 1 : (mode_num == 2 ?
            I_test_value_dot : 0)));
    }
    static exp_t check_overflow(const int mode_num,const
        ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> interm[]){
        exp_t overflow,cond;
        headroom_t hr=headroom_calc(mode_num,interm);
        if(mode_num==1){
            cond=0;
        }
        else if(mode_num==2){
            cond=I;
        }
        else{
            cond=I+I_test_value_dot;
        }

        if(hr<=cond){
            overflow=cond-hr;
        }
        else{
            overflow=0;
        }
      return overflow;

    }


    static bfp_t denomalize(bfp_t& a,const bfp_t& b,exp_t c){//used by add
        and sub operations
        for(unsigned i = 0; i < num_elements; ++i) {
                if(c<0){//shift depending on the sign of c
                a.data[i]=b.data[i]<<(-c);
                }
                else{
                a.data[i]=b.data[i]>>c;
                }
            }
            a.length = num_elements;
            return a;
        }

    static bfp_t renomalize(bfp_t& a){
        ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
            conv_a_data[num_elements];
        for(int i = 0; i < num_elements; ++i) {
            conv_a_data[i] =a.data[i];
                }
        a.hr=headroom_calc(0,conv_a_data);//Calculate the result's headroom
            to get the new shared exp
```

```
        for(unsigned i = 0; i < num_elements; ++i) {
            a.data[i]<<=a.hr;//shift left the value according to headroom area
        }
        a.exp=-a.hr;
        a.length = num_elements;
        return a;
    }

    static bfp_dot_product_t renomalize_dot_product(bfp_dot_product_t& a){//same as bfp_init but with different struct
        headroom_t hr;
        ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> conv_a_data[1];
        conv_a_data[0]=a.data;
        hr=headroom_calc(3,conv_a_data);
        a.data<<=hr;
        a.exp=-hr;
        return a;
    }

    static bfp_t add(const bfp_t& a, const bfp_t& b) {
              unsigned i;
              exp_t exp_val,interm_exp,a_shr,b_shr;
              bfp_t c,d,result,result_temp;
            ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> interm[num_elements];
              //ap_fixed<W+1,I+1> interm[num_elements];
              result.length = num_elements;
               if(a.exp<b.exp){//new bfp's shared exp is equal to max(a.exp,b.exp)+1
                   interm_exp=b.exp+1;
               }
               else if(a.exp>b.exp){
                   interm_exp=a.exp+1;
               }
               else{
                   interm_exp=0;
               }
               a_shr=interm_exp-a.exp;//get the shift right number for denomalizing
               b_shr=interm_exp-b.exp;
              denomalize(c,a,a_shr);//return c,d with same exp, so they could be added together
              denomalize(d,b,b_shr);
               for(i = 0; i < num_elements; ++i) {
                interm[i] = c.data[i] + d.data[i];
               }
               exp_val=check_overflow(1,interm);//check ovf if it needs to renomalize
               if(exp_val==0){//no ovf
                   for(i = 0; i < num_elements; ++i) {
                    result_temp.data[i] =interm[i];//send data to result bfp
                   }
                      renomalize(result_temp);//find the largest number to comfirm the exp, and make sure the largest number' MSB is 1 behind the sign bit
                      result.exp=result_temp.exp+interm_exp;
                   for(i = 0; i < num_elements; ++i) {
                        result.data[i] =result_temp.data[i];
               }
```

```
                }
                else{//ovf, but it could only be 1 more bit than the original bit width in add and sub operations
                 for(i = 0; i < num_elements; ++i) {
                     result.data[i] =interm[i]>>exp_val;
                     result.exp=1+interm_exp;
                    }
                }
               result.length = num_elements;
               return result;
    }

    friend bfp_t operator+(const bfp_t& a, const bfp_t& b) {
        return add(a, b);
    }

    static bfp_t subtract(const bfp_t& a, const bfp_t& b) {
       unsigned i;
       exp_t exp_val,interm_exp,a_shr,b_shr;
       bfp_t c,d,result,result_temp;
       ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
           interm[num_elements];
       //ap_fixed<W+1,I+1> interm[num_elements];
        if(a.exp<b.exp){
            interm_exp=b.exp+1;
        }
        else if(a.exp>b.exp){
            interm_exp=a.exp+1;
        }
        else{
            interm_exp=0;
        }
        a_shr=interm_exp-a.exp;
        b_shr=interm_exp-b.exp;
           denomalize(c,a,a_shr);
           denomalize(d,b,b_shr);
        for(i = 0; i < num_elements; ++i) {
         interm[i] = c.data[i] - d.data[i];// different from add here
        }
        exp_val=check_overflow(1,interm);
        if(exp_val==0){
            for(i = 0; i < num_elements; ++i) {
             result_temp.data[i] =interm[i];
            }
               renomalize(result_temp);
               result.exp=result_temp.exp+interm_exp;
            for(i = 0; i < num_elements; ++i) {
                result.data[i] =result_temp.data[i];
        }
        }
        else{
         for(i = 0; i < num_elements; ++i) {
             result.data[i] =interm[i]>>exp_val;
             result.exp=1+interm_exp;
            }
        }
     result.length = num_elements;
     return result;
    }
```

```

    friend bfp_t operator-(const bfp_t& a, const bfp_t& b) {
       return subtract(a, b);
    }

    static bfp_t multiply(const bfp_t& a, const bfp_t& b) {
           unsigned i;
           exp_t exp_val,interm_exp;
           bfp_t c,d,result;
         ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
             interm[num_elements];
          //ap_fixed<2*W,2*I> interm[num_elements];
          interm_exp=a.exp+b.exp;//new shared exp
          for (i = 0; i < num_elements; ++i) {
              interm[i] = a.data[i] * b.data[i];
          }

           exp_val=check_overflow(2,interm);
           if(exp_val==0){
            for(i = 0; i < num_elements; ++i) {
             result.data[i] =interm[i];
            }
               renomalize(result);
               result.exp=result.exp+interm_exp;

        }
        else{
         for(i = 0; i < num_elements; ++i) {
           result.data[i] =interm[i]>>exp_val;
            }
           result.exp=interm_exp+exp_val;
        }
           result.length = num_elements;
          return result;

      }
    friend bfp_t operator*(const bfp_t& a, const bfp_t& b) {
          return multiply(a, b);
      }

    static bfp_dot_product_t dot_product(const bfp_t& a, const bfp_t& b) {
       unsigned i;
       bfp_dot_product_t result;
       exp_t exp_val=0,interm_exp;
       //constexpr unsigned int I_test_value_dot = compute_ITest();
       ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> interm[1];
       //ap_fixed<2*W+I_test_value,2*I+I_test_value> interm_test=0;
       interm_exp=a.exp+b.exp;
       for (i = 0; i < num_elements; ++i) {
          interm[0] += a.data[i] * b.data[i];
          //interm_test=interm>>16;
          //std::cout << interm_test << " ";
       }
       exp_val=check_overflow(3,interm);
       if(exp_val==0){
           result.data =interm[0];
           renomalize_dot_product(result);
           result.exp=result.exp+interm_exp;
        }
```

```
        else{
             result.data =interm[0]>>exp_val;
             result.exp=interm_exp+exp_val;
        }
          return result;
        }
    static bfp_dot_product_t dot_product_m(const mantissa_t data1[num_elements],const mantissa_t data2[num_elements],
            const exp_t exp1, const exp_t exp2, int start1, int start2,int MATRIX_SIZE) {
       unsigned i;
       bfp_dot_product_t result;
       exp_t exp_val=0,interm_exp;
       //constexpr unsigned int I_test_value_dot = compute_ITest();
       ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> interm[1];
       //ap_fixed<2*W+I_test_value,2*I+I_test_value> interm_test=0;
       interm_exp=exp1+exp2;
       for (i = 0; i < MATRIX_SIZE; ++i) {
          interm[0] += data1[i+start1] * data2[i+start2];
          //interm_test=interm>>16;
          //std::cout << interm_test << " ";
       }
       exp_val=check_overflow(3,interm);
       if(exp_val==0){
           result.data =interm[0];
           renomalize_dot_product(result);
           result.exp=result.exp+interm_exp;
        }
        else{
            result.data =interm[0]>>exp_val;
            result.exp=interm_exp+exp_val;
        }
          return result;
        }

    void to_double_array(const mantissa_t data,const exp_t exp, double& result_array) {
                double data_double,exp_double,aa;
                data_double=data.to_double();
                exp_double=exp.to_double();
             result_array =data_double*pow(2,exp_double);
        }
    void to_double_array_1(const bfp_t& a, double result_array[num_elements]) {
                double data_double,exp_double;
                exp_double=a.exp.to_double();
             for (int i = 0; i < num_elements; ++i){
                data_double=a.data[i].to_double();
             result_array[i] =data_double*pow(2,exp_double);
             }
        }
};
```

Listing 13: matrix_multiply.h

```
#define MATRIX_SIZE 7
#define ARRAY_SIZE 50

using BfpMatrix = bfp_fpga<7, 16, 32, 16>;//MATRIX_SIZE=50
using BfpFir = bfp_fpga<4,5,13,5>;
```

```
BfpMatrix::bfp_t init(double a[]);
BfpFir::bfp_t init_fir(double a[]);
BfpMatrix::bfp_dot_product_t matrix_multiply(BfpMatrix::bfp_t
    bfp_a,BfpMatrix::bfp_t bfp_b);
BfpFir::bfp_dot_product_t FIR(BfpFir::bfp_t bfp_a,BfpFir::bfp_t bfp_b);
```

Listing 14: matrix_multiply.cpp

```
#include "bfp_fpga.h"
#include "matrix_multiply.h"


BfpMatrix::bfp_t init(double a[]) {
    BfpMatrix bfp_matrix_multiply;
    BfpMatrix::bfp_t A_matrix_data,bfp_a;
    bfp_matrix_multiply.receive_double(A_matrix_data, a);
    bfp_matrix_multiply.bfp_init(bfp_a, A_matrix_data);
    return bfp_a;
}
BfpMatrix::bfp_dot_product_t matrix_multiply(BfpMatrix::bfp_t
    bfp_a,BfpMatrix::bfp_t bfp_b){
    BfpMatrix::bfp_dot_product_t C_matrix_data;
    BfpMatrix bfp_matrix_multiply;
    C_matrix_data = bfp_matrix_multiply.dot_product(bfp_a,bfp_b);
    return C_matrix_data;
    //bfp_matrix_multiply.to_double_array(C_matrix_data.data,C_matrix_data.exp,
        array);
}

BfpFir::bfp_t init_fir(double a[]) {
    BfpFir bfp_fir;
    BfpFir::bfp_t A_fir_data,bfp_a;
    bfp_fir.receive_double(A_fir_data, a);
    bfp_fir.bfp_init(bfp_a, A_fir_data);
    return bfp_a;
}
BfpFir::bfp_dot_product_t FIR(BfpFir::bfp_t bfp_a,BfpFir::bfp_t bfp_b) {
    BfpFir fir_bfp;
    BfpFir::bfp_dot_product_t output_data;
    output_data =fir_bfp.dot_product(bfp_a,bfp_b);
    return output_data;
    //fir_bfp.to_double_array(output_data.data,output_data.exp, array);
}
```

Listing 15: testbench.cpp

```
#include "bfp_fpga.h"
#include "matrix_multiply.h"
#include <hls_math.h>
#include <ap_int.h>
#include <ap_fixed.h>
#include <iostream>
void generateRandomMatrix(double matrix[MATRIX_SIZE][MATRIX_SIZE]) {
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                matrix[i][j] = -100.0 + static_cast<double>(rand()) /
                    (static_cast<double>(RAND_MAX/200.0));
            }
        }
    }
```

```
void generateRandomdoublearray(double a[ARRAY_SIZE]) {
        for (int i = 0; i < ARRAY_SIZE; i++) {
            a[i] = -100.0 + static_cast<double>(rand()) /
               (static_cast<double>(RAND_MAX/200.0));
        }
    }
void printMatrix(const double matrix[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            std::cout << std::setw(10) << std::fixed << matrix[i][j] << "␣";
        }
        std::cout << std::endl;
    }
}
int main() {
  std::srand(static_cast<unsigned int>(std::time(nullptr)));

  double Amatrix[MATRIX_SIZE][MATRIX_SIZE];
    double Bmatrix[MATRIX_SIZE][MATRIX_SIZE];

    generateRandomMatrix(Amatrix);
    generateRandomMatrix(Bmatrix);
    std::cout << "\nAmatrix:" << std::endl;
    printMatrix(Amatrix);

    std::cout << "\nBmatrix:" << std::endl;
    printMatrix(Bmatrix);
    //double input_A[MATRIX_SIZE], input_B[MATRIX_SIZE];
    double Cmatrix_bfp[MATRIX_SIZE][MATRIX_SIZE] = {0};
    //double array1[MATRIX_SIZE][MATRIX_SIZE]= {0};
     /* double Amatrix[7][7] = {
              {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1}
    };

     double Bmatrix[7][7] = {
              {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1},
                {1, 1, 1, 1, 1, 1, 1}
    };*/
     double Cmatrix[MATRIX_SIZE][MATRIX_SIZE];
    BfpMatrix::bfp_t bfpa;
    BfpMatrix::bfp_t bfpb;
    BfpMatrix::bfp_dot_product_t C_matrix_data;
    double array[MATRIX_SIZE][MATRIX_SIZE]={0};
    double result_array=0;
    BfpMatrix bfp_matrix_multiply;
    double input_A[MATRIX_SIZE]={0}, input_B[MATRIX_SIZE]={0};
    for(int i = 0; i < MATRIX_SIZE; ++i) {
        for(int k = 0; k < MATRIX_SIZE; ++k) {
```

```
                input_A[k]= Amatrix[i][k];
            }
            for(int j = 0; j < MATRIX_SIZE; ++j) {
                for(int k = 0; k < MATRIX_SIZE; ++k) {
                    input_B[k] = Bmatrix[k][j];
                }
                BfpMatrix::bfp_t bfpa=init(input_A);
                BfpMatrix::bfp_t bfpb=init(input_B);
                C_matrix_data=matrix_multiply(bfpa,bfpb);
                //std::cout << "result_array1111: " << result_array<< std::endl;
                bfp_matrix_multiply.to_double_array(C_matrix_data.data,C_matrix_data.exp, result_array);
                array[i][j]=result_array;
            }

        }
        double max_diff = 0.0;
        double sum_diff = 0.0;
        for (int i = 0; i < MATRIX_SIZE; ++i) {
            for (int j = 0; j < MATRIX_SIZE; ++j) {
                for (int k = 0; k < MATRIX_SIZE; ++k) {
                    Cmatrix[i][j] += Amatrix[i][k] * Bmatrix[k][j];
                }
            }
        }

        std::cout << "matrix C computered by fp:\n";
        for (int i = 0; i < MATRIX_SIZE; ++i) {
            for (int j = 0; j < MATRIX_SIZE; ++j) {
                std::cout << std::setw(10) << std::fixed <<Cmatrix[i][j] << " ";
            }
            std::cout << std::endl;
        }

        std::cout << "matrix C computered by bfp:\n";
        for (int i = 0; i < MATRIX_SIZE; ++i) {
            for (int j = 0; j < MATRIX_SIZE; ++j) {
                std::cout <<std::setw(10) << std::fixed << array[i][j] << " ";
            }
            std::cout << std::endl;
        }

        double diff[MATRIX_SIZE][MATRIX_SIZE];
        std::cout << "different between bfp and fp:\n";
        for (int i = 0; i < MATRIX_SIZE; ++i) {
            for (int j = 0; j < MATRIX_SIZE; ++j) {
                diff[i][j]=array[i][j]-Cmatrix[i][j];
                max_diff = std::max(max_diff, abs(diff[i][j]));
                sum_diff += abs(diff[i][j]);
                std::cout << std::setw(10) << std::fixed <<abs(diff[i][j]) << " ";
            }
        }
        std::cout << "\nMax difference: " << max_diff << std::endl;
        std::cout << "Average difference: " << sum_diff / (MATRIX_SIZE*MATRIX_SIZE) << std::endl;
        std::cout << "--------------------------- ADD-SUB-----------------------------" << std::endl;
        double a[ARRAY_SIZE];
```

```
    double b[ARRAY_SIZE];
    double c[ARRAY_SIZE];
    double c_from_bfp[ARRAY_SIZE];
    double diff_add[ARRAY_SIZE];
    max_diff = 0.0;
    sum_diff = 0.0;
    generateRandomdoublearray(a);
    generateRandomdoublearray(b);
    std::cout << "ADD result C computered by fp:\n";
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        c[i]=a[i]+b[i];
        std::cout << std::setw(10) << std::fixed <<c[i] << " ";
    }
    std::cout << std::endl;
    bfp_fpga <ARRAY_SIZE,16,32,16> bfp;
    bfp_fpga <ARRAY_SIZE,16,32,16>::bfp_t
        A_double,B_double,result,bfp_a,bfp_b;
    bfp.receive_double(A_double, a);
    bfp.receive_double(B_double, b);
    bfp.bfp_init(bfp_a, A_double);
    bfp.bfp_init(bfp_b, B_double);
    result=A_double+B_double;
    std::cout << "ADD result C computered by bfp:\n";
    bfp.to_double_array_1(result,c_from_bfp);
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        std::cout << std::setw(10) << std::fixed <<c_from_bfp[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "different between fp and bfp:\n";
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        diff_add[i]=c_from_bfp[i]-c[i];
        max_diff = std::max(max_diff, abs(diff_add[i]));
        sum_diff += abs(diff_add[i]);
        //if(abs(diff_add[i])>0.0001){
        std::cout << std::setw(10) << std::fixed <<abs(diff_add[i]) << " ";
        //}
    }
    std::cout << "\nMax difference: " << max_diff << std::endl;
    std::cout << "Average difference: " << sum_diff / ARRAY_SIZE <<
        std::endl;
    max_diff = 0.0;
    sum_diff = 0.0;
    diff_add[ARRAY_SIZE]=0;
    std::cout << "SUB result C computered by fp:\n";
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        c[i]=a[i]-b[i];
        std::cout << std::setw(10) << std::fixed <<c[i] << " ";
    }
    std::cout << std::endl;
    result=A_double-B_double;
    std::cout << "SUB result C computered by bfp:\n";
    bfp.to_double_array_1(result,c_from_bfp);
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        std::cout << std::setw(10) << std::fixed <<c_from_bfp[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "different between fp and bfp:\n";
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        diff_add[i]=c_from_bfp[i]-c[i];
```

```
        max_diff = std::max(max_diff, abs(diff_add[i]));
        sum_diff += abs(diff_add[i]);
        //if(abs(diff_add[i])>0.0001){
        std::cout << std::setw(10) << std::fixed <<abs(diff_add[i]) << " ";
        //}
    }
    std::cout << "\nMax difference: " << max_diff << std::endl;
    std::cout << "Average difference: " << sum_diff / ARRAY_SIZE << std::endl;
    std::cout << "---------------------------- FIR------------------------------" << std::endl;
        double input[10] = {10.0, 9.0, 11.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0};
    double coeffs[4] = {0.1, 0.2, 0.3, 0.4};
    double output1[10];
    double output0;
    double receive_h[4]= {0.0};
    double receive_x[4]= {0.0};
    max_diff = 0.0;
    sum_diff = 0.0;
    BfpFir::bfp_dot_product_t output_data;
    BfpFir::bfp_t receive_h_data,receive_x_data,bfp_h[10],bfp_x[10];
    BfpFir fir_bfp;
    for (int i = 0; i < 10; ++i) {
        int index = 0;
        for (int j = 0; j < 4; ++j) {
            if (i - j >= 0) {
                receive_h[index] = coeffs[j];
                receive_x[index] = input[i - j];
                //std::cout << "receive_h[index]: " << receive_h[index] << std::endl;
                //std::cout << "receive_x[index]: " << receive_x[index] << std::endl;
                index++;
            }
        }
        BfpFir::bfp_t bfpa=init_fir(receive_h);
        BfpFir::bfp_t bfpb=init_fir(receive_x);

        output_data=FIR(bfpa,bfpb);
        fir_bfp.to_double_array(output_data.data,output_data.exp, output0);
        output1[i] = output0;

    }
    std::cout << "Output by BFP: ";
    for (int i = 0; i < 10; ++i) {
        std::cout << output1[i] << ' ';
    }
    std::cout << std::endl;

    double output[10];
    for (int i = 0; i < 10; ++i) {
        output[i] = 0.0;
    }

    for (int i = 0; i < 10; ++i) {
        for (int j = 0; j < 4; ++j) {
            if (i - j >= 0) {
                output[i] += coeffs[j] * input[i - j];
```

```
                }
            }
        }
        std::cout << "Output_exp by FP: ";
        for (int i = 0; i < 10; ++i) {
            std::cout << output[i] << " ";
        }
        std::cout << std::endl;
        max_diff = 0.0;
        sum_diff = 0.0;
        diff_add[10]=0;
        for (int i = 0; i < 10; ++i) {
            diff_add[i]=output1[i]-output[i];
            max_diff = std::max(max_diff, abs(diff_add[i]));
            sum_diff += abs(diff_add[i]);
            //if(abs(diff_add[i])>0.0001){
            std::cout << std::setw(10) << std::fixed <<abs(diff_add[i]) << " ";
            //}
        }
        std::cout << "\nMax difference: " << max_diff << std::endl;
        std::cout << "Average difference: " << sum_diff / 10 << std::endl;
        }
```

7 APPENDIX B

Amatrix: -86.504715 -7.296976 41.331217 -79.387799 55.607776 87.084567 -81.621754 42.118595 -66.472365 24.857326 93.902402 38.462477 -92.571795 19.418928 -53.807184 19.357891 14.297922 -38.560137 98.242134 -69.188513 71.318705 -66.130558 -2.206488 39.799188 -53.245643
Bmatrix: 24.143193 5.703909 94.384594 4.770043 14.932707 48.631245 -9.604175 30.173650 34.672079 -7.431257 76.104007 39.646596 -50.938444 -6.790368 -74.095889 85.436567 -25.095370 -20.413831 -85.467086 57.933287 97.369304 -72.869045 22.800378 -6.692709 -27.921384
matrix C computered by fp: -666.020287 -844.510901 -7601.749292 5466.590415 -10451.839316 -1920.304333 2807.310007 5534.871628 2814.237011 -5758.867764 -6487.584146 -70.393144 13115.717777 1110.525810 10602.955273 -115.260597 1020.595892 639.677915 -7083.496123 10663.290535 -3446.257972 3835.628962 2821.907032 -4982.870782 5512.296034
matrix C computered by bfp: -666.021628 -844.509803 -7601.749722 5466.590553 -10451.837296 -1920.305948 2807.311750 5534.870266 2814.236042 -5758.867874 -6487.587711 -70.393507 13115.716293 1110.526584 10602.955276 -115.264430 1020.595206 639.677373 -7083.495144 10663.290321 -3446.260942 3835.629539 2821.906139 -4982.870094 5512.295147
different between bfp and fp: 0.001342 0.001098 0.000429 0.000139 0.002020 0.001615 0.001744 0.001362 0.000969 0.000110 0.003566 0.000363 0.001483 0.000774 0.000003 0.003833 0.000686 0.000542 0.000979 0.000214 0.002971 0.000577 0.000893 0.000688 0.000887
Max difference: 0.003833 Average difference: 0.001171
——————————- ADD-SUB——————————
ADD result C computered by fp: 120.322275 11.230811 118.906217 98.220771 -77.425459 -45.442061 -32.514420 -69.948424 -145.524461 -3.808710 -28.009888 4.089480 -57.667776 -115.286721 4.040651 153.324992 -56.331065 178.240303 -102.645955 152.562029 106.039613 17.835017 -22.724082 -20.099490 32.929472 -19.983520 -46.406446 -11.828974 19.122898 -79.555651 -160.844752 -0.824000 18.152409 -78.292184 156.761376 -66.957610 4.254280 57.374798 96.780297 -14.825892 -108.645894 85.048982 -111.825922 -36.530656 -28.363903 -19.214454 45.185705 -1.910459 36.024049 27.411725
ADD result C computered by bfp: 120.322250 11.230789 118.906189 98.220749 -77.425476 -

45.442078 -32.514435 -69.948441 -145.524490 -3.808731 -28.009903 4.089462 -57.667786 -115.286728 4.040634 153.324982 -56.331085 178.240295 -102.645966 152.562012 106.039597 17.834991 -22.724106 -20.099503 32.929459 -19.983536 -46.406464 -11.828995 19.122879 -79.555664 -160.844757 -0.824020 18.152390 -78.292206 156.761368 -66.957626 4.254272 57.374786 96.780273 -14.825897 -108.645905 85.048965 -111.825928 -36.530670 -28.363922 -19.214462 45.185699 -1.910477 36.024033 27.411713
different between fp and bfp: 0.000025 0.000022 0.000028 0.000022 0.000018 0.000017 0.000015 0.000017 0.000030 0.000021 0.000015 0.000018 0.000010 0.000007 0.000016 0.000010 0.000020 0.000007 0.000011 0.000017 0.000016 0.000025 0.000024 0.000012 0.000013 0.000016 0.000018 0.000020 0.000019 0.000014 0.000005 0.000021 0.000020 0.000022 0.000008 0.000017 0.000008 0.000011 0.000024 0.000005 0.000011 0.000017 0.000005 0.000014 0.000019 0.000009 0.000006 0.000018 0.000016 0.000013
Max difference: 0.000030 Average difference: 0.000016
SUB result C computered by fp: 32.642598 1.324503 66.090884 88.241218 -14.294870 -38.282418 -52.027955 -112.240974 48.518326 -1.495407 133.829768 125.888852 -106.076235 -57.686087 184.130375 -35.798212 -91.775262 21.027253 55.055391 14.612262 14.343699 -9.942930 -92.287973 -88.271737 -45.509201 84.481338 51.722770 82.271798 -17.859432 -56.904813 -26.960051 61.220130 -86.581011 54.274117 -3.613392 -95.052950 87.807855 73.763237 -68.160039 51.161229 -11.713004 -35.663930 55.946532 -24.268319 70.827357 -92.684713 -22.913297 -3.576769 153.245643 135.953856
SUB result C computered by bfp: 32.642593 1.324509 66.090881 88.241226 -14.294861 -38.282410 -52.027954 -112.240982 48.518326 -1.495407 133.829758 125.888840 -106.076233 -57.686081 184.130386 -35.798218 -91.775269 21.027252 55.055389 14.612274 14.343704 -9.942932 -92.287979 -88.271744 -45.509201 84.481339 51.722778 82.271805 -17.859421 -56.904816 -26.960052 61.220139 -86.581009 54.274109 -3.613388 -95.052963 87.807861 73.763245 -68.160034 51.161224 -11.713013 -35.663925 55.946533 -24.268311 70.827362 -92.684708 -22.913300 -3.576767 153.245651 135.953857
different between fp and bfp: 0.000004 0.000005 0.000003 0.000008 0.000009 0.000009 0.000001 0.000008 0.000001 0.000000 0.000010 0.000012 0.000002 0.000006 0.000011 0.000006 0.000007 0.000001 0.000002 0.000012 0.000005 0.000002 0.000006 0.000007 0.000000 0.000001 0.000008 0.000007 0.000011 0.000003 0.000001 0.000009 0.000002 0.000008 0.000003 0.000014 0.000006 0.000007 0.000005 0.000005 0.000009 0.000005 0.000002 0.000008 0.000005 0.000006 0.000003 0.000002 0.000008 0.000001
Max difference: 0.000014 Average difference: 0.000006
——————————- FIR———————————
Output by BFP: 0.099609 0.408554 1.017273 2.035522 3.034546 4.033447 5.032471 6.070068 7.069092 8.067871
Output exp by FP: 0.100000 0.400000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
different between fp and bfp: 0.000391 0.008554 0.017273 0.035522 0.034546 0.033447 0.032471 0.070068 0.069092 0.067871 Max difference: 0.070068 Average difference: 0.036924

8 Bibliography

References

[1] David Elam, Cesar Iovescu, A Block Floating Point Implementation for an N-Point FFT on the TMS320C55x DSP.September 2003. extension://oikmahiipjniocckomdccmplodldodja/pdf-viewer/web/viewer.html?file=https%3A%2F%2Fwww.ti.com%2Flit%2Fan%2Fspra948%2Fspra948.pdf#=&zoom=80 (cit. on p. 4).