



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	2
1.1	Block floating point	2
1.2	Code operations	3
2	Simulations with base calculations	3
2.1	ADD	3
2.1.1	Code implements	3
2.1.2	Testbench codes	4
2.1.3	Comparison of results	5
2.2	SUB	6
2.2.1	Code implements	6
2.2.2	Testbench codes	7
2.2.3	Comparison of results	8
2.3	MUL	8
2.3.1	Code implements	8
2.3.2	Testbench codes	9
2.4	DOT-MUL	9
2.4.1	Code implements	9
2.4.2	Testbench codes	10
3	Simulations with special operations	10
3.1	Matrix-multiply	10
3.1.1	Testbench codes	10
3.1.2	Comparison of results	11
3.2	FIR	11
3.2.1	Testbench codes	11
3.2.2	Comparison of results	12
4	Synthesise with matrix multiply and FIR	12
4.1	Global setting	12
4.2	Matrix-multiply	13
4.2.1	Codes	13
4.3	FIR	13
4.3.1	Codes	13
4.4	Synthesis summary	13
5	Cosimulation	14
6	APPENDIX A	16
7	APPENDIX B	28
8	Bibliography	30

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

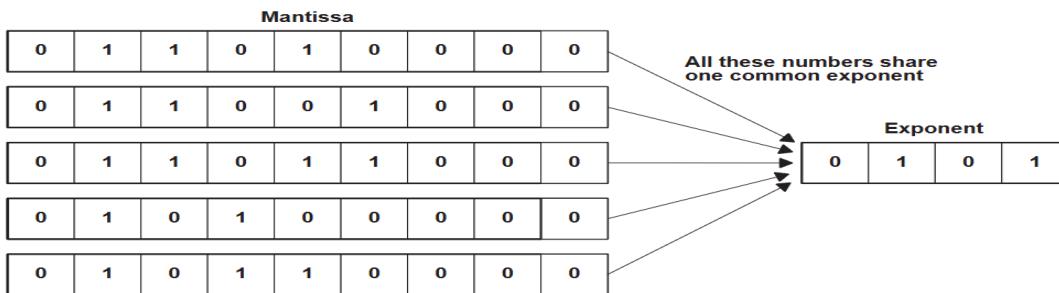


Figure 1: Diagram of Block Floating-Point Representation

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: normalize.

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.

denormalize: used by addition and subtraction operations.

renormalize: renormalize block floating point.

renormalize_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

```

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

```

```

7     result.length = num_elements;
8     if(a.exp<b.exp){//new bfp's shared exp is equal to
9         max(a.exp,b.exp)+1
10        interm_exp=b.exp+1;
11    }
12    else if(a.exp>b.exp){
13        interm_exp=a.exp+1;
14    }
15    else{
16        interm_exp=0;
17    }
18    a_shr=interm_exp-a.exp;//get the shift right number for
19    denormalizing
20    b_shr=interm_exp-b.exp;
21    denormalize(c,a,a_shr);//return c,d with same exp, so they
22    could be added together
23    denormalize(d,b,b_shr);
24    for(i = 0; i < num_elements; ++i) {
25        interm[i] = c.data[i] + d.data[i];
26    }
27    exp_val=check_overflow(1,interm);//check ovf if it needs to
28    renormalize
29    if(exp_val==0){//no ovf
30        for(i = 0; i < num_elements; ++i) {
31            result_temp.data[i] = interm[i];//send data to result bfp
32        }
33        renormalize(result_temp);//find the largest number to
34        comfirm the exp, and make sure the largest number'
35        MSB is 1 behind the sign bit
36        result.exp=result_temp.exp+interm_exp;
37        for(i = 0; i < num_elements; ++i) {
38            result.data[i] =result_temp.data[i];
39        }
40    }
41    else{//ovf, but it could only be 1 more bit than the original
42        bit width in add and sub operations
43        for(i = 0; i < num_elements; ++i) {
44            result.data[i] =interm[i]>>exp_val;
45            result.exp=1+interm_exp;
46        }
47    }
48    result.length = num_elements;
49    return result;
50 }
51
52 friend bfp_t operator+(const bfp_t& a, const bfp_t& b) {
53     return add(a, b);
54 }
```

2.1.2 Testbench codes

Listing 2: ADD Testbench

```

1 std::cout << "ADD result computed by bfp:\n";
2 bfp.to_double_array_1(result,c_from_bfp);
3 for (int i = 0; i < ARRAY_SIZE; ++i) {
4     std::cout << std::setw(10) << std::fixed << c_from_bfp[i] << " ";
5 }
6 std::cout << std::endl;
```

```

7 std::cout << "different between fp and bfp:\n";
8 for (int i = 0; i < MATRIX_SIZE; ++i) {
9     diff_add[i]=c_from_bfp[i]-c[i];
10    if(abs(diff_add[i])>0.0001){
11        std::cout << std::setw(10) << std::fixed <<diff_add[i] << "\n";
12    }
13 }
```

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

```

1 static bfp_t subtract(const bfp_t& a, const bfp_t& b) {
2     unsigned i;
3     exp_t exp_val, interm_exp, a_shr, b_shr;
4     bfp_t c, d, result, result_temp;
5     ap_fixed<2*W+I_test_value_dot, 2*I+I_test_value_dot>
6         interm[num_elements];
7     //ap_fixed<W+1, I+1> interm[num_elements];
8     if(a.exp<b.exp){
9         interm_exp=b.exp+1;
10    }
11    else if(a.exp>b.exp){
12        interm_exp=a.exp+1;
13    }
14    else{
15        interm_exp=0;
16    }
17    a_shr=interm_exp-a.exp;
18    b_shr=interm_exp-b.exp;
19    denormalize(c,a,a_shr);
20    denormalize(d,b,b_shr);
21    for(i = 0; i < num_elements; ++i) {
22        interm[i] = c.data[i] - d.data[i];// different from add here
23    }
24    exp_val=check_overflow(1,interm);
25    if(exp_val==0){
26        for(i = 0; i < num_elements; ++i) {
27            result_temp.data[i] = interm[i];
28        }
29        renormalize(result_temp);
30        result.exp=result_temp.exp+interm_exp;
31        for(i = 0; i < num_elements; ++i) {
32            result.data[i] = result_temp.data[i];
33        }
34    }
35    else{
36        for(i = 0; i < num_elements; ++i) {
37            result.data[i] = interm[i]>>exp_val;
38            result.exp=1+interm_exp;
39        }
40    }
41    result.length = num_elements;
42    return result;
43}
44 friend bfp_t operator-(const bfp_t& a, const bfp_t& b) {
45     return subtract(a, b);
46}
```

2.2.2 Testbench codes

Listing 4: SUB Testbench

```
1 std::cout << "SUB result C computered by fp:\n";
2 for (int i = 0; i < ARRAY_SIZE; ++i) {
3     c[i]=a[i]-b[i];
4     std::cout << std::setw(10) << std::fixed << c[i] << " ";
5 }
6 std::cout << std::endl;
7 result=A_double-B_double;
8 std::cout << "SUB result C computered by bfp:\n";
9 bfp.to_double_array_1(result,c_from_bfp);
10 for (int i = 0; i < ARRAY_SIZE; ++i) {
11     std::cout << std::setw(10) << std::fixed << c_from_bfp[i] << " ";
12 }
13 std::cout << std::endl;
14 std::cout << "different between fp and bfp:\n";
15 for (int i = 0; i < MATRIX_SIZE; ++i) {
16     diff_add[i]=c_from_bfp[i]-c[i];
17     if(abs(diff_add[i])>0.0001){
18         std::cout << std::setw(10) << std::fixed << diff_add[i] << " ";
19     }
20 }
```

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

```

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

```

5     ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot>
6         interm[num_elements];
7 //ap_fixed<2*W,2*I> interm[num_elements];
8 interm_exp=a.exp+b.exp;//new shared exp
9 for (i = 0; i < num_elements; ++i) {
10     interm[i] = a.data[i] * b.data[i];
11 }
12
13     exp_val=check_overflow(2,interm);
14 if(exp_val==0){
15     for(i = 0; i < num_elements; ++i) {
16         result.data[i] =interm[i];
17     }
18     renormalize(result);
19     result.exp=result.exp+interm_exp;
20 }
21 else{
22     for(i = 0; i < num_elements; ++i) {
23         result.data[i] =interm[i]>>exp_val;
24     }
25     result.exp=interm_exp+exp_val;
26 }
27     result.length = num_elements;
28 return result;
29 }
30 friend bfp_t operator*(const bfp_t& a, const bfp_t& b) {
31     return multiply(a, b);
32 }

```

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

```

1 static bfp_dot_product_t dot_product(const bfp_t& a, const bfp_t& b) {
2     unsigned i;
3     bfp_dot_product_t result;
4     exp_t exp_val=0,interm_exp;
5     bfp_t c,d;
6 //constexpr unsigned int I_test_value_dot = compute_ITest();
7 ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot> interm[1];
8 //ap_fixed<2*W+I_test_value ,2*I+I_test_value> interm_test=0;
9 interm_exp=a.exp+b.exp;
10 for (i = 0; i < num_elements; ++i) {
11     interm[0] += a.data[i] * b.data[i];
12     //interm_test=interm>>16;
13     //std::cout << interm_test << " ";
14 }
15 exp_val=check_overflow(3,interm);
16 if(exp_val==0){

```

```

17     result.data = interm[0];
18     renormalize_dot_product(result);
19     result.exp=result.exp+interm_exp;
20 }
21 else{
22     result.data = interm[0]>>exp_val;
23     result.exp=interm_exp+exp_val;
24 }
25 return result;
26 }
```

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

```

1 int main() {
2     std::srand(static_cast<unsigned int>(std::time(nullptr)));
3
4     double Amatrix[MATRIX_SIZE][MATRIX_SIZE];
5     double Bmatrix[MATRIX_SIZE][MATRIX_SIZE];
6
7     generateRandomMatrix(Amatrix);
8     generateRandomMatrix(Bmatrix);
9     std::cout << "\nAmatrix:" << std::endl;
10    printMatrix(Amatrix);
11
12    std::cout << "\nBmatrix:" << std::endl;
13    printMatrix(Bmatrix);
14    double input_A[MATRIX_SIZE], input_B[MATRIX_SIZE];
15    double Cmatrix[MATRIX_SIZE][MATRIX_SIZE];
16    BfpMatrix::bfp_t bfpa;
17    BfpMatrix::bfp_t bfpb;
18    BfpMatrix::bfp_dot_product_t C_matrix_data;
19    double array[MATRIX_SIZE][MATRIX_SIZE]={0};
20    double result_array=0;
21    BfpMatrix bfp_matrix_multiply;
22    double input_A[MATRIX_SIZE]={0}, input_B[MATRIX_SIZE]={0};
23    for(int i = 0; i < MATRIX_SIZE; ++i) {
24        for(int k = 0; k < MATRIX_SIZE; ++k) {
25            input_A[k]= Amatrix[i][k];
26        }
27        for(int j = 0; j < MATRIX_SIZE; ++j) {
28            for(int k = 0; k < MATRIX_SIZE; ++k) {
29                input_B[k] = Bmatrix[k][j];
29            }
29            BfpMatrix::bfp_t bfpa=init(input_A);
30            BfpMatrix::bfp_t bfpb=init(input_B);
31            C_matrix_data=matrix_multiply(bfpa,bfpb);
32        }
33    }
33 }
```

```

34     //std::cout << "result_array1111: " << result_array<< std::endl;
35     bfp_matrix_multiply.to_double_array(C_matrix_data.data,C_matrix_data.exp,
36                                         result_array);
37     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

```

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

```

```

8   for (int i = 0; i < 10; ++i) {
9     int index = 0;
10    for (int j = 0; j < 4; ++j) {
11      if (i - j >= 0) {
12        receive_h[index] = coeffs[j];
13        receive_x[index] = input[i - j];
14        //std::cout << "receive_h[index]: " << receive_h[index] <<
15        std::endl;
16        //std::cout << "receive_x[index]: " << receive_x[index] <<
17        std::endl;
18        index++;
19    }
20    BfpFir::bfp_t bfpa = init_fir(receive_h);
21    BfpFir::bfp_t bfpb = init_fir(receive_x);
22
23    output_data = FIR(bfpa, bfpb);
24    fir_bfp.to_double_array(output_data.data, output_data.exp, output0);
25    output1[i] = output0;
26  }

```

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

```

1
2 #define MATRIX_SIZE 7
3 #define ARRAY_SIZE 50
4
5 using BfpMatrix = bfp_fpga<MATRIX_SIZE, 16, 32, 16>;
6 using BfpFir = bfp_fpga<4,5,13,5>;

```

4.2 Matrix-multiply

4.2.1 Codes

Listing 10: Matrix-multiply

```

1 BfpMatrix::bfp_t init(double a[]) {
2     BfpMatrix bfp_matrix_multiply;
3     BfpMatrix::bfp_t A_matrix_data,bfp_a;
4     bfp_matrix_multiply.receive_double(A_matrix_data, a);
5     bfp_matrix_multiply.bfp_init(bfp_a, A_matrix_data);
6     return bfp_a;
7 }
8 BfpMatrix::bfp_dot_product_t matrix_multiply(BfpMatrix::bfp_t
9     bfp_a,BfpMatrix::bfp_t bfp_b){
10    BfpMatrix::bfp_dot_product_t C_matrix_data;
11    BfpMatrix bfp_matrix_multiply;
12    C_matrix_data = bfp_matrix_multiply.dot_product(bfp_a,bfp_b);
13    return C_matrix_data;
14    //bfp_matrix_multiply.to_double_array(C_matrix_data.data,C_matrix_data.exp,
15    array);
15 }
```

4.3 FIR

4.3.1 Codes

Listing 11: FIR

```

1 BfpFir::bfp_t init_fir(double a[]) {
2     BfpFir bfp_fir;
3     BfpFir::bfp_t A_fir_data,bfp_a;
4     bfp_fir.receive_double(A_fir_data, a);
5     bfp_fir.bfp_init(bfp_a, A_fir_data);
6     return bfp_a;
7 }
8 BfpFir::bfp_dot_product_t FIR(BfpFir::bfp_t bfp_a,BfpFir::bfp_t bfp_b) {
9     BfpFir fir_bfp;
10    BfpFir::bfp_dot_product_t output_data;
11    output_data =fir_bfp.dot_product(bfp_a,bfp_b);
12    return output_data;
13    //fir_bfp.to_double_array(output_data.data,output_data.exp, array);
14 }
```

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		Performance & Resource Estimates ⓘ											
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		Performance & Resource Estimates ⓘ											
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	-	-	-	-	-	-	1	1	-	no	0	0	63	89	0
↳ VITIS_LOOP_119_2	-	-	-	-	-	-	-	-	-	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		Solution:																																																																																												
Date: Fri Jun 14 17:04:33 2024		solution1 (Vivado IP Flow Target)																																																																																												
Version: 2022.2 (Build 3670227 on Oct 13 2022)		Product family: virtexplus																																																																																												
Project: v7		Target device: xcvu11p-flga2577-1-e																																																																																												
Status: Pass																																																																																														
Cosim Options																																																																																														
Tool: Vivado XSIM		RTL: Verilog																																																																																												
Performance Estimates																																																																																														
<table border="1"> <thead> <tr> <th>Modules & Loops</th> <th>Avg II</th> <th>Max II</th> <th>Min II</th> <th>Avg Latency</th> <th>Max Latency</th> <th>Min Latency</th> </tr> </thead> <tbody> <tr> <td>matrix multiply</td> <td>17</td> <td>17</td> <td>17</td> <td>16</td> <td>16</td> <td>16</td> </tr> <tr> <td>dot_product</td> <td>17</td> <td>17</td> <td>17</td> <td>11</td> <td>11</td> <td>11</td> </tr> <tr> <td>dot_product_Pipeline_VITIS_LOOP_335_1</td> <td>17</td> <td>17</td> <td>17</td> <td>7</td> <td>7</td> <td>7</td> </tr> <tr> <td> ↳ VITIS_LOOP_335_1</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td>dot_product_Pipeline_VITIS_LOOP_116_3</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td> ↳ VITIS_LOOP_116_3</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td>dot_product_Pipeline_VITIS_LOOP_108_2</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td> ↳ VITIS_LOOP_108_2</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td>dot_product_Pipeline_VITIS_LOOP_116_32</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td> ↳ VITIS_LOOP_116_3</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td>dot_product_Pipeline_VITIS_LOOP_108_21</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> <tr> <td> ↳ VITIS_LOOP_108_2</td> <td>17</td> <td>17</td> <td>17</td> <td>8</td> <td>8</td> <td>8</td> </tr> </tbody> </table>				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	17	17	17	8	8	8	↳ VITIS_LOOP_116_3	17	17	17	8	8	8	dot_product_Pipeline_VITIS_LOOP_108_2	17	17	17	8	8	8	↳ VITIS_LOOP_108_2	17	17	17	8	8	8	dot_product_Pipeline_VITIS_LOOP_116_32	17	17	17	8	8	8	↳ VITIS_LOOP_116_3	17	17	17	8	8	8	dot_product_Pipeline_VITIS_LOOP_108_21	17	17	17	8	8	8	↳ VITIS_LOOP_108_2	17	17	17	8	8	8
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	17	17	17	8	8	8																																																																																								
↳ VITIS_LOOP_116_3	17	17	17	8	8	8																																																																																								
dot_product_Pipeline_VITIS_LOOP_108_2	17	17	17	8	8	8																																																																																								
↳ VITIS_LOOP_108_2	17	17	17	8	8	8																																																																																								
dot_product_Pipeline_VITIS_LOOP_116_32	17	17	17	8	8	8																																																																																								
↳ VITIS_LOOP_116_3	17	17	17	8	8	8																																																																																								
dot_product_Pipeline_VITIS_LOOP_108_21	17	17	17	8	8	8																																																																																								
↳ VITIS_LOOP_108_2	17	17	17	8	8	8																																																																																								

Figure 6: matrix mul by bfp cosimulation result

General Information		Solution:																						
Date: Mon Jun 24 16:42:39 2024		solution1 (Vivado IP Flow Target)																						
Version: 2022.2 (Build 3670227 on Oct 13 2022)		Product family: virtexplus																						
Project: fp		Target device: xcvu11p-flga2577-1-e																						
Status: Pass																								
Cosim Options																								
Tool: Vivado XSIM		RTL: Verilog																						
Performance Estimates																								
<table border="1"> <thead> <tr> <th>Modules & Loops</th> <th>Avg II</th> <th>Max II</th> <th>Min II</th> <th>Avg Latency</th> <th>Max Latency</th> <th>Min Latency</th> </tr> </thead> <tbody> <tr> <td>matrix_multiply_fp</td> <td>36</td> <td>36</td> <td>36</td> <td>35</td> <td>35</td> <td>35</td> </tr> <tr> <td>dot_product</td> <td></td> <td></td> <td></td> <td>35</td> <td>35</td> <td>35</td> </tr> </tbody> </table>				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
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

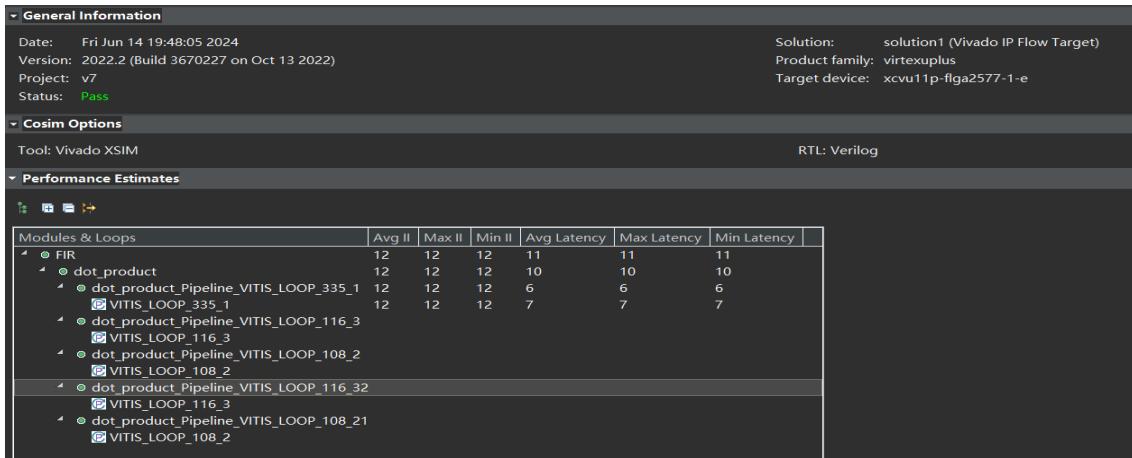


Figure 8: fir by bfp cosimulation result

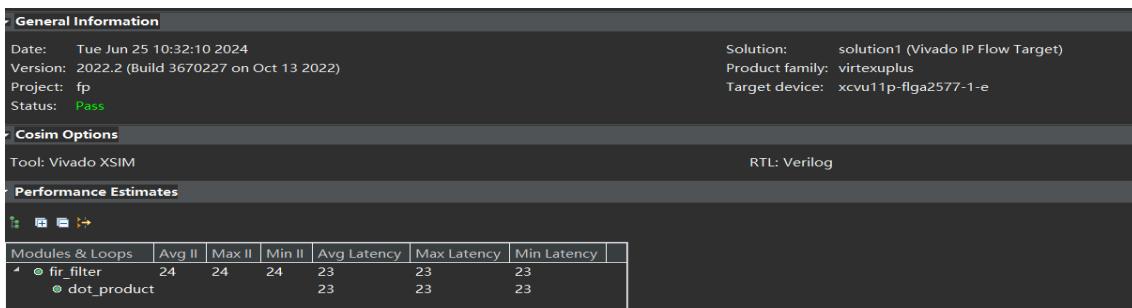


Figure 9: fir by fp cosimulation result

6 APPENDIX A

Listing 12: bfp_fpga.cpp

```

1 #pragma once
2 #include <hls_math.h>
3 #include <ap_int.h>
4 #include <ap_fixed.h>
5 //#include "matrix_multiply.h"
6
7
8 template <unsigned num_elements, unsigned exp_t_width, unsigned W,unsigned I>
9 class bfp_fpga {
10 private:
11 public:
12     static constexpr unsigned custom_log2(unsigned n) {
13         int log = 0;
14         while (n >= 1) ++log;
15         return log;
16     }

```

```

17
18     static constexpr unsigned I_test_value_dot = custom_log2(num_elements)+1;
19     using mantissa_t = ap_fixed<W,I>;
20     using exp_t = ap_int<exp_t_width>;
21     using headroom_t = ap_int<exp_t_width>;
22
23
24
25 struct bfp_t {
26     mantissa_t data[num_elements];
27     exp_t exp;
28     headroom_t hr;
29     unsigned length;
30     bfp_t() : exp(0), hr(0), length(0){}
31     //LHS and RHS by using []
32     mantissa_t& operator[](unsigned index) {
33         return data[index];
34     }
35
36     const mantissa_t& operator[](unsigned index) const {
37         return data[index];
38     }
39     //copy constructor
40     bfp_t(const bfp_t& other)
41         : exp(other.exp), hr(other.hr), length(other.length) {
42         for (unsigned i = 0; i < num_elements; ++i) {
43             data[i] = other.data[i];
44         }
45     }
46 };
47
48 struct bfp_dot_product_t {//new struct for the result of dop-mul
49     mantissa_t data;
50     exp_t exp;
51     bfp_dot_product_t() : data(0), exp(0) {}
52 };
53
54 static void receive_int(bfp_t& a,const int input_data[num_elements])
55 {
56     for (int i = 0; i < num_elements; i++) {
57         a.data[i]=input_data[i];
58     }
59     bfp_headroom(a);//pre-calculate the headroom of received array, so
60     // shared exp can be confirmed
61 }
62
63 static void receive_double(bfp_t& a,const double input_data[])
64 {
65     for (int i = 0; i < num_elements; i++) {
66         a.data[i]=input_data[i];
67     }
68     bfp_headroom(a);
69 }
70
71 static void bfp_init( bfp_t& a,const bfp_t& b)//normalize
72 {
73     for (int i = 0; i < num_elements; i++) {
74         a.data[i] = b[i]<<b.hr;// every element shift left to ensure the
75         // largest one's MSB is 1 behind the sign bit

```

```

74     }
75     a.length = num_elements;
76     a.exp = -b.hr;
77 }
78
79 static headroom_t bfp_headroom(bfp_t& a)
80 {
81     ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot>
82         conv_a_data[num_elements];
83     for(int i = 0; i < num_elements; ++i) {
84         conv_a_data[i] = a.data[i];
85     }
86     a.hr = headroom_calc(0,conv_a_data);
87     return a.hr;
88 }
89 static unsigned __attribute__((noinline)) __attribute__((naked))
90     __attribute__((noreturn)) __attribute__((noinline))
91     __attribute__((noinline)) __attribute__((naked))
92     __attribute__((noinline)) __attribute__((naked))
93     __attribute__((noinline)) __attribute__((naked))
94     __attribute__((noinline)) __attribute__((naked))
95     __attribute__((noinline)) __attribute__((naked))
96     __attribute__((noinline)) __attribute__((naked))
97     __attribute__((noinline)) __attribute__((naked))
98     __attribute__((noinline)) __attribute__((naked))
99     __attribute__((noinline)) __attribute__((naked))
100    __attribute__((noinline)) __attribute__((naked))
101
102    if (x == 0) return 0;
103    unsigned int n = 0;
104    if ((x & 0xFFFF0000) == 0) { n += 16; x <= 16; }
105    if ((x & 0xFF000000) == 0) { n += 8; x <= 8; }
106    if ((x & 0xF0000000) == 0) { n += 4; x <= 4; }
107    if ((x & 0xC0000000) == 0) { n += 2; x <= 2; }
108    if ((x & 0x80000000) == 0) { n += 1; }
109    return n;
110 }
111
112 static headroom_t headroom_calc(const int mode_num, const
113     ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot> v[]){
114     ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot> largest=0;
115     ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot> abs_val=0;
116
117     if(mode_num!=3){
118         for(int k = 0; k < num_elements; k++){
119             abs_val = (v[k] < 0) ?
120                 ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot>(-v[k])
121                 : v[k];
122             if(abs_val > largest){
123                 largest = abs_val;
124             }
125         }
126     }
127     else{
128         largest = (v[0] < 0) ?
129             ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot>(-v[0]) : v[0];
130     }
131     unsigned headroom = 0;
132     unsigned headroom_result = 0;
133     if (largest == 0) {
134         headroom_result = 2*I+I_test_value_dot-1;
135     }
136     else if(largest>=1){
137         while (largest >= 1)
138         {
139             largest >>= 1;
140             headroom++;
141         }
142         headroom_result = 2*I-headroom-1+I_test_value_dot;
143     }
144     else{
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1495
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1677
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1689
1689
1690
1691
1692
1693
1694
1695
1695
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1887
1888
1889
1889
1890
1891
1892
1893
1894
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2087
2088
2089
2089
2090
2091
2092
2093
2094
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2187
2188
2189
2189
2190
2191
2192
2193
2194
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2287
2288
2289
2289
2290
2291
2292
2293
2294
2294
2295
2296
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2368
2369
2370
2371
23
```

```

127         while (largest < 1 )
128     {
129         largest <= 1;
130         headroom++;
131     }
132     headroom_result =2*I+headroom-2+I_test_value_dot;
133 }
134
135     return headroom_result - (mode_num == 0 ? I + I_test_value_dot :
136     (mode_num == 1 ? I + I_test_value_dot - 1 : (mode_num == 2 ?
137     I_test_value_dot : 0)));
138 }
139
140 static exp_t check_overflow(const int mode_num,const
141 ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> interm[]){
142     exp_t overflow,cond;
143     headroom_t hr=headroom_calc(mode_num,interm);
144     if(mode_num==1){
145         cond=0;
146     }
147     else if(mode_num==2){
148         cond=I;
149     }
150     else{
151         cond=I+I_test_value_dot;
152     }
153
154     if(hr<=cond){
155         overflow=cond-hr;
156     }
157     else{
158         overflow=0;
159     }
160
161     return overflow;
162 }
163
164 static bfp_t denominize(bfp_t& a,const bfp_t& b,exp_t c){//used by add
165     and sub operations
166     for(unsigned i = 0; i < num_elements; ++i) {
167         if(c<0){//shift depending on the sign of c
168             a.data[i]=b.data[i]<<(-c);
169         }
170         else{
171             a.data[i]=b.data[i]>>c;
172         }
173     }
174     a.length = num_elements;
175     return a;
176 }
177
178 static bfp_t renormalize(bfp_t& a){
179     ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
180     conv_a_data[num_elements];
181     for(int i = 0; i < num_elements; ++i) {
182         conv_a_data[i] =a.data[i];
183     }
184     a.hr=headroom_calc(0,conv_a_data);//Calculate the result's headroom
185     to get the new shared exp

```

```

180     for(unsigned i = 0; i < num_elements; ++i) {
181         a.data[i]<<=a.hr;//shift left the value according to headroom area
182     }
183     a.exp=-a.hr;
184     a.length = num_elements;
185     return a;
186 }
187
188 static bfp_dot_product_t renormalize_dot_product(bfp_dot_product_t&
189 a){//same as bfp_init but with different struct
190     headroom_t hr;
191     ap_fixed<2*W+I_I_test_value_dot,2*I+I_I_test_value_dot> conv_a_data[1];
192     conv_a_data[0]=a.data;
193     hr=headroom_calc(3,conv_a_data);
194     a.data<<=hr;
195     a.exp=-hr;
196     return a;
197 }
198
199 static bfp_t add(const bfp_t& a, const bfp_t& b) {
200     unsigned i;
201     exp_t exp_val,interm_exp,a_shr,b_shr;
202     bfp_t c,d,result,result_temp;
203     ap_fixed<2*W+I_I_test_value_dot,2*I+I_I_test_value_dot>
204         interm[num_elements];
205     //ap_fixed<W+1,I+1> interm[num_elements];
206     result.length = num_elements;
207     if(a.exp<b.exp){//new bfp's shared exp is equal to
208         max(a.exp,b.exp)+1
209         interm_exp=b.exp+1;
210     }
211     else if(a.exp>b.exp){
212         interm_exp=a.exp+1;
213     }
214     else{
215         interm_exp=0;
216     }
217     a_shr=interm_exp-a.exp;//get the shift right number for
218     //denomalizing
219     b_shr=interm_exp-b.exp;
220     denormalize(c,a,a_shr);//return c,d with same exp, so they
221     //could be added together
222     denormalize(d,b,b_shr);
223     for(i = 0; i < num_elements; ++i) {
224         interm[i] = c.data[i] + d.data[i];
225     }
226     exp_val=check_overflow(1,interm);//check ovf if it needs to
227     //renomalize
228     if(exp_val==0){//no ovf
229         for(i = 0; i < num_elements; ++i) {
230             result_temp.data[i] = interm[i];//send data to result bfp
231         }
232         renormalize(result_temp);//find the largest number to
233         //comfirm the exp, and make sure the largest number'
234         //MSB is 1 behind the sign bit
235         result.exp=result_temp.exp+interm_exp;
236         for(i = 0; i < num_elements; ++i) {
237             result.data[i] = result_temp.data[i];
238         }
239     }
240 }
```

```

231     }
232     else{//ovf, but it could only be 1 more bit than the original
233         bit width in add and sub operations
234         for(i = 0; i < num_elements; ++i) {
235             result.data[i] = interm[i]>>exp_val;
236             result.exp=1+interm_exp;
237         }
238         result.length = num_elements;
239         return result;
240     }
241
242     friend bfp_t operator+(const bfp_t& a, const bfp_t& b) {
243         return add(a, b);
244     }
245
246     static bfp_t subtract(const bfp_t& a, const bfp_t& b) {
247         unsigned i;
248         exp_t exp_val,interm_exp,a_shr,b_shr;
249         bfp_t c,d,result,result_temp;
250         ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot>
251             interm[num_elements];
252         //ap_fixed<W+1,I+1> interm[num_elements];
253         if(a.exp<b.exp){
254             interm_exp=b.exp+1;
255         }
256         else if(a.exp>b.exp){
257             interm_exp=a.exp+1;
258         }
259         else{
260             interm_exp=0;
261         }
262         a_shr=interm_exp-a.exp;
263         b_shr=interm_exp-b.exp;
264         denomalize(c,a,a_shr);
265         denomalize(d,b,b_shr);
266         for(i = 0; i < num_elements; ++i) {
267             interm[i] = c.data[i] - d.data[iif(exp_val==0){
271             for(i = 0; i < num_elements; ++i) {
272                 result_temp.data[i] = interm[i];
273             }
274             renomalize(result_temp);
275             result.exp=result_temp.exp+interm_exp;
276             for(i = 0; i < num_elements; ++i) {
277                 result.data[i] = result_temp.data[i];
278             }
279         }
280         else{
281             for(i = 0; i < num_elements; ++i) {
282                 result.data[i] = interm[i]>>exp_val;
283                 result.exp=1+interm_exp;
284             }
285         }
286         result.length = num_elements;
287         return result;
288     }

```

```
288
289     friend bfp_t operator-(const bfp_t& a, const bfp_t& b) {
290         return subtract(a, b);
291     }
292
293     static bfp_t multiply(const bfp_t& a, const bfp_t& b) {
294         unsigned i;
295         exp_t exp_val, interm_exp;
296         bfp_t c,d,result;
297         ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot>
298             interm[num_elements];
299         //ap_fixed<2*W,2*I> interm[num_elements];
300         interm_exp=a.exp+b.exp;//new shared exp
301         for (i = 0; i < num_elements; ++i) {
302             interm[i] = a.data[i] * b.data[i];
303
304             exp_val=check_overflow(2,interm);
305             if(exp_val==0){
306                 for(i = 0; i < num_elements; ++i) {
307                     result.data[i] =interm[i];
308                 }
309                 renormalize(result);
310                 result.exp=result.exp+interm_exp;
311
312             }
313             else{
314                 for(i = 0; i < num_elements; ++i) {
315                     result.data[i] =interm[i]>>exp_val;
316                 }
317                 result.exp=interm_exp+exp_val;
318             }
319             result.length = num_elements;
320             return result;
321
322         }
323     friend bfp_t operator*(const bfp_t& a, const bfp_t& b) {
324         return multiply(a, b);
325     }
326
327     static bfp_dot_product_t dot_product(const bfp_t& a, const bfp_t& b) {
328         unsigned i;
329         bfp_dot_product_t result;
330         exp_t exp_val=0,interm_exp;
331         //constexpr unsigned int I_test_value_dot = compute_ITest();
332         ap_fixed<2*W+I_test_value_dot ,2*I+I_test_value_dot> interm[1];
333         //ap_fixed<2*W+I_test_value ,2*I+I_test_value> interm_test=0;
334         interm_exp=a.exp+b.exp;
335         for (i = 0; i < num_elements; ++i) {
336             interm[0] += a.data[i] * b.data[i];
337             //interm_test=interm>>16;
338             //std::cout << interm_test << " ";
339         }
340         exp_val=check_overflow(3,interm);
341         if(exp_val==0){
342             result.data =interm[0];
343             renormalize_dot_product(result);
344             result.exp=result.exp+interm_exp;
345         }
```

```

346     else{
347         result.data = interm[0]>>exp_val;
348         result.exp=interm_exp+exp_val;
349     }
350     return result;
351 }
352 static bfp_dot_product_t dot_product_m(const mantissa_t
353 data1[num_elements], const mantissa_t data2[num_elements],
354 const exp_t exp1, const exp_t exp2, int start1, int start2,int
355 MATRIX_SIZE) {
356     unsigned i;
357     bfp_dot_product_t result;
358     exp_t exp_val=0,interm_exp;
359     //constexpr unsigned int I_test_value_dot = compute_ITest();
360     ap_fixed<2*W+I_test_value_dot,2*I+I_test_value_dot> interm[1];
361     //ap_fixed<2*W+I_test_value,2*I+I_test_value> interm_test=0;
362     interm_exp=exp1+exp2;
363     for (i = 0; i < MATRIX_SIZE; ++i) {
364         interm[0] += data1[i+start1] * data2[i+start2];
365         //interm_test=interm>>16;
366         //std::cout << interm_test << " ";
367     }
368     exp_val=check_overflow(3,interm);
369     if(exp_val==0){
370         result.data = interm[0];
371         renormalize_dot_product(result);
372         result.exp=result.exp+interm_exp;
373     }
374     else{
375         result.data = interm[0]>>exp_val;
376         result.exp=interm_exp+exp_val;
377     }
378     return result;
379 }
380 void to_double_array(const mantissa_t data,const exp_t exp, double&
381 result_array) {
382     double data_double,exp_double,aa;
383     data_double=data.to_double();
384     exp_double=exp.to_double();
385     result_array =data_double*pow(2,exp_double);
386 }
387 void to_double_array_1(const bfp_t& a, double result_array[num_elements])
388 {
389     double data_double,exp_double;
390     exp_double=a.exp.to_double();
391     for (int i = 0; i < num_elements; ++i){
392         data_double=a.data[i].to_double();
393         result_array[i] =data_double*pow(2,exp_double);
394     }
395 }
396 };

```

Listing 13: matrix_multiply.h

```

1 #define MATRIX_SIZE 7
2 #define ARRAY_SIZE 50
3
4 using BfpMatrix = bfp_fpga<7, 16, 32, 16>;//MATRIX_SIZE=50
5 using BfpFir = bfp_fpga<4,5,13,5>;

```

```

6 BfpMatrix::bfp_t init(double a[]);
7 BfpFir::bfp_t init_fir(double a[]);
8 BfpMatrix::bfp_dot_product_t matrix_multiply(BfpMatrix::bfp_t
    bfp_a,BfpMatrix::bfp_t bfp_b);
9 BfpFir::bfp_dot_product_t FIR(BfpFir::bfp_t bfp_a,BfpFir::bfp_t bfp_b);

```

Listing 14: matrix_multiply.cpp

```

1 #include "bfp_fpga.h"
2 #include "matrix_multiply.h"
3
4
5 BfpMatrix::bfp_t init(double a[]) {
6     BfpMatrix bfp_matrix_multiply;
7     BfpMatrix::bfp_t A_matrix_data,bfp_a;
8     bfp_matrix_multiply.receive_double(A_matrix_data, a);
9     bfp_matrix_multiply.bfp_init(bfp_a, A_matrix_data);
10    return bfp_a;
11 }
12 BfpMatrix::bfp_dot_product_t matrix_multiply(BfpMatrix::bfp_t
13     bfp_a,BfpMatrix::bfp_t bfp_b){
14     BfpMatrix::bfp_dot_product_t C_matrix_data;
15     BfpMatrix bfp_matrix_multiply;
16     C_matrix_data = bfp_matrix_multiply.dot_product(bfp_a,bfp_b);
17     return C_matrix_data;
18     //bfp_matrix_multiply.to_double_array(C_matrix_data.data,C_matrix_data.exp,
19     //array);
20 }
21 BfpFir::bfp_t init_fir(double a[]) {
22     BfpFir bfp_fir;
23     BfpFir::bfp_t A_fir_data,bfp_a;
24     bfp_fir.receive_double(A_fir_data, a);
25     bfp_fir.bfp_init(bfp_a, A_fir_data);
26     return bfp_a;
27 }
28 BfpFir::bfp_dot_product_t FIR(BfpFir::bfp_t bfp_a,BfpFir::bfp_t bfp_b) {
29     BfpFir fir_bfp;
30     BfpFir::bfp_dot_product_t output_data;
31     output_data = fir_bfp.dot_product(bfp_a,bfp_b);
32     return output_data;
33     //fir_bfp.to_double_array(output_data.data,output_data.exp, array);
}

```

Listing 15: testbench.cpp

```

1 #include "bfp_fpga.h"
2 #include "matrix_multiply.h"
3 #include <hls_math.h>
4 #include <ap_int.h>
5 #include <ap_fixed.h>
6 #include <iostream>
7 void generateRandomMatrix(double matrix[MATRIX_SIZE][MATRIX_SIZE]) {
8     for (int i = 0; i < MATRIX_SIZE; i++) {
9         for (int j = 0; j < MATRIX_SIZE; j++) {
10             matrix[i][j] = -100.0 + static_cast<double>(rand()) /
11                 (static_cast<double>(RAND_MAX/200.0));
12         }
13     }
}

```

```
14 void generateRandomdoublearray(double a[ARRAY_SIZE]) {
15     for (int i = 0; i < ARRAY_SIZE; i++) {
16         a[i] = -100.0 + static_cast<double>(rand()) /
17             (static_cast<double>(RAND_MAX/200.0));
18     }
19 }
20 void printMatrix(const double matrix[MATRIX_SIZE][MATRIX_SIZE]) {
21     for (int i = 0; i < MATRIX_SIZE; i++) {
22         for (int j = 0; j < MATRIX_SIZE; j++) {
23             std::cout << std::setw(10) << std::fixed << matrix[i][j] << " ";
24         }
25         std::cout << std::endl;
26     }
27 }
28 int main() {
29     std::srand(static_cast<unsigned int>(std::time(nullptr)));
30
31     double Amatrix[MATRIX_SIZE][MATRIX_SIZE];
32     double Bmatrix[MATRIX_SIZE][MATRIX_SIZE];
33
34     generateRandomMatrix(Amatrix);
35     generateRandomMatrix(Bmatrix);
36     std::cout << "\nAmatrix:" << std::endl;
37     printMatrix(Amatrix);
38
39     std::cout << "\nBmatrix:" << std::endl;
40     printMatrix(Bmatrix);
41     //double input_A[MATRIX_SIZE], input_B[MATRIX_SIZE];
42     double Cmatrix_bfp[MATRIX_SIZE][MATRIX_SIZE] = {0};
43     //double array1[MATRIX_SIZE][MATRIX_SIZE]={0};
44     /* double Amatrix[7][7] = {
45         {1, 1, 1, 1, 1, 1, 1},
46         {1, 1, 1, 1, 1, 1, 1},
47         {1, 1, 1, 1, 1, 1, 1},
48         {1, 1, 1, 1, 1, 1, 1},
49         {1, 1, 1, 1, 1, 1, 1},
50         {1, 1, 1, 1, 1, 1, 1}
51     };
52
53     double Bmatrix[7][7] = {
54         {1, 1, 1, 1, 1, 1, 1},
55         {1, 1, 1, 1, 1, 1, 1},
56         {1, 1, 1, 1, 1, 1, 1},
57         {1, 1, 1, 1, 1, 1, 1},
58         {1, 1, 1, 1, 1, 1, 1},
59         {1, 1, 1, 1, 1, 1, 1},
60         {1, 1, 1, 1, 1, 1, 1}
61     };*/
62     double Cmatrix[MATRIX_SIZE][MATRIX_SIZE];
63     BfpMatrix::bfp_t bfpa;
64     BfpMatrix::bfp_t bfpb;
65     BfpMatrix::bfp_dot_product_t C_matrix_data;
66     double array[MATRIX_SIZE][MATRIX_SIZE]={0};
67     double result_array=0;
68     BfpMatrix bfp_matrix_multiply;
69     double input_A[MATRIX_SIZE]={0}, input_B[MATRIX_SIZE]={0};
70     for(int i = 0; i < MATRIX_SIZE; ++i) {
71         for(int k = 0; k < MATRIX_SIZE; ++k) {
```

```

72     input_A[k] = Amatrix[i][k];
73 }
74 for(int j = 0; j < MATRIX_SIZE; ++j) {
75     for(int k = 0; k < MATRIX_SIZE; ++k) {
76         input_B[k] = Bmatrix[k][j];
77     }
78     BfpMatrix::bfp_t bfpa = init(input_A);
79     BfpMatrix::bfp_t bfpb = init(input_B);
80     C_matrix_data = matrix_multiply(bfpa, bfpb);
81     //std::cout << "result_array1111: " << result_array << std::endl;
82     bfp_matrix_multiply.to_double_array(C_matrix_data.data, C_matrix_data.exp,
83                                         result_array);
84     array[i][j] = result_array;
85 }
86 }
87 double max_diff = 0.0;
88 double sum_diff = 0.0;
89 for (int i = 0; i < MATRIX_SIZE; ++i) {
90     for (int j = 0; j < MATRIX_SIZE; ++j) {
91         for (int k = 0; k < MATRIX_SIZE; ++k) {
92             Cmatrix[i][j] += Amatrix[i][k] * Bmatrix[k][j];
93         }
94     }
95 }
96 std::cout << "matrix_C_computed_by_fp:\n";
97 for (int i = 0; i < MATRIX_SIZE; ++i) {
98     for (int j = 0; j < MATRIX_SIZE; ++j) {
99         std::cout << std::setw(10) << std::fixed << Cmatrix[i][j] << " ";
100    }
101    std::cout << std::endl;
102 }
103 std::cout << "matrix_C_computed_by_bfp:\n";
104 for (int i = 0; i < MATRIX_SIZE; ++i) {
105     for (int j = 0; j < MATRIX_SIZE; ++j) {
106         std::cout << std::setw(10) << std::fixed << array[i][j] << " ";
107     }
108     std::cout << std::endl;
109 }
110
111 double diff[MATRIX_SIZE][MATRIX_SIZE];
112 std::cout << "different_between_bfp_and_fp:\n";
113 for (int i = 0; i < MATRIX_SIZE; ++i) {
114     for (int j = 0; j < MATRIX_SIZE; ++j) {
115         diff[i][j] = array[i][j] - Cmatrix[i][j];
116         max_diff = std::max(max_diff, abs(diff[i][j]));
117         sum_diff += abs(diff[i][j]);
118         std::cout << std::setw(10) << std::fixed << abs(diff[i][j]) << " ";
119     }
120 }
121
122 std::cout << "\nMax_difference: " << max_diff << std::endl;
123 std::cout << "Average_difference: " << sum_diff /
124     (MATRIX_SIZE*MATRIX_SIZE) << std::endl;
125 std::cout << "-----"
126     ADD-SUB-----" << std::endl;
127 double a[ARRAY_SIZE];

```

```
127     double b[ARRAY_SIZE];
128     double c[ARRAY_SIZE];
129     double c_from_bfp[ARRAY_SIZE];
130     double diff_add[ARRAY_SIZE];
131     max_diff = 0.0;
132     sum_diff = 0.0;
133     generateRandomdoublearray(a);
134     generateRandomdoublearray(b);
135     std::cout << "ADD result C computered by fp:\n";
136     for (int i = 0; i < ARRAY_SIZE; ++i) {
137         c[i]=a[i]+b[i];
138         std::cout << std::setw(10) << std::fixed << c[i] << " ";
139     }
140     std::cout << std::endl;
141     bfp_fpga <ARRAY_SIZE,16,32,16> bfp;
142     bfp_fpga <ARRAY_SIZE,16,32,16>::bfp_t
143         A_double,B_double,result,bfp_a,bfp_b;
144     bfp.receive_double(A_double, a);
145     bfp.receive_double(B_double, b);
146     bfp.bfp_init(bfp_a, A_double);
147     bfp.bfp_init(bfp_b, B_double);
148     result=A_double+B_double;
149     std::cout << "ADD result C computered by bfp:\n";
150     bfp.to_double_array_1(result,c_from_bfp);
151     for (int i = 0; i < ARRAY_SIZE; ++i) {
152         std::cout << std::setw(10) << std::fixed << c_from_bfp[i] << " ";
153     }
154     std::cout << std::endl;
155     std::cout << "different between fp and bfp:\n";
156     for (int i = 0; i < ARRAY_SIZE; ++i) {
157         diff_add[i]=c_from_bfp[i]-c[i];
158         max_diff = std::max(max_diff, abs(diff_add[i]));
159         sum_diff += abs(diff_add[i]);
160         //if(abs(diff_add[i])>0.0001){
161             std::cout << std::setw(10) << std::fixed << abs(diff_add[i]) << " ";
162         //}
163     }
164     std::cout << "\nMax difference: " << max_diff << std::endl;
165     std::cout << "Average difference: " << sum_diff / ARRAY_SIZE <<
166         std::endl;
167     max_diff = 0.0;
168     sum_diff = 0.0;
169     diff_add[ARRAY_SIZE]=0;
170     std::cout << "SUB result C computered by fp:\n";
171     for (int i = 0; i < ARRAY_SIZE; ++i) {
172         c[i]=a[i]-b[i];
173         std::cout << std::setw(10) << std::fixed << c[i] << " ";
174     }
175     std::cout << std::endl;
176     result=A_double-B_double;
177     std::cout << "SUB result C computered by bfp:\n";
178     bfp.to_double_array_1(result,c_from_bfp);
179     for (int i = 0; i < ARRAY_SIZE; ++i) {
180         std::cout << std::setw(10) << std::fixed << c_from_bfp[i] << " ";
181     }
182     std::cout << std::endl;
183     std::cout << "different between fp and bfp:\n";
184     for (int i = 0; i < ARRAY_SIZE; ++i) {
185         diff_add[i]=c_from_bfp[i]-c[i];
```

```

184     max_diff = std::max(max_diff, abs(diff_add[i]));
185     sum_diff += abs(diff_add[i]);
186     //if(abs(diff_add[i])>0.0001){
187     std::cout << std::setw(10) << std::fixed <<abs(diff_add[i]) << " ";
188     //}
189 }
190 std::cout << "\nMax difference:" << max_diff << std::endl;
191 std::cout << "Average difference:" << sum_diff / ARRAY_SIZE <<
192     std::endl;
193 std::cout << "-----";
194 FIR-----" << std::endl;
195 double input[10] = {10.0, 9.0, 11.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0,
196     1.0};
197 double coeffs[4] = {0.1, 0.2, 0.3, 0.4};
198 double output1[10];
199 double output0;
200 double receive_h[4]= {0.0};
201 double receive_x[4]= {0.0};
202 max_diff = 0.0;
203 sum_diff = 0.0;
204 BfpFir::bfp_dot_product_t output_data;
205 BfpFir::bfp_t receive_h_data,receive_x_data,bfp_h[10],bfp_x[10];
206 BfpFir fir_bfp;
207 for (int i = 0; i < 10; ++i) {
208     int index = 0;
209     for (int j = 0; j < 4; ++j) {
210         if (i - j >= 0) {
211             receive_h[index] = coeffs[j];
212             receive_x[index] = input[i - j];
213             //std::cout << "receive_h[index]: " << receive_h[index] <<
214                 std::endl;
215             //std::cout << "receive_x[index]: " << receive_x[index] <<
216                 std::endl;
217             index++;
218         }
219     }
220     BfpFir::bfp_t bfpa=init_fir(receive_h);
221     BfpFir::bfp_t bfpb=init_fir(receive_x);
222
223     output_data=FIR(bfpa,bfpb);
224     fir_bfp.to_double_array(output_data.data,output_data.exp, output0);
225     output1[i] = output0;
226 }
227 std::cout << std::endl;
228
229 double output[10];
230 for (int i = 0; i < 10; ++i) {
231     output[i] = 0.0;
232 }
233
234 for (int i = 0; i < 10; ++i) {
235     for (int j = 0; j < 4; ++j) {
236         if (i - j >= 0) {
237             output[i] += coeffs[j] * input[i - j];

```

```

238         }
239     }
240 }
241 std::cout << "Output_exponentialFP:" ;
242 for (int i = 0; i < 10; ++i) {
243     std::cout << output[i] << " ";
244 }
245 std::cout << std::endl;
246 max_diff = 0.0;
247 sum_diff = 0.0;
248 diff_add[10]=0;
249 for (int i = 0; i < 10; ++i) {
250     diff_add[i]=output1[i]-output[i];
251     max_diff = std::max(max_diff, abs(diff_add[i]));
252     sum_diff += abs(diff_add[i]);
253     //if(abs(diff_add[i])>0.0001){
254     std::cout << std::setw(10) << std::fixed <<abs(diff_add[i]) << " ";
255     //}
256 }
257 std::cout << "\nMax_difference:" << max_diff << std::endl;
258 std::cout << "Average_difference:" << sum_diff / 10 << std::endl;
259 }
```

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