Design and Implementation of an Improved Floating Point Adder Unit on FPGA

Author: stud. Adrian-Răzvan Deaconescu, C3 351

September 4, 2006

Contents

Chapter 1

Introduction

Advances in FPGA (Field Programmable Gate Array) chip design have lead to the possibility of using them for floating point arithmetic. Floating point arithmetic is quite complex and only recently has it found its way into FPGA, due to the improvement both on clock rate and circuit space.

We have implemented an improved floating point adder unit that works on FPGAs. We have been using Celoxica DK Suite and the Celoxica's RC2000 (ADMXRC2) board. The board is using Xilinx Virtex II xc2v6000 chip. We have done our implementation using Celoxica's Handel-C, a high level hardware design language, and a set of libraries and API provided by Celoxica's Platform Development Kit.

While working on implementing the floating point adder unit we have also implemented a framework for testing the floating point adder unit. At the same time we have also implemented a C program that can generate an configuration-dependent Handel-C project for implementing the adder. The Handel-C project can be customized using a configuration file to use special circuit parts and also various exponent/mantissa widths.

Chapter 2

Components

As mentioned above, the project contains both the Handel-C implementation but also a testing framework. What follows is a detailed list of what the project consists of.

2.1 Random input generation

A binary input file can be obtained using a specialized C program. This input file will subsequently be used to test the actual implementation. The C program can be specified the type of floating point numbers that are used and the number of input pairs to be generated.

2.2 Correctness checking

A C program is used to verify whether the results the FPGA has computed are correct. For that they are compared against the results computed on the host (the PC); that means that we are using the PC's internal floating point unit to check the results for correctness.

2.3 RAM bitstream loading

There are two associated programs: a C program and a Handel-C program. They are used to load data into RC2000 board's ZBT RAM. The data is usually read from a binary file generated with a specialized C program. Both programs use the DMA API provided with ADMXRC2-SDK (for the C program) and Celoxica's Platform Development Kit (PDK) (for the Handel-C program). The two programs are also used to read data from the board's ZBT RAM into a binary output file.

2.4 "Cover as many possible cases" input generation

Two C programs are used to generate a binary input file for testing. Their main aim is to generate a binary input file that should cover as many input cases as possible. This would mean breaking down the input data into several classes: zero, denormalized number, close-to-denormalized normalized number, average normalized number, close-to-infinity normalized number, infinity, not-a-number (NaN). Then we would combine these classes to get all possible pairs of input data. One of the program generates a text file (human-readable); the second works in a pipeline with the first one and reads the text file to generate a binary one. This allows a greater degree of flexibility, as one can generate a human-readable input file (with number as he/she wishes) and use the second program to generate a binary file.

2.5 Floating Point Adder Unit implementation

The implementation consists of a Handel-C program and the associated C program. The C program uses the ADMXRC2-SDK to load the implementation bitstream file on the RC2000 board and handle the communication between it and the host. The Handel-C program is using the floating point implementation on the data from the ZBT RAM; after computing the data it writes the result back in memory, from where it can be later retrieved.

2.6 Handel-C implementation generation

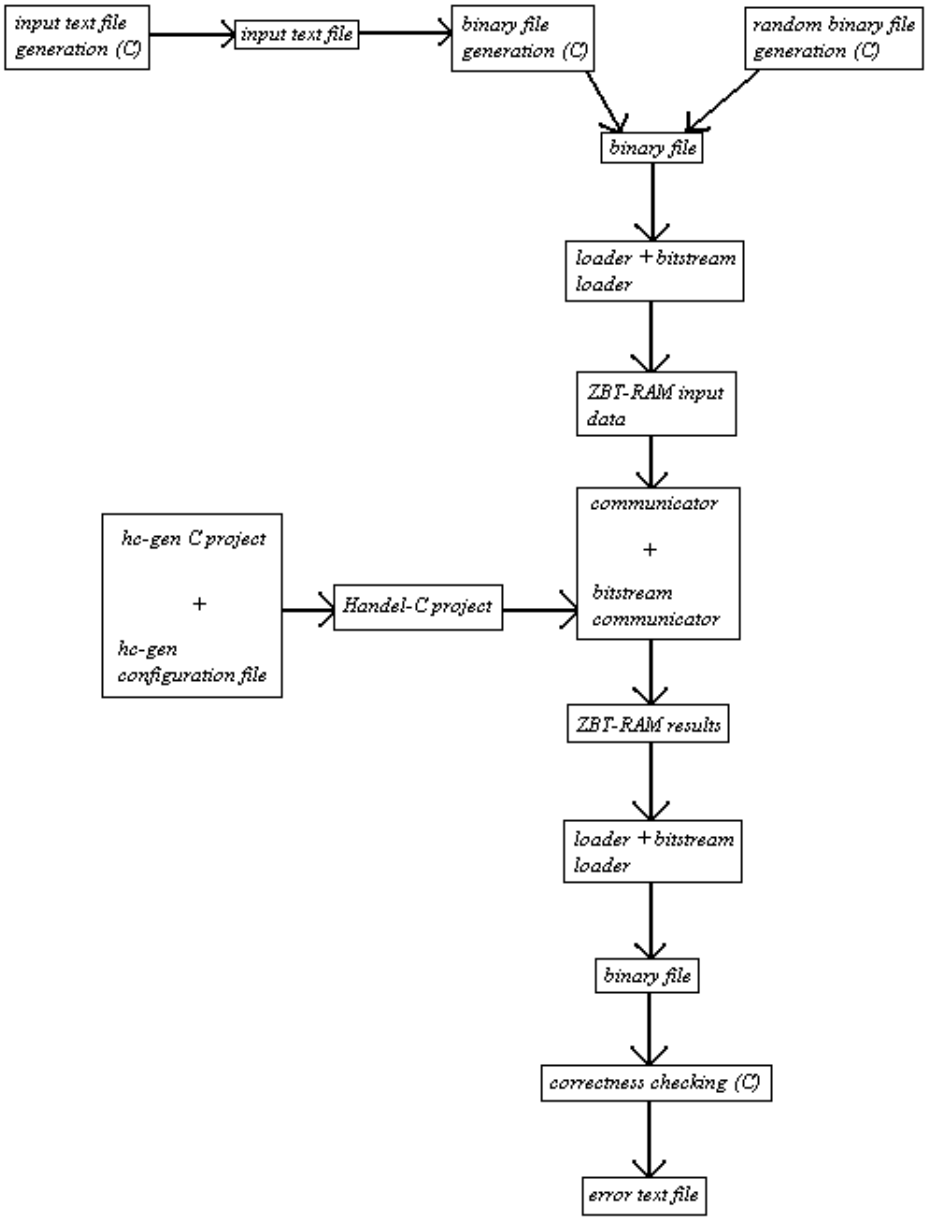
Several C files and a configuration file create a project that is designed to generate the Handel-C files required for the floating point adder unit implementation. The aim is to create a configuration-enabled project. We could thus select the bit-width of the exponent, the bit-width of the mantissa, the types of adders that we use and many other things, etc. Some configurations may be suitable for a particular exponent/mantissa bit-width, some for another (in terms of speed and circuit space complexity).

2.7 One time testing

A bash script is used to do handle the implementation testing. It presumes all the bitstream files have been generated (thus it also assumes that we have used the Handel-C implementation generation and then compiled and placed & routed it into a bitstream). This script will automatically generate an input test file, will load the data into the ZBT RAM, it will load the implementation bitstream to do the floating point computation, retrieve the results and afterward do the checking to see whether the implementation is flawless. It has a series of internal variables that can be used as configuration options. Some of them can be passed as command-line arguments.

2.8 Data flow

The following graphics show the way the components are working together. We present the project flow from how input data is generated to how the results obtained are compared against the PC generated results (which are assumed correct).



Most of the flow is included in the bash script that makes it easy to do a

test case.

2.8.1 Details of the data flow

The *input text file generation* and *binary file generation* are the two C programs that create a *binary input file* considering as many possible cases as possible (as stated in Section **??**). We created two programs for the sake of flexibility (but they can be "piped" together to work as a single entity). The first one outputs an *input text file*, while the second outputs a *binary input file*.

The *random binary file generation* is a C program that creates a *binary input file* with random data (as stated in Section **??**). The *binary input file* contains pairs of input data that would be loaded on the ADMXRC2 board's RAM banks for testing.

The *loader + bitstream loader* are a C program and its associated Handel-C program that are used to upload/download data to/from the RAM banks (as stated in Section **??**). Their first appearance in the graphics marks uploading of data from the *binary input file* to the RAM to form the *ZBT-RAM input data*. This data is going to be used test the floating point adder unit. The *loader* has a number of command line arguments; one of them specifies the *bitstream* that coordinates the upload. This is actually the .bit file as the final result of the Handel-C program compile and place & route steps.

The *communicator + communicator bitstream* are a C program and a Handel-C program. The C program (just like the *loader* from above) uses the ADMXRC2-SDK to load a bitstream onto the ADMXRC2 board and direct its work. The communicator bitstream is the result of the compile and place & route steps on the Handel-C program. The Handel-C program contains the actual files that implement the floating point adder unit (as stated in Section **??**). The Handel-C program can be written from scratch, or it can be generated from a C program. This is the job of the *hc-gen C program*: a C program that reads a *hc-gen configuration file* and outputs a series of Handel-C files (the *Handel-C project*) implementing a configuration-dependent floating point adder unit (as stated in Section **??**).

The bitstream program has a simple job: it reads a pair of data from the memory, does the floating point addition and then stores the result back in the memory (by memory we are referring to the ZBT-RAM banks of the ADMXRC2 board). The *ZBT-RAM results* are subsequently read by the *loader + bitstream loader* and stored in a *binary file*. This *binary file* thus contains the results of the floating point adder unit having its input from the initial *binary input file*.

A C program is used to handle the *correctness checking* (as stated in Section **??**). This C program will receive as input the initial *binary input file* and the *binary file* containing the results. It will do a floating point addition using the host CPU (PC) floating point unit and compare the results against those of the Handel-C implementation. The results will be stored in an *error text file* (human readable). The user can thus find out whether there are cases when the implementation does not work or whether the running frequency has been set up too high.

What we have not mentioned in the graphics is the fact that we could run a Handel-C program in **Debug Mode**. This would be mean that we wouldn't have to load the bitstream on the board but rather do some small testing using

a simulator. As its non-debug mode equivalent, the Handel-C program can be written from scratch or it can be generated from a C program.

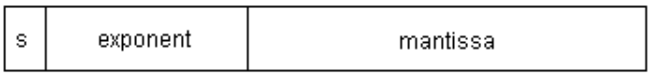
Chapter 3

Floating point numbers

Fractionary (non-integer) numbers may be represented as fixed point numbers or floating point numbers. In case of fixed point numbers, one bit is reserved for the sign; if that bit is 1 the number is negative, otherwise it is positive. The rest of the bits are split between the integer part of the number and the fractionary part. Thus the number would actually be $num = (-1) * sign * (int + 0.fract)$.

There are obvious defficiencies with fixed point number representation: limited fractionary and integer part, very limited range, integer and fractionary part bit widths are not standard. This is why most applications use floating point numbers. A floating point number brings the term of *exponent*. The representation would generate a number that has the value of $(-1) * sign * base^{exponent}$. As we are working with binary representation, the base is usually 2. In its internal representation, the base is also called a *mantissa*.

A typical representation for a floating point number is shown below:



s is the sign and has 1 bit width. Bit width for the exponent and the mantissa may vary. Of course, the order of the three fields may vary.

There are two types of representation for floating point numbers: standard compliant representation (IEEE 754, IEEE 754r) and non-standard compliant (also called arbitrary precision). Arbitrary precision deals with different bit widths for the exponent and the mantissa making the application non-portable. Our implementation may deal with arbitrary precision representation, as we can generate a Handel-C program using a C program and a configuration file that can specify the bit widths. Emphasis is given, however, to standard compliant (IEEE 754) representation.

3.1 Floating point number standard - IEEE 754

Before IEEE decided to standardize the floating point number representation, there were various other representations that had some inconsistencies: subtracting a number from another would not be the reverse of subtracting the second number from the first, there was no way to represent an error (like di-

vision by zero). IEEE decided to create a standard that would remove all the inconsistencies and give a more natural and easy to use floating point number representation.

IEEE 754 defines a floating point number as having three fields:

- **sign** - 1 bit with; 1 signifies a negative number and 0 signifies a positive number
- **exponent** - 8 bit width for single precision (32 bit representation) and 11 bit width for double precision (64 bit representation)
- **mantissa** - 23 bit width for single precision and 52 for double precision

The order of the three fields in the binary representation is sing, exponent, mantissa.

Aiming to create a unified and consistent representation, IEEE defines classes of floating point numbers. These classes are used to represent a variety of possible numbers, including infinities and NaNs (Not a Number). The exponent is the criterion for the classification. In order to represent both small and large numbers the exponent is biased. The bias is subtracted from the binary representation and the result is the real value for the exponent. The bias is $2^{len-1} - 1$where *len* is the bit width of the exponent. That means that the bias is $2^7 - 1 = 127$for single precision and $2^{11-1} - 1 = 1023$for double precision.

The IEEE 754 classes of numbers are:

- **normalized numbers**: normalized numbers have values for the exponent binary representation between 1 and $2^{len} - 2$; the biased value is between $-(2^{len-1} - 2)$and$2^{len-1} - 1$, where len is the bit width of the exponent; the mantissa (fraction) has a hidden one that is not shown in its representation; that means that the value of the floating point number is $val = (-1)^s * 1.m * 2^{e-bias}$, where s is the sign, m is the mantissa (fraction) and e is the exponent.
- **denormalized numbers**: denormalized numbers have a zero exponent, the real value is $denorm\_bias = -(2^{len-1} - 2)$; there is no hidden one for the mantissa; this means that the value is $val = (-1)^s * 0.m * 2^{denorm\_bias}$
- **zero**: zero has "all bits zero" exponent and "all bits zero" mantissa; there are two zeros: a positive zero (when sign is 0) and negative zero (when sign is 1)
- **infinities**: there are two infinities: $-\infty$and $+\infty$; infinities have an exponent whose binary value is $2^{len} - 1$ and an "all bits zero" mantissa
- **NaN**: NaNs have a representation close to the infinities, except for the fact that the mantissa has to be different from zero

3.1.1 Single precision

Single precision floating point numbers are represented on 32 bits. The exponent has 8 bit width and the mantissa has 23 bit width (1 + 8 + 23). The range of numbers that can be represented is $(-2^{-126-22}; 2 * 2^{127})$.The classes of floating point numbers are as follows:

- **normalized numbers**: the bias is $2^{8-1} - 1 = 127$; the range of the exponent is thus $[-126; 127]$, while its binary value is in the range $[1; 254]$
- **denormalized numbers**: the exponent is -126, while its binary value is 0 (as is the case for denormalized numbers)
- **infinities & NaN**: these special representation have a binary value of $2^8 - 1 = 255$for the exponent (all ones)

3.1.2 Double precision

Single precision floating point numbers are represented on 64 bits. The exponent has 11 bit width and the mantissa has 52 bit width (1 + 11 + 52). The range of numbers that can be represented is $(-2^{-1022-52}; 2*2^{1023})$.The classes of floating point numbers are as follows:

- **normalized numbers**: the bias is $2^{11-1} - 1 = 1023$; the range of the exponent is thus $[-1022; 1023]$, while its binary value is in the range $[1; 2046]$
- **denormalized numbers**: the exponent is -1022, while its binary value is 0 (as is the case for denormalized numbers)
- **infinities & NaN**: these special representation have a binary value of $2^{11} - 1 = 2047$for the exponent (all ones)

3.1.3 Quadruple precision

Quadruple precision is not very often used because of the lack of 128 bit processing units. However, due to recent advances in chip design, numbers in quadruple precision start finding their way in applications. The exponent has 15 bits, and the mantissa has 112 bits.

3.2 Floating point operations

It is visibly harder to implement floating point operations than integer operations. The breaking of the binary representation into three fields (sign, exponent, mantissa) and the various classes of numbers require additional circuitry and analysis. A lot of research has been (and still is) employed in improving the efficiency of floating point operations. These research aim at using less circuitry, improving speed, or designing pipelined operations; pipelined operations require breaking the basic floating point operation (addition, multiplication, etc.) into stages and executing each stage in one cycle. This requires multpiple floating point units working in parallel, but recent advances in FPGA and ASIC design have made all this easily achievable.

The basic operations dealing with floating point numbers are addition, subtraction, multiplication and division. Any other complex operation can be broken down to these simple operations. Addition and subtraction are implemented using the same hardware unit as only the sign differs. Division is typically more complex and left out as few applications really need division (and most of those that need it use different algorithms to leave division out). A typical operation is the multiply and add fused (MAF) who uses both an addition circuit and a multiplication circuit. It is a fairly frequent operation in signal processing.

3.2.1 Addition/subtraction

Addition and subtraction are implemented in the same circuit as they are basically the same operation. Addition of two numbers having different signs is a subtraction.

The basic algorithm is as follows:

1. compare the exponents and find the exponent difference; the larger exponent is the exponent of the result
2. shift the smaller number's mantissa right by the proper amount (the exponent difference)
3. add or subtract the mantissas, depending on the signs of the two numbers
4. find out whether the result has to be normalize; if there is a series of leading zero bits, the mantissa of the result has to be shifted and the result exponent has to be corrected
5. obtain the final result; special cases are looked for here: infinities, NaNs, denormalized numbers are operators or result

As seen in the above algorithm, a normalization stage occurs after the addition. It is, however, possible to find out the shift amount required for the actual normalization concurrently with the addition (with the cost of extra circuitry). This is called LZA (Leading Zero Anticipation) or LOP (Leading One Prediction) and is employed in our floating point adder unit design.

3.2.2 Multiplication

Multiplication has a simpler algorithmic implementation than addition (there is no need for exponent comparison). The circuitry, however, has a higher degree of complexity because of the internal operations: integer multiplication (mantissa), when compared to integer addition. The steps for the algorithm are the following:

1. add the exponents; the result of the addition is the exponent of the result
2. multiply the mantissas
3. truncate the result and, eventually, adjust the exponent

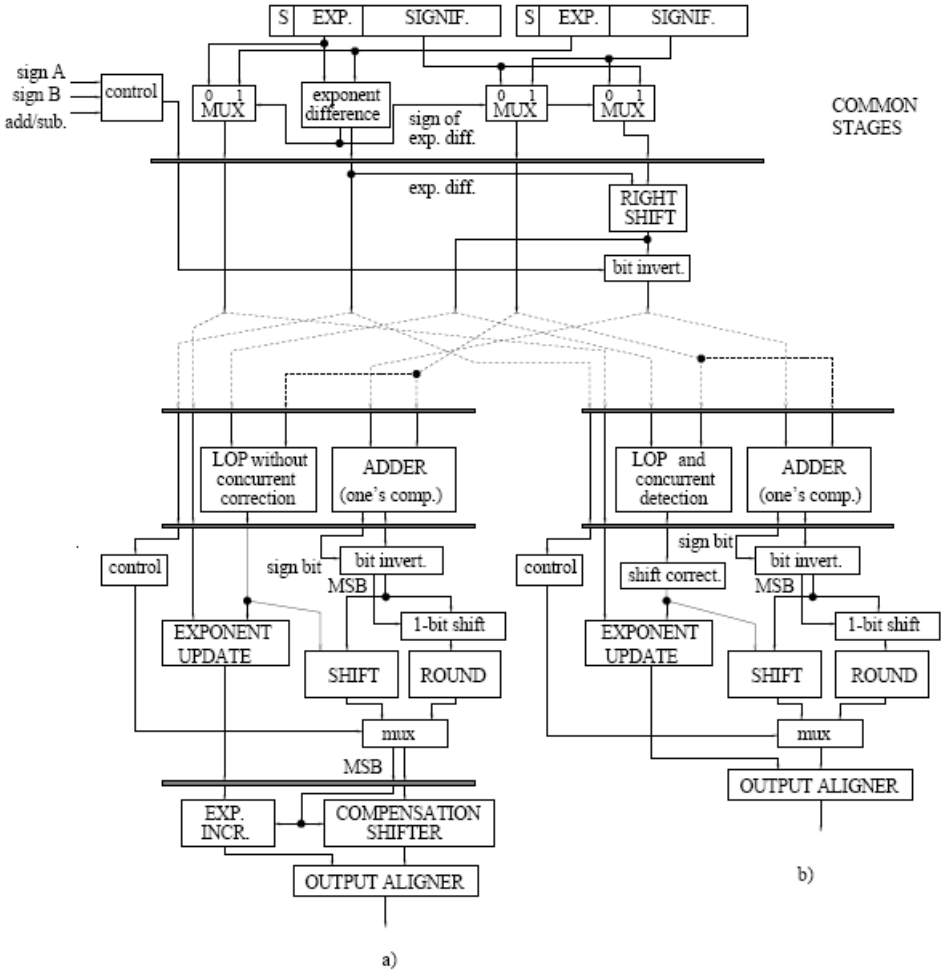
Because integer multiplication (of the mantissas) is the most complex stage of the floating point multiplication, a lot of optimizations occur at this part. FPGA chips typically ship with internal multipliers (Xilinx Virtex II chip includes a number of 144 18x18 bit multipliers). These multipliers are carefully tuned for maximum speed and space eficiency so that they can be used as they are without any need for reimplementation by the designer.

Chapter 4

Floating Point Adder

4.1 Floating Point Adder Unit structure

The following graphic shows the overall design of our floating point adder unit:



The structure that we have implemented is described by the b) variant in the graphics. That means that our LZA (Leading Zero Anticipation unit) or LOP (Leading One Predictor unit) also handles the concurrent correction (there is a potential 1 bit error). This is just an overall view of the implemented floating point adder structure; this means that during the implementation some stages have been tuned to get the most out of the design.

4.1.1 Floating point adder unit design

The above graphics was the starting point for our floating point adder unit design. We have improved the design and also adjusted it to particular requirements that resulted during the implementation. As it can be seen from the graphics, there are certain stages in the overall design. These stages could potentially be used for creating a pipelined implementation of the adder. We will walk through the stages and present the problems we faced with each particular stage and the solutions we found.

4.1.1.1 Stage 1 - Exponent comparison and mantissa adjustment

For an addition/subtraction we need to find out the greater number. This means we have to compare the exponents of the two floating point numbers. But we also need the difference between the two exponents for adjusting the two numbers to the same exponent. So we are subtracting the two exponents (the second exponent from the first one). As we are implementing subtraction as 1's complement addition, if the carry out is 1 the result is negative (which translates in the second number being greater); otherwise the result is positive (which translates in the first number being greater).

The operation is a subtraction if the signs of the two numbers are different. Otherwise the operation is an addition.

The signal being 1 or 0 can be used as the selection input for a multiplexer; it can choose the greater sign/exponent/mantissa for the subsequent operations.

We are also doing a zero comparison with each of the exponents; this is required for special cases (like denormalized numbers).

The mantissa is afterwards padded with a 1 bit prefix and the Round, Guard and Sticky bits as the suffix. The smaller mantissa is right shifted with the proper amount so that the the exponents will be adjusted to the same value.

4.1.1.2 Stage 2 - Mantissa addition and LZA

The padded mantissas are added or subtracted. While the addition is can be implemented fairly easy, there are issues when subtracting the mantissas. The mantissas are represented in sign magnitude. When implementing the subtraction we could be using 2's complement or 1's complement. 2's complement uses an entire circuit so this option is not suitable. On the other hand, when using 1's complement we need to adjust the result; if there is a carry out that has to be added to the result. We are using a modified carry-select adder (named improved dual carry select adder) to enable the subtraction of the two mantissas with highest degree of efficiency.

After the addition we are checking to see if there is an overflow of the result

(which would require a shift right); in case of the subtraction we are implementing the LZA (which could potentially require a shift left of the result).

4.1.1.3 Stage 3 - Normalization and exponent correction

The normalization step adjusts the mantissa of the result; a shift right may be required for an addition and a shift left for a subtraction.

In case of a subtraction the course shift value and the fine shift value have of the LZA to be added together for the final shift value.

After normalizing the result mantissa, we are also adjusting the result exponent.

The third stage is also concerned with denormalized numbers. This means additional complexity as denormalized numbers have a value of -126 for the exponent (with a 0 representation) for single precision numbers (8 bit width exponent).

4.1.1.4 Stage 4 - Output alignment

The last stage considers all cases: dealing with denormalized numbers, dealing with infinities, NaNs. It considers when we are doing an addition and when we are doing a subtraction.

4.1.2 Floating point adder unit implementation

We will be presenting the adder unit implementation closely related to the design stages. For each stage, part of the Handel-C source code will be shown and discussed.

4.1.2.1 Stage 1 - Exponent comparison and mantissa adjustment

The floating point adder (like everyrhing else) is implemented as a macro. This is required in order to obtain a pure combinational - non-sequencial - circuit.

The macro header is:

```
#define FLOAT(ExpWidth,MantWidth)                                          \
        struct {
                unsigned int (1)                        Sign;
                unsigned int (ExpWidth)         Exponent;                  \
                unsigned int (MantWidth)        Mantissa;                  \
        }

#endif


/*
 * Floating Point Addition
 */

extern macro proc fp_add (xx1, xx2, result);
```

```
/* first exponent is zero and second exponent is zero signals */
e1_zero = (x1.Exponent == 0);
e2_zero = (x2.Exponent == 0);

/* simple ripple carry subtraction; the second operand is in 1's complemen
rc_add (ZERO @ x1.Exponent, ONE @ ~(x2.Exponent), esigndiff, ZERO);

// multiplexer choosing the largest/smallest exponent
// esigndiff[E] will be the sign of the exponent subtraction; if it is one
// means that the result is negative and that x2 is greater than x1
largeexp = esigndiff[E] ? x2.Exponent : x1.Exponent;
smallexp = esigndiff[E] ? x1.Exponent : x2.Exponent;

// same goes for mantissas

// sorting into larger and smaller
// NOTE: we are using these signals to relieve some useless recomputation
// in the next instructions (the ones that choose the smaller and larger m
// we are using the three standard guard bits at the end (GRS - Ground Rou
m1guarded = e1_zero ? ZERO @ x1.Mantissa[M-1] @ x1.Mantissa[M-2:0] @ ZERO
m2guarded = e2_zero ? ZERO @ x2.Mantissa[M-1] @ x2.Mantissa[M-2:0] @ ZERO

// get the largest mantissa (with guard bit, hidden bit and round bit) - 2
glargemant = esigndiff[E] ? m2guarded : m1guarded;
gsmallmant = esigndiff[E] ? m1guarded : m2guarded;

// get the exponent difference (as a positive value)
ediff = esigndiff[E] ? ~(esigndiff <- E) : (esigndiff <- E);

// compute the mantissa shift value (we have to have a SHIFT_LEN width) */
shiftvalue = get_shift8 (ediff);

// get mask to compute the Sticky bit
decode_fill_ones5 (shiftvalue, mask);

ma = gsmallmant & mask[27:0];

sticky_bit = ma[0] | ma[1] | ma[2] | ma[3] | ma[4] | ma[5] | ma[6] | ma[7]
                                ma[9] | ma[10] | ma[11] | ma[12] | ma[13]
                                ma[17] | ma[18] | ma[19] | ma[20] | ma[21]
                                ma[25] | ma[26] | ma[27];

gsmallmantshift_no_sticky = (gsmallmant \\ 1) >> shiftvalue;

gsmallmantshift = gsmallmantshift_no_sticky @ sticky_bit;

// we might get an overflow at the exponent subtraction in case the result
// negative; this would require that our addition shoud receive a carry_in
// because we only use the exponent difference at small mantissa shifting,
// will be translated in a shift right of the mantissa
```

```
gsmallmantshiftaux = gsmallmantshift >> adju (esigndiff[E+1], MSHIFT_LEN);

// NOTE: 'optimized' (alternative sequal = (x1.Sign == x2.Sign)
sequal = ~(x1.Sign ^ x2.Sign);

// operation -> addition/subtraction
op = x1.Sign ^ x2.Sign;

// final small mantissa
gsmallmantfin = op ? ~gsmallmantshiftaux : gsmallmantshiftaux;
```

4.1.2.2 Stage 2 - Mantissa addition and LZA

```
im_cin_carry_add28 (glargemant, gsmallmantfin, addmant);

/* possible overflow in case we do an addition */
addmantoflow = addmant[M+4];

/* sign of the result in case of a subtraction */
resultSign = addmant[M+4];

// do the lza implementation
lza (glargemant[M+3:0], gsmallmantshiftaux[M+3:0], lzacshift, lzafshift, i
```

4.1.2.3 Stage 3 - Normalization and exponent correction

```
addmantinvert = resultSign ? ~(addmant <- (M+4)) : (addmant <- (M+4));

// compute the total lza shift amount
rc_add (lzacshift, adju (lzafshift, SHIFT_LEN), lzatshift_c, ZERO);

// we might get an overflow; if that is the case the total shift value wil
lzatshift = lzatshift_c[SHIFT_LEN] ? (unsigned (SHIFT_LEN)) -1 : lzatshift

// if we have an addition we would have to correct the exponent and shift
// with addmantoflow; if we have a subtraction we would have to correct th
// lzatshift
expcorr = op ? ~(adju (lzatshift, E)) : adju (addmantoflow, E);

// Addition Exponent
rc_add (ZERO @ largeexp, ONE @ expcorr, addexp0, ZERO);
rc_add (ZERO @ largeexp, ONE @ expcorr, addexp1, ONE);

addexp = addexp0[E+1] ? addexp1 <- E : addexp0 <- E;

// Addition Mantissa - also considering the fine shift values
addmantresultfin = op ? ((addmantinvert << lzatshift) <- (M+3)) \\ 3: ((e1

grs = op ? (addmantinvert << lzatshift) <- 3 : ((e1_zero & e2_zero & ~adde
```

```
/*
 * here be the part where we be computing the denormalized number signals
 *
 * a number is denormalized if the shift value for the mantissa is more th
 * of the exponent; in that situation the exponent will have the least pos
 * (specific for a denormalized number) and the result mantissa will be sh
 * the difference between the shiftvalue and the exponent
 *
 * if addexp0[E] is 1 then the result is negative, and that means we would
 * number
 */

denorm = addexp0[E];

/*
 * we would then have to shift the mantissa with the value of the exponent
 */

denorm_mant = /*(e1_zero & e2_zero) ? ((addmantinvert << adju (largeexp, S
denorm_grs = ((addmantinvert << adju (largeexp, SHIFT_LEN)) <- 4) \\ 1;
```

4.1.2.4 Stage 4 - Output alignment

```
// check for special values - both values special
if (x1.Exponent == TwoToE_1 && x2.Exponent == TwoToE_1 && ~sequal) {
        par {
                Sign = 0;
                Exp = x1.Exponent;
                Mant = 4;
        }
}

// x1 special
else if (x1.Exponent == TwoToE_1) {
        par {
                Sign = x1.Sign;
                Exp = x1.Exponent;
                Mant = x1.Mantissa;
        }
}

// x2 special
else if (x2.Exponent == TwoToE_1) {
        par {
                Sign = x2.Sign;
                Exp = x2.Exponent;
                Mant = x2.Mantissa;
        }
}
```

```
// Signed equal
// Having the same sign we will be using the addition results for the expo
// (except of course for the exceptional cases)
else if (sequal == 1) {
        par {
                Sign = x1.Sign;

                if ((addmantoflow == 1) && (largeexp == TwoToE_2)) {
                        // return infinity
                        par {
                                Exp = TwoToE_1;
                                Mant = 0;
                        }
                }
                else {
                        // add same sign
                        par {
                                Exp = addexp;
                                Mant = addmantresultfin;
                        }
                }
        }
}

// different signs
// NOTE: this is where we would use our LZA/adder and store the appropiate
// the results are stored in the subtraction exponent/mantissa
else {
        par {
                // we have a zero result
                if (~is_zero) {
                        par {
                                Sign = ZERO;
                                Exp = 0;
                                Mant = 0;
                        }
                }
                else if (denorm) {
                        par {
                                Sign = resultSign ^ (~esigndiff[E] & x1.Si
                                Exp = 0;
                                Mant = denorm_mant;
                        }
                }
                else {
                        par {
                                Sign = resultSign ^ (~esigndiff[E] & x1.Si
                                Exp = addexp;
                                Mant = addmantresultfin;
```

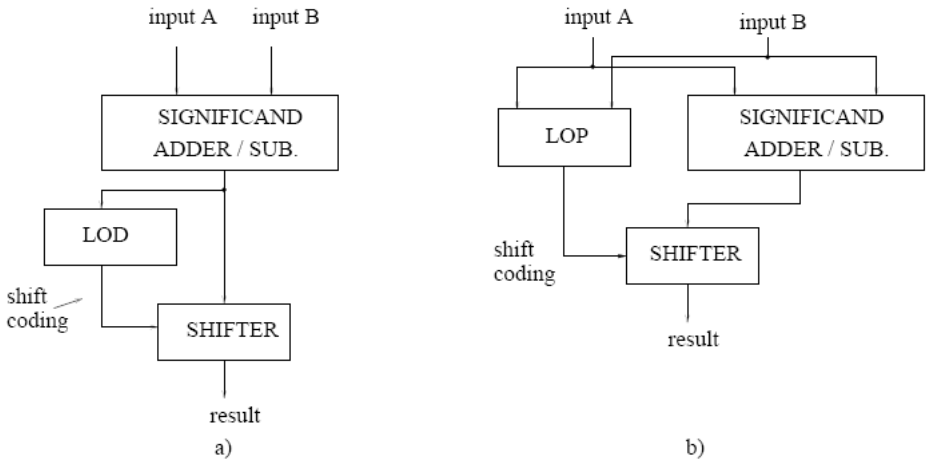
```
                                    }
                                }
                            }
                        }

                        result = (FLOAT(E, M)) {Sign, Exp, Mant};
```

4.2 Leading Zero Anticipation Unit structure

The LZA (Leading Zero Anticipation unit), also called LOP (Leading One Predictor unit), is the main improvement that we have brought to the overall floating point unit design. Typically, after doing the mantissa addition, there is a normalization step; this includes detecting the leading one (the first 1 bit of the result), shift the mantissa left by the proper amount and also add some value to the exponent.

We can however do the leading one detection while performing the addition (concurrently). This way we will need additional circuitry but we will also shorten the critical path. The two ways of doing the addition and normalization are shown below:



4.2.1 LZA design

4.2.2 LZA implementation

```
extern macro proc lza (xx1, xx2, cshift_ret, fshift_ret, is_zero);

                        /* pre-encoding leading one predictor signal */
                        F = ER & (G & ~SL | S & ~GL) | ~ER & (S & ~SL | G & ~GL);

                        /* pre-encoding leading one predictor correction signal - positive case */
                        PP = G & ~SL | ~ER & S & ~SL;
                        NP = ER & S;
                        ZP = ~(PP | NP);

                        /* pre-encoding leading one predictor correction signal - negative case */
```

```
                NN = S & ~GL | ~ER & G & ~GL;
                PN = ER & G;
                ZN = ~(PN | NN);

                MY_YL = _YNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 15);
                MY_YR = _YNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 14);
                MY_ZL = _ZNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 15);
                MY_ZR = _ZNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 14);
                MY_NL = _NNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 15);
                MY_NR = _NNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 14);
                MY_PL = _PNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 15);
                MY_PR = _PNSigExpr (ZN, PN, NN, LOG_WIDTH-4, 14);

                fshiftp = YPSigExpr (ZP, PP, NP, LOG_WIDTH);
                fshiftn = YNSigExpr (ZN, PN, NN, LOG_WIDTH);

                cshift = ~(adju (PExpr (F, LOG_WIDTH), LOG_WIDTH));

                par {
                        /* store the course shift value and fine shift value signals */
                        cshift_ret = adju (cshift, width (cshift_ret));
                        fshift_ret = fshiftp | fshiftn;

                        /* return the 'result is zero' signal */
                        is_zero = VExpr (F, LOG_WIDTH);
                }
        }

/* compute correction PPSig signal in a tree like manner (auxiliary macro is used) */
macro expr _PPSigExpr (ZP, PP, NP, len, index) =
        (select (len == 1, ZP[2 * index + 1] & PP[2 * index] | PP[2 * index + 1] & ZP[2 *
                                _ZPSigExpr (ZP, PP, NP, len-1, 2 * index + 1) & _PPSigExpr
                                _PPSigExpr (ZP, PP, NP, len-1, 2 * index + 1) & _ZPSigExpr

macro expr PPSigExpr (ZP, PP, NP, len) =
        _PPSigExpr (ZP, PP, NP, len, 0);
```

4.3 Adder structures

We have been using several basic adder structures throughout the implementation. Some of them have the advantage of a smaller critical path, the others have the advantage of a smaller circuit complexity.

4.3.1 Ripple carry adder

4.3.2 Carry select adder

4.3.3 Improved carry select adder

4.3.4 "Dual carry in" carry select adder

Chapter 5

Bitstream and ZBT interaction (Celoxica API)

We have been using the DMA API to handle communication between the host and the ZBT memory, and between the FPGA chip (through the bitstreamed Handel-C program) and the memory. Thus there are two sets of APIs: one for C files, also called ADMXRC2-SDK and one for Handel-C files, through the PSL API (from Celoxica PDK - Platform Development Kit). These APIs are used for interaction with the ZBT RAM banks on the ADMXRC2 (RC2000) board.

5.1 PDK API (Handel-C)

5.2 ADMXRC2-SDK (C)

Chapter 6

Generators and checkers

The *generators* are C programs that create input files for testing the adder unit implementation. The generated file are binary files. They contain pairs of input floating point numbers (in their binary representation) to be loaded on the RAM banks of the ADMXRC2 board. The *checker* is a C program that takes two files as its arguments: one contains the original input data and the other contains the result data. The results are thus compared against the host (PC) computation. If there is a mismatch it will be shown in human-readable form in an output file.

6.1 File Formats

There is a binary input file and a binary output file on a typical run of the whole floating point adder flow. There is also a text input file format. The text file is the intermediate for the "all cases" generation.

As stated before, the binary input file comes as a sequence of input pairs. Two floating point numbers in binary representation are stored in the file. The storage width for a single number could be 32 bit, 64 bit or 128 bit (it has to be enough for the number to be stored in its binary representation). So if we want to generate 1000 (one thousand) pairs, the input file will be 1000 * 2 * byte_width (bytes) in length.

Because of certain restrictions imposed by the DMA API, we had to write the result of the floating point addition **over** the second number. That means that the binary output file resembles the binary input file: we have successions of floating point number pairs in binary representation except that the second number is substituted for the result.

The "all cases" generator has two stages that can be piped together: the first stage outputs a text file and the second stage uses this text file to output a binary file. This is a small part of the text file:

```
; N + N -> N (positive Normalized + positive Normalized;
;               Result is also positive Normalized); big difference
=
; First Number
s=0
e=100
```

```
f=3
; Second Number
s=0
e=142
f=7
=
```

You can have comments on a line by themselves, but the line has to start with ';' (semicolon). The '=' (equal) sign can be used as pair separator. Each number will have a sign (s), an exponent (e) and a mantissa (f - fraction). These numbers can be decimal or hexadecimal numbers (for hexadecimal a 0x must precede the actual number).

6.2 "Random" Generator

The "random" generator is a C program. Its path is **c/raw/random_input_gen** (this is the path to the C project). The actual implementation resides in the **random_input_gen.c** file. The program generates a given number of floating point number pairs and stores them in a binary file. There are command line options to generate all zeros, all ones, numbers in increasing order or "random" numbers. These "random" numbers are not totally-random because the exponents of the two numbers of the same pair are rather close (most of the errors of a floating point adder unit implementation occur when the two number are close to each other).

The program can generate single-precision or double-precision pairs (32 bit or 64 bit floating point numbers). In case of the random option, it randomly generates the sign and the mantissa, and "pseudo-randomly" generates the exponent (in such a way that they are close to each other). The pairs are written in binary form to the output file.

6.3 "All Cases" Generator

As previously mentioned, there are two C programs contained in the *"all cases" generator*. One outputs a text file, and the other takes this text file as its input and outputs a binary file.

The path for the first program is **c/raw/all_cases_input_gen**. The implementing file is **all_cases_input_gen.c**. The format of the output file is presented in Section **??**. As we are trying to get as many cases as possible, there is an **enum float_operand** that defines various floating point number types (ranging from zero through denormalized and normalized to infinity, both positive and negative). All these types are combined in all possible pairs and an output file (with the specified format) is generated.

The path for the second program is **c/raw/text2bin_input_gen**. **text2bin_input_gen.c** contains the actual implementation. The program acts like a filter: it takes a text file as its input and outputs an equivalent binary file. The text input file needn't be generated by the first program. It could be created from scratch as long as it follows the specific syntax. This flexibility allows testing the floating point adder with any desired pairs of numbers. The program outputs a binary

file that will subsequently be uploaded in the ZBT RAM (it will have the format mentioned in Section ??).

6.4 Checker

The correctness checker program is the last part of the test flow. Its path is **c/raw/fp-add-check**. It will use the initial binary input file and the final binary result file to see whether the computation is correct. The results in the binary output file are compared against the results from the local CPU floating point unit. If there is a mismatch it will be outputted to a text file. A mismatch will contain the input pair, the FPGA computed result and the local CPU computed result. All these data are in both hexadecimal form (%x - so as to see the bits where the mismatch occurs) and in floating point form (%f - to see the actual number).

6.5 Handel-C Generator Project

The Handel-C generator project is a C program that can generate an entire Handel-C implementation for using the floating adder unit. Its path is **c/raw/hc-gen**. The behavior of the program is given by the various directives in the configuration file, **hc-gen.conf**. The directives in the configuration file specify which adder should be used, what exponent/mantissa width is going to be used, etc. The set of Handel-C files and headers have to be copied and assembled into a project using Celoxica DK. The Handel-C project will be compiled into a bitstream, and that can be used within the test framework.

There are two top-module Handel-C files: **fp_add_admxrc2_top.hcc** and **fp_add_sim_top.hcc**. The first one is to be used in the test flow; that means it will be used in the Handel-C project that outputs the bitstream file. The second one is to be used in a debug-mode project where no ADMXRC2 board interaction is required.

There are three C files in the Handel-C generator project. **hc-main.c** contains the main function; it is the top module and is responsible for calling the necessary functions from the other modules. **conf.c** is responsible for reading the contents of the **hc-gen.conf** file and storing the required data in a given set of variables that will be used when generating the Handel-C files. **adder-gen.c** contains the actual Handel-C files generation: it will generate the top files and then the actual floating point adder unit implementation. Besides the top module files (for the bitstream and debug-mode cases), these additional files are generated and must be integrated in the Handel-C project:

- adder.hcc, adder.hch - various generic (bit-width independent) adder implementations
- fp_adder.hcc, fp_adder.hch - the main floating point adder macro
- conf_dep.hcc, conf_dep.hch - configuration dependent macros
- non_generic.hcc, non_generic.hch - non-generic (bit-width dependent) adder implementations

- fp_ops.hcc, fp_ops.hch - various macros
- lza.hcc, lza.hch - Leading Zero Anticipation unit implementation

Chapter 7

Source Tree. How to Compile and Run

7.1 Source Tree

We will consider the source tree as relative path from the root directory. There are two parts in the source tree: the first is concerned with C programs (with the path **c**/), while the other is concerned with Handel-C programs (with the path **handel-c**/). There is an additional path, starting from the root directory, that is used by the test script to store input and output files; this is the path **data**/. The test script also resides in the root directory and is called **fpga-adder-test.sh**.

7.1.1 c/ path

The **c**/ path contains C programs used throughout the project. There are two sub-directories: **raw**/ contains "pure" C programs (that don't use any special library or API) an **admxrc2**/ which contains C programs using the ADMXRC2-SDK for communication with the bitstream loaded on the FPGA.

Each of the two sub-directories contains its own subset of projects. For the **raw**/ sub-directory, these projects are:

- **all-cases-input-gen** - "cover as many possible cases" input generator; outputs text file
  - **all-cases-input-gen.c**: contains the actual implementation for generating a text file with as many possible pair of floating point numbers as possible
  - **fp-types.h**: header file with floating point type structures
- **fp-add-check** - correctness checker
  - **fp-add-check.c**: implements the checker (as stated in Section **??**)
- **hc-gen** - Handel-C programs generator

  - **hc-main.c**: top module (contains the main function)
  - **conf.c**: configuration file analysis
  - **adder-gen.c**: generates the actual Handel-C files
  - **common.h**: common data
  - **conf.h**: conf.c associated header file
  - **default.h**: default common data values (expressed as macros)
  - **hc-gen.h**: adder-gen.c associated header file

- **random-input-gen** - random input generator
  - **random-input-gen.c**: implements the generator

- **text2bin-input-gen** - binary input generator; input is a text file from **all_cases_input_gen**; output is a binary file for testing
  - **text2bin-input-gen.c**: implements the converter
  - **fp-types.h**: header file with floating point type structures

For the **admxrc2**/ sub-directory these projects are:

- **old_api_bit_comm** - bitstream communicator program; synchronizes with the bitstream while on the FPGA board
  - **old_api_bit_comm.c**: communicator program implementation

- **old_api_zbt_ram_loader** - load data into ZBT RAM banks with the help of the FPGA chip
  - **old_api_zbt_ram_loader.c**: ram loader implementation

7.1.2 handel-c/ path

The **handel-c**/ path contains Handel-C programs. There are two sub-directories: **from-scratch**/ contains Handel-C programs that were written from the beginning (they were not generated, they were created "from scratch"); **generated**/ contains Handel-C programs that resulted from the Handel-C generator program (consisting of a C program and a configuration file).

The projects residing in th **from-scratch**/ sub-directory are:

- **clx_fp_adder_admxrc2** - test case for Celoxica's floating point adder implementation

- **clx_fp_adder_debug** - simulation-mode project for Celoxica's floating point adder implementation

- **cs_add_debug** - simulation-mode project for carry select adder; contains **cs_main.hcc**, **cs.hcc**, **cs.hch**

- **my_fp_adder32_admxrc2** - test case for my floating point adder implementation for 32 bit floating point numbers (single precision)

- **my_fp_adder32_debug** - simulation-mode project for 32 bit floating point adder
- **my_fp_adder64_admxrc** - test case for my floating point adder implementation for 64 bit floating point numbers (double precision)
- **my_fp_adder64_debug** - simulation-mode project for 64 bit floating point adder
- **my_fp_adder128_admxrc2** - test case for my floating point adder implementation for 128 bit floating point numbers (quadruple precision)
- **my_fp_adder128_debug** - simulation-mode project for 128 bit floating point adder
- **ram_load** - Handel-C project that intermediates the transfer of data from the host to the ZBT RAM banks of the ADMXRC2 board; contains **ram_load.hcc**

All the floating point adder projects contain these files:

- **c2s_adder.hcc, c2s_adder.hch** - carry select adder implementation
- **fp_adder.hcc, fp_adder.hch** - floating point adder implementation
- **decode1.hcc, decode1.hch** - macros for decoding a number
- **fp_ops.hcc** - various operations required by the floating point adder (comparisons, etc.)
- **lza.hcc, lza.hch** - LZA (Leading Zero Anticipation unit) implementation
- **fp_test.hcc** - top file for simulation-mode
- **fp_top.hcc, runtest.hcc** - top files for bitstream generation mode

Besides the top files, the projects dealing with Celoxica's implementation contain **float.hch** and **float.hcc**, used for the actual floating point adder unit implementation.

There is just two projects residing in the **generated/** sub-directory, namely **generated32_admxrc2** and **generated32_debug**.

7.1.3 data/ path

The **data/** path is initially empty (it contains no other files) but will subsequently contain **input.dat**, **output.dat** and **float_log.txt**. The first two are binary files: the file used for loading data *in* the ZBT RAM banks and the other used for retrieving data *from* the ZBT RAM banks. The third is a text file and is the output of the correctness checker; it shows the inconsistencies encountered why comparing the FPGA synthesized floating point adder results and the host (PC) results.

7.2 Compile and Configuration under Celoxica DK

As shown in the previous sections, there are two modes for creating projects under Celoxica DK. The first one is creating a debug-mode project (simulation only). The second one is creating a bitstream generation project; this would require using specialized ADMXRC2 settings and libraries.

7.2.1 Debug mode

Creating a project in debug-mode doesn't require any special setting. It is likely that **stdlib.hcl** (standard Handel-C library) should be linked to the project. The P**roject -> Settings** menu item has to be chosen for this. Under the **Preprocessor** tab, one should specify the path to the **PDK include directory** (something like C:\Program Files\Celoxica\PDK\Hardware\Include). Under the **Linker** tab, one should specify the path to the **PDK library directory** and also specify **stdlib.hcl** as the library to be included in the linking process.

These settings done, the program can be compiled and, eventually, simulated. One should select **Debug** option from the top combo-box in the IDE and the project will be ready for compiling and simulation.

Debug mode is required for small tests that try to highlight a certain error, but also to avoid the long time required for obtaining a bitstream; if there is no need for a complete FPGA-based testing, debug mode should be considered.

7.2.2 Bitstream generation mode

Creating a project in "bitstream generation mode" (that is with the goal of obtaining a bitstream to be loaded on the ADMXRC2 board) requires a great deal of configurations.

The usual steps undertaken to create such a configuration are detailed below.

7.2.2.1 Create an ADMXRC2 (RC2000) configuration

Every projects works with an active build configuration. The default is usually **Debug**. Trying to create a bitstream for the ADMXRC2 board requires creating a specialized configuration. This is done by using **Build -> Configurations**. The **Add** button is to be used to create a new configuration. It is advisable that its name be ADRMXRC2 or RC2000; usually, the settings will be imported from the **EDIF** configuration.

7.2.2.2 Update the ADMXRC2 configuration

Now that the configuration is created all we must edit it. The **Project -> Settings** menu item brings us there. The desired project has to be selected and the desired configuration from the top combo-box in the new window (the configuration should be named ADMXRC2, if previous advice was followed).

We will detail the steps for each particular tab in the **Project Settings** window.

General tab

- the **Output directories** text areas should specify **ADMXRC2** (instead of EDIF)

Preprocessor tab

- the **Additional include directories** must specify the path to the **PDK include directory** (something like C:\Program Files\Celoxica\PDK\Hardware\Include)

Synthesis tab

- check **Enable retiming**

Chip tab The exact chip for the ADMXRC2 board has to be specified.

- Family: **Xilinx Virtex-II (XilinxVirtexII)**
- Device: **xc2v6000**
- Package: **ff1152**
- Speed Grade: **4**
- Part is automatically detected from the above settings (**xc2v6000ff1152-4**)

Linker tab

- **Object/library modules** must contain **stdlib.hcl,admxrc2.hcl**
- **Additional library path** must point to the **PDK library directory** (something like C:\Program Files\Celoxica\PDK\Hardware\Lib)

Build commands tab

- after selecting the **Commands** view, the following commands have to be added
  - cd ADMXRC2 (assuming ADMXRC2 was specified in the **General** tab)
  - edifmakeADMXRCx project_name (where project_name is the name of the project)
- after selecting the **Outputs** view, the following output has to be inserted
  - ADMXRC2\project_name.bit (assuming ADMXRC2 was specified in the **General** tab; project_name is the name of the project)

Building the project will produce the required bitstream that can be loaded on the ADMXRC2 board. Xilinx ISE must be installed on the system; edifmakeADMXRCx is actually a batch script that will also run Xilinx' **ngdbuild** to do the place & route and produce the actual bitstream file.

7.3 Examples on Running Various C Projects

Here are some run-cases of the various C projects in the test frame work. There is at least one run-case for each project, accompanied by an explanation of its syntax and its behavior. The project name is preceded by its C sub-directory path (**raw** or **admxrc2**).

7.3.1 raw/all-cases-input-gen

Run-cases:

all-cases-input-gen.exe 32 cases.txt

all-cases-input-gen.exe 64

The first argument has to be 32, 64 or 128; it will specify the length of the floating point number pairs to be generated in the output text file. The second argument is optional and specifies the output file. If it is not present everything gets written to standard output; the program can thus be used as part of a piped command shell; it's output will be used to feed another program's input (typically the **text2bin-input-gen** project).

7.3.2 raw/fp-add-check

Run-cases:

fp-add-check.exe input.dat output.dat

The first argument is the initial binary file loaded in the ZBT RAM banks (prior to running the Handel-C floating point adder unit project). The second is the result binary file. The format of the two files is described in Section **??**.

7.3.3 raw/hc-gen

Run-cases:

hc-gen.exe -f special-config.conf

hc-gen.exe

The first argument should be the configuration file used for generating the Handel-C programs. If that is missing the default configuration file is assumed. This file is defined as a macro in **default.h** (DEFAULT_CONFIG_FILE). The second run-case would use the default configuration file.

7.3.4 raw/random-input-gen

Run-cases:

random-input-gen 32 2000 r

random-input-gen 64 1000 o rand_input.dat

random-input-gen 32 2000 i rand_input.dat

The first argument is the bit-width of the floating point numbers to be generated. It can be 32 or 64. The second arguments specifies the number of pairs to be generated. The third argument can be *r* or *o* or *z* or *i*, that is random numbers, only numbers that have the value 1 (in binary representation), only numbers that have the value 0, or numbers with continuous increasing value (in their binary representation) starting from 0. As stated in Section **??**, the random numbers are not truly "random"; they are randomly generated but under

the constraint that the exponents for the two numbers must be fairly close to each other. The last argument may be missing. If it is, the output file is the one defined by the internal GEN_FILE macro. Otherwise the output will be stored in the specified file.

7.3.5 raw/text2bin-input-gen

Run-cases:

text2bin-input-gen - cases_input.dat

text2bin-input-gen cases.txt cases_input.dat

The executable requires two arguments. The first is the text file used as input. The other is the output binary file. If the first argument is '-' then standard input will be used as input to the program. Thus, the program can be used as a second part of a piped command together with **all-cases-input-gen**.

A piped command could be one like this:

all-cases-input-gen 32 | text2bin-input-gen - cases_input.dat

7.3.6 admxrc2/old_api_bit_comm

Run-cases:

old_api_bit_comm -bit generated32_admxrc2.bit

old_api_bit_comm -cr 50000000 -bit generated32_admxrc2.bit

There has to be a *'-bit filename.bit'* set of arguments that specify the bit-stream file containing the actual floating point adder unit implementation. We can optionally specify the clock rate (in Hz) using the *'-cr clock_rate'* set of arguments. If the clock rate is not specified the default is assumed (it is defined in the source code by the DEFAULT_CLOCK_RATE macro).

7.3.7 admxrc2/old_api_zbt_ram_loader

Run-cases:

old_api_zbt_ram_loader -bit ram_loader.bit -m FROM_FILE -dir TO_ZBT -file rand_input.dat -offset 0 -cr 40000000

old_api_zbt_ram_loader -bit ram_loader.bit -m FROM_FILE -dir FROM_ZBT -file output.dat -offset 0 -size 2000

This is the most complex command in terms of command line arguments. Sets of arguments are used to specify:

- the bitstream file to intermediate the data transfer between the host and the ZBT RAM banks
- the method used for transferring data; it can be FROM_FILE (data is read from/written to a binary file); the alternative would be FROM_CMD and data is read from standard input and is written to standard output
- the direction of the transfer (TO_ZBT is from host to RAM banks; FROM_ZBT is from RAM banks to ZBT)
- the file to be used as input/output
- the offset in the RAM banks where data should be read from/written to

- in the case where we want to read data from RAM we have to specify the size of the block we wish to read
- we can optionally specify the clock rate; if not, the default is assumed (defined by the internal DEFAULT_CLOCK_RATE macro)

7.4 Running the Test Script

The **fpga-adder-test.sh** shell script contains the whole flow for testing the adder unit. Running it is as simple as running the command

sh fpga-adder-test.sh (I have used MinGW to run the script)

There are no command line arguments. The script can be customized through the use of internal variables. There are two sets of internal variables. Some that control the flow of the script, and other to specify the location of the C programs and bitstream files used throughout the flow.

The **flow control** variables are:

- **INPUT_FILE**: specifies the name of the input file; the random generator will generate an input file with this specific name
- **OUTPUT_FILE**: specifies the name of the output file; when the results are retrieved from the RAM banks, they will be stored in a file with this name
- **LOG_FILE**: the name of the human-readable error file outputted by the correction checker
- **MAIN_CLOCK_RATE**: the clock rate to be used by the floating point adder unit
- **STEPS**: the number of steps in the session that we repeat the whole test flow (for more thorough investigation it is recommended to repeat the test flow within a test session)
- **NBYTES**: number of bytes for input data
- **ZERO_BYTE_OFFSET**: because the DMA API ("old API") is somewhat broken we can not trust the first few hundred bytes of data from the RAM banks; as such we are using an initial zero byte filled offset (before the actual data)

The variables specifying the location of C programs and bitstream files are:

- **WORK_DIR**: the root directory for the project (the root path - MinGW style syntax)
- **WORK_DIR_WIN**: the root directory (Windows syntax)
- **DATA_DIR**: the **data** path
- **C_DIR**: the **c** path
- **HC_DIR**: the **hc** path

- **RAW_DIR**: the **raw** sub-directory (of the **c** path)
- **ADMXRC2_DIR**: the **admxrc2** sub-directory (of the **c** path)
- **SCRATCH_DIR**: the **scratch** sub-directory (of the **hc** path)
- **GENERATED_DIR**: the **generated** sub-directory (of the **hc** path)
- **RAND_GEN**: the random generator executable path (MinGW syntax)
- **CHECKER**: the correction checker executable path (MinGW syntax)
- **RAM_LOADER**: the ram loader executable path (MinGW syntax)
- **BIT_COMM**: the bitstream communicator executable path (MinGW syntax)
- **RAM_LOAD_BITFILE**: the ram loader bistream path (Windows syntax)
- **MAIN_BITFILE**: the floating point adder unit test bitstream path (Windows syntax)

The script will clean the log file and then run for **$STEPS** times the whole test flow as described in Section **??**.

Chapter 8

Conclusions

8.1 Results

We have managed to do a complete testing and simulation for different types of floating point adder circuits (32 bit and 64 bit). Our testing framework enables testing different implementations with relative ease; that means that we can very easily find out errors in the implementation and also discover the maximum frequency the circuit can operate on.

An information file (with the extension .twr) can be obtained by issuing the command:

trce -a project_name.ngd (*project_name* is the project's name; the .ngd file is the result of the place & route accomplished by Xilinx' **ngdbuild** program)

This file shows the maximum estimated frequency. While this is most likely, the maximum frequency that the circuit is assumed to behave without flaws, higher frequencies can usually be reached. By using the script, one can easily modify the **MAIN_CLOCK_RATE** variable and find the frequency where the circuit stops working (errors will be appear in the log file - the output file of the correctness checker).

Our circuit usually outperforms Celoxica's own implementation in terms of speed (not in terms of circuit complexity though). Our implementation is accurate as it correctly treats denormalized numbers and considers RGS (Round, Guard, Sticky) set of bits. Celoxica's floating point adder unit had some errors when the operands would be denormalized numbers.

We have been using our Handel-C program to test various basic adder units to be used. As it would turn out, a ripple-carry adder is indeed suitable for small widths (no significant amount of speed can be achieved by using another adder, not to mention that it would add complexity to the circuit). The carry select adder is a good trade-off between speed and circuit complexity. We have found that Celoxica's add implementation (when using the '+' operator) is not just a ripple carry adder, but a dynamic adder that changes its behavior as the width of the operands increases. From a certain operand width on, Celoxica's internal adder implementation outperforms the carry select adder both in terms of speed and circuit complexity. However, we have been using a modified version of the carry select adder, a thing that can't be done with Celoxica's implementation. Not only that, but Celoxica's implementation doesn't deal with an initial carry

in that we would require for most of our work.

8.2 What Can be Improved?

There are many points in the project that can benefit from some improvements. The most critical aspect is concerned with making the test framework as general as possible. That would mean the whole test flow could be able to deal with arbitrary precision (arbitrary exponent width and arbitrary mantissa width) floating point numbers: the input generators, the Handel-C generator, the checker, etc. This would require that some of the host floating point testing be able to handle arbitrary precision. This could be emulated using C code or maybe a specific library (such as GNU Multiprecision Library - useful for large bit width numbers).

8.3 Further Work

We have been trying to make everything as efficient as possible in terms of circuit design. We have tried to use as little signals as possible, to use our own implementation, to squeeze the last drop out of the design. However, Handel-C is still a "high-level" HDL (Hardware Description Language) and it is up to the compiler to manage the optimizations which could render our efforts useless. A Verilog (VHDL) tuning of certain parts of the circuit is an idea to be considered.

Moreover, working with FPGAs does have its disadvantages in not knowing how the mapping is done (that is we cannot easily predict how parts of the circuit are mapped to the LUTs - Lookup Tables - of the FPGA). Some analysis could be done on the netlist resulted from compiling the project, but it's tedious and unrewarding work. Once again, working with a "low-level" HDL (such as Verilog or VHDL) could potentially improve the overall behavior by managing to control some aspects of the LUT mapping.

Besides a floating point adder unit, a multiplier is also required. Most of current floating point workings (such as DSP) generally do a lot of multiplications and additions. While we have designed a floating point multiplication unit (**fp_mul.hcc** and **fp_mul.hch** in **handel-c/from-scratch/my_fp_adder32_admxrc2**), but no testing framework has been established for it nor have we improved it in terms of speed and circuit complexity. As most of the Xilinx chips (including the Virtex-II from the ADMXRC2 board) posses internal 18x18 bit multiplier units, they should be considered for the multiply implementation; under the **Project -> Settings** and **Synthesis** tab, one should check the **Enable Mappings to ALU**.

Upon completing the multiply implementation, a MAF (Multiply and Add Fused unit) should be created as it is a fairly frequent operation.