To the Graduate Council:

I am submitting herewith a thesis written by Bo Su entitled " Study of Second-Order Floating-Point Low Round-Off Noise IIR Filter Implementation in FPGA." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Bruce W. Bomar, Major Professor

We have read this thesis and recommend its acceptance.

Roy D. Joseph

Bruce A. Whitehead

Accepted for the Council:

Associate Vice Chancellor and
Dean of the Graduate School

STUDY OF SECOND-ORDER FLOATING-POINT LOW ROUNDOFF NOISE IIR FILTER IMPLEMENTATION IN FPGA

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Bo Su

December 1998

ACKNOWLEDGMENTS

There are many people to whom I am grateful for making my time at The University of Tennessee Space Institute rewarding. I am particularly thankful to my Thesis Committee, Drs. Bruce W. Bomar, Roy D. Joseph and Bruce A. Whitehead for their advice and assistance in the preparation of this thesis. I also express my appreciation to the UTSI administration and the CIPG Research Group for granting me the Graduate Research Assistantship to support my study. I would like to thank Ms. Callie Taylor for her help with my insurance and to thank Ms. Clara Ferguson for her secretarial assistance.

Also I want to thank the Altera Company for donating the software and programming hardware through their University Support Program.

Lastly, the greatest debt owed to my wife, Jenny. Without her help and encouragement I would never have reached this moment.

ABSTRACT

In this study, the implementation of second-order floating-point IIR filters by FPGAs was investigated. A small word length floating-point representation was chosen to keep the roundoff noise below the input quantization error while also keeping the logic resources required as low as possible. The state-space and direct form structures were compared; the state-space structure was chosen because of its low roundoff noise at the filter output.

A bit-parallel floating-point multiplier used especially for filter design was developed on the base of a megafunction of Altera Company. A bit-serial floating-point multiplier was designed to reduce the logic cells. The logic resources used and speed of computation were investigated with different small floating-point representations.

The second-order state-space floating-point IIR filter was designed using the floating-point multipliers and a floating-point adder megafunction. The results show that floating-point filter design can be implemented in current FPGA with reasonable speed and logic resources. Depending on the number bits of mantissa and exponent, the second-order IIR sample rate can be over 1 MHz if the bit-parallel floating-point multiplier is used. Logic resources can be saved by using bit-serial multiplier, especially for a large number of mantissa bits, but the speed is comparatively low.

Representative implementations were described in VHDL or AHDL and complied and simulated using Altera MAX+PLUS II software.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Chapter I

INTRODUCTION

Normally there are two ways to implement digital filters. One uses a general-purpose digital signal processor (DSP), while the other uses a custom application-specific integrated circuit (ASIC). The DSP approach is simple and easy to change but may not provide sufficient speed or be cost effective. While the ASIC approach can provide high filter speeds, the ASIC is costly to develop and difficult to modify once developed.

With the introduction of the Field Programmable Gate Array (FPGA; a configurable logic chip) in the early 90's, the hardware engineer was empowered to implement chip-level designs in silicon without having to fabricate a chip. Today's FPGAs offer an attractive solution for the computationally intensive functions found in DSP applications like digital filter design. By combining the flexibility of a general-purpose, programmable DSP with the speed and density of a ASIC implementation, FPGAs can provide increased DSP system performance at a reduced system cost.

In current implementations of digital filters in FPGAs, fixed-point arithmetic is normally used instead of floating-point arithmetic. Until recently, the use of floating-point arithmetic on FPGAs has been virtually impossible because floating-point operators require excessive area (time) due to the inherent complexity of floating-point arithmetic. But in some DSP applications such as IIR filter implementation, floating-point arithmetic has significant advantages compared with fixed-point arithmetic. In IIR filter design, floating-point arithmetic simultaneously eliminates most concerns regarding scaling, limit cycles, and overflow oscillations [1]. With the increasing density of FPGAs and the

introduction of a high level hardware design language such as VHDL, rapid prototyping and implementation of floating point arithmetic has become possible in FPGAs. Elaborate simulation and synthesis tools at a higher design level aid the designer in accomplishing this. These make the floating-point implementation of IIR digital filters feasible in the current FPGAs.

The purpose of this research was to study floating-point IIR filter implementation in current FPGAs. To simplify the work, the study is restricted to second-order filter design. High-order IIR filters can be realized by either a parallel or cascade connection of first and second-order subfilters; the first-order subfilter is a special case of the second-order subfilter. Also, a state-space structure was suggested for use [1] because of its low round-off noise.

The first area of concern is the number of bits of mantissa and exponent requried in the design's floating-point representation. Programs are developed to compute the roundoff noise with different representations. The mantissa bits are chosen to keep the roundoff noise below the input quantization error. Then, the floating-point multiplier and adder are studied in detail and the relative performance in terms of logic resources and speed of computation are presented. Using the floating-point operators and the state-space structure, the second-order floating-point filter is implemented and analyzed.

The organization of the thesis is as follows: Chapter II presents the necessary background information on the Altera FPGA and MAX+PLUS II software , the second-order IIR filter structure and the floating-point representation. In Chapter III programs are developed to simulate FPGA floating-point computation in order to select the required number of mantissa and exponent bits. Chapter IV considers the detailed

implementation of a floating-point multiplier and a floating-point adder. Chapter V provides an actual implementation of a second-order floating-point IIR filter. Chapter VI summarizes the results and gives recommendations.

Chapter II

BACKGROUND INFORMATION

2.1 Altera FLEX 10K FPGA and MAX+PLUS II Software

In the thesis the Altera FLEX 10K series FPGAs and MAXPLUS II software [10] were used. The Altera University Program provides the hardware and software for digital logic desgn in an academic environment. MAXPLUS II software is used to design, simulate, analyze and prototype the implementation of the design in FPGAs.

2.1.1 Altera FLEX 10K FPGA

Altera offers three families of FPGAs: FLEX 10K, FLEX 8K, FLEX 6K, with densities ranging up to 250,000 usable gates. The SRAM-based FLEX 10K family has an architecture similar to the 6K and 8K families but is unique in that it includes an embedded memory array, as shown in Figure 2.1.

The flexible, programmable embedded array consists of embedded array blocks (EABs) that contain 2K bits of RAM (4K bits for FLEX 10KE). The EAB can be used to create ROMs, FIFOs, and asynchronous, synchronous, and dual-port RAM. Flexible EAB width and depth configuration are possible without incurring speed penalties.

The logic array contains logic array blocks (LABs) that communicate through a fully populated local interconnect structure. Each LAB contains eight logic elements (LEs). The LE is the smallest unit of logic in the device architecture, which has a carry chain and cascade chain along with a four-input look-up table (LUT) and programmable

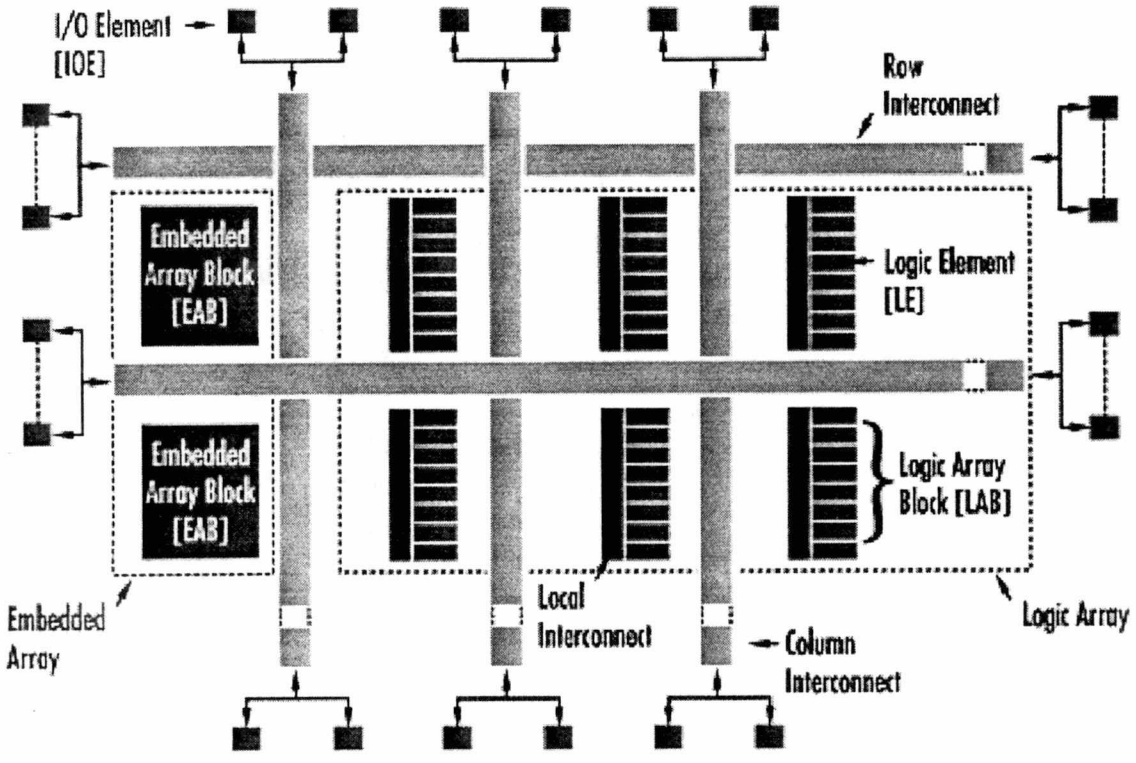


Figure 2.1 FLEX10K block diagram

flip-flop with local clock enable control as shown in Figure 2.2. The LE offers a dual-output structure that allows for both sequential and combinatorial outputs.

Both the embedded array and the logic array communicate through programmable row and column interconnects. This continuous interconnect structure offers high performance and predictable timing . Each interconnect row and column feeds multiple I/O elements (IOEs), each of which contains an I/O register with flexible control signals, programmable slew-rate control and an output enable pin. Table 2.1 summarizes the FLEX 10K family features.

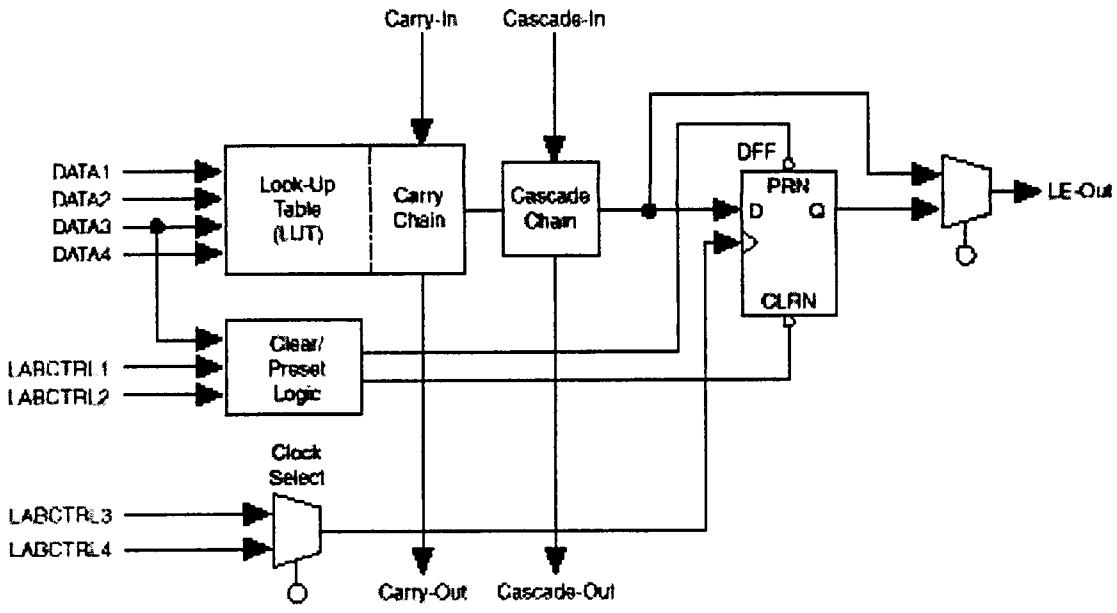


Figure 2.2 FLEX 10K logic element block diagram

Table 2.1 FLEX 10K family series

FLEX10K Series	Typical Gates	Logic Elements	LABs	EABs	Total RAM bits
EPF10K10	10,000	576	72	3	6144
EPF10K20	20,000	1152	144	6	12,288
EPF10K30	30,000	1728	216	6	12,288
EPF10K40	40,000	2304	288	8	16,384
EPF10K50	50,000	2880	360	10	20,480
EPF10K70	70,000	3744	468	9	18,432
EPF10K100	100,000	4992	624	12	24,576
EPF10K130	130,000	6656	832	16	32,768
EPF10K200E	200,000	9984	1248	24	98,304
EPF10K250E	250,000	12160	1520	20	81,920

2.2 MAX+PLUS II Software

Programmable logic designs that require the density of the FLEX 10K family also need fast, efficient and easy-to-use design software. Altera's MAX+PLUS II [10] development software is a fully integrated programmable logic design environment. It provides the flexibility to enter a design using all the major design entry methodologies including:

- VHSIC Hardware Description Language(VHDL)
- Verilog HDL
- Schematic Capture
- Altera Hardware Description Language(AHDL)
- Design Entry Utilizing Megafuncions
- Design Entry Utilizing the Library of Parameterized Modules(LPM)
- Waveform Design Entry

High-density logic designs can use megafunctions to simplify the design process. Designers can implement megafunctions or use a variety of predefined megafunctions--including memory, arithmetic logic, microprocessors/microcontrollers, and DSP functions.

Once a design is entered, it is processed by the MAX+PLUS II compiler producing the various files used for verification or device programming. The compiler consists of a series of modules that check a design for errors, synthesize the logic, fit the design into Altera device, and generate the files for simulation, timing analysis, and

device programming. It also provides a visual presentation of the compilation process, showing which of the modules is currently active and allowing this process to be stopped.

MAX+PLUS II supports LPM, an industry-standard, technology-independent library that makes the design of complex megafunctions possible. Altera provides many predefined megafunctions and also supports megafunctions provided by Altera Megafuntions Partners Program (AMPP), an alliance between software and developers of synthesizable megafunctions. MAX+PLUS II and these megafunctions significantly reduce design tasks and dramatically shorten design cycles.

2.2 IIR filter Realization

Digital infinite impulse response (IIR) filters are widely used in such areas as digital audio, digital control and telecommunication. In practice, high-order transfer functions are realized as cascades or parallel combinations of second and/or first order subfilters.

2.2.1 Finite Wordlength Effects

Practical digital filters must be implemented with finite precision numbers and arithmetic. The finite wordlength causes the following nonideal IIR filter performance:

- Input quantization noise, which results from representing the samples of the input by only a small number of bits.
- Coefficient quantization errors caused by representing the IIR filter coefficients by a finite number of bits.
- Roundoff noise caused when the output and the result of internal arithmetic operations are rounded to the permissible wordlength.

- Limit cycles, which result from the nonlinearity of the quantization of the filter calculation.
- Overflow oscillation, which can exist when fixed-point arithmetic calculations overflow.

The analysis of finite wordlength effects can be found in many DSP books. In [1] both fixed-point and floating-point number representations are examined in detail and it is concluded that " *It is highly recommended that floating-point arithmetic be used for IIR filters. Floating-point arithmetic simultaneously eliminates most concerns regarding scaling, limit cycles, and overflow oscillations. Regardless of the arithmetic employed, a low round-off noise structure should be used.* "

2.2.2 Second-order IIR Filter Structure

The second-order IIR filter is characterized by the following transfer function:

$$H(z)=\frac{a_0+a_1z^{-1}+a_2z^{-2}}{1+b_1z^{-1}+b_2z^{-2}} \qquad (2.1)$$

The direct form realization is depicted in Figure 2.3. Other forms like the canonic form, transpose of canonic, and direct form can be found in [2]. These structures only need four multiplications but are much more sensitive to finite wordlength effects.

Among the methods for implementing the second-order filter or subfilters, the state-space structure has emerged as one of the most useful, due in large part to the

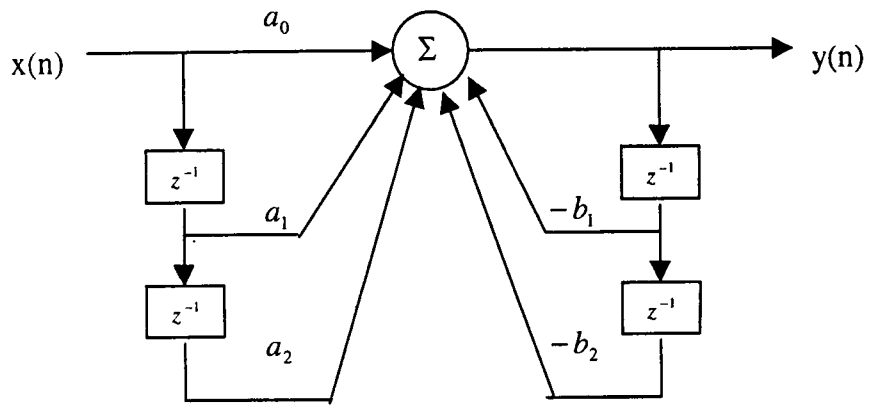


Figure 2.3 Direct form second-order section

flexibility it provides in terms of reducing the effect of finite precision arithmetic roundoff on the filter output. If the transfer function of (2.1) is rewritten in the form

$$H(z) = d + \frac{\alpha_1 z^{-1} + \alpha_2 z^{-2}}{1 + \beta_1 z^{-1} + \beta_2 z^{-2}} \tag{2.2}$$

where

$$d = a_0$$

$$\alpha_1 = a_1 - a_0 b_1 \qquad \alpha_2 = a_2 - a_0 b_1$$

$$\beta_1 = b_1 \qquad \beta_2 = b_2$$

it can be realized by a state-space structure of the form

$$x(k+1) = Ax(k) + bu(k) \tag{2.3}$$

$$y(k) = c^t x(k) + du(k) \tag{2.4}$$

Where A is a 2 by 2 matrix and b, c, and x are 2 by 1 matrix. There are an infinite number of such realizations. In [3] and [4] several coefficient choices are given and the minimum roundoff noise case will be chosen for this study. Because floating-point arithmetic does not require filter scaling [7], seven multiplies can be used instead of nine in a fixed-point realization. From [4] and [12] the minimum roundoff noise coefficient can be found by the following design equations:

$$b_1 = b_2 = 1.0 \tag{2.5}$$

$$a_{11} = a_{22} = -\frac{\beta_1}{2} \tag{2.6}$$

$$\mu = \sqrt{\left(\frac{\alpha_2}{\alpha_1}\right)^2 - \frac{\alpha_2}{\alpha_1}\beta_1 + \beta_2} \tag{2.7}$$

$$\gamma = \frac{\alpha_2}{\alpha_1} - \mu \tag{2.8}$$

$$\xi = \frac{\alpha_2}{\alpha_1} + \mu \tag{2.9}$$

$$\varepsilon = \left(\frac{\beta_1}{2}\right)^2 - \beta_2 \tag{2.10}$$

$$t_1 = \sqrt{\frac{\lambda}{2\beta_1\gamma - (\beta_2 + 1)(1 + \gamma^2)}} \tag{2.11}$$

$$t_2 = \sqrt{\frac{\lambda}{2\beta_1\xi - (\beta_2 + 1)(1 + \xi^2)}} \tag{2.12}$$

$$\lambda = (\beta_2 - 1)\left[(\beta_2 + 1)^2 - \beta_1{}^2\right] \tag{2.13}$$

$$a_{21} = \frac{t_2}{t_1}\sqrt{\frac{b_2{}^2 + \beta_2 - 1}{b_1{}^2 + \beta_2 - 1}}\varepsilon \tag{2.14}$$

$$a_{12} = \frac{\varepsilon}{a_{21}} \frac{t_2}{t_1} \tag{2.15}$$

$$c_1 = c_2 = \frac{\alpha_1}{2} \tag{2.16}$$

The design equations presented are suitable for calculator use and can be easily combined into a simple computer program. This state-space structure is shown in Figure 2.4. In the following chapters the floating-point arithmetic implementation of this state-space structure will be studied in detail.

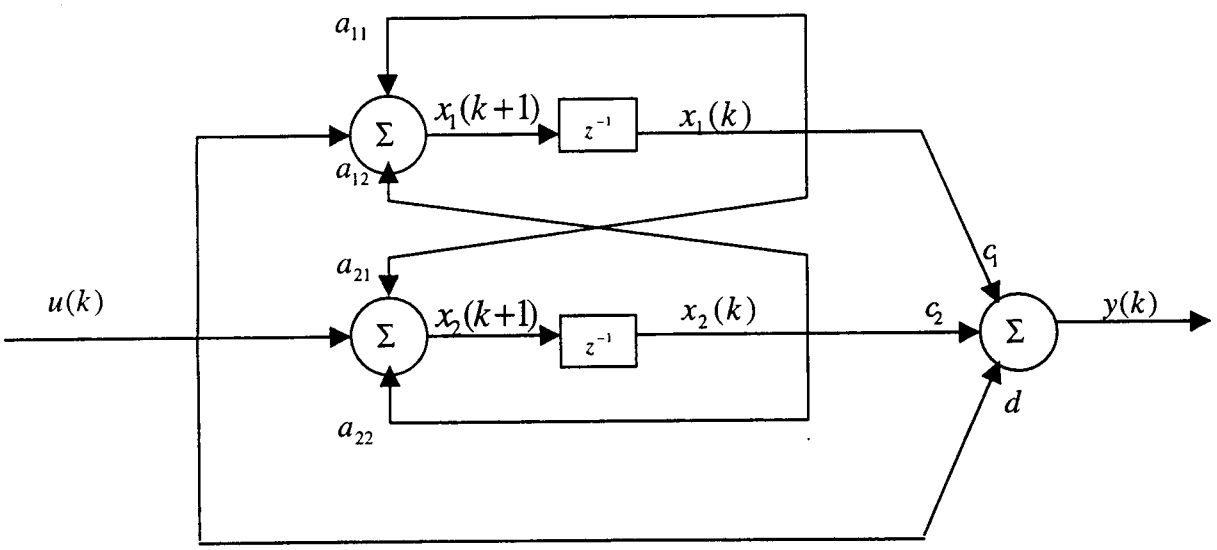


Figure 2.4 State-space second-order section

2.3 Representation of Floating-point Numbers

In floating-point representation the binary word is logically broken into three fields. The first field is the sign bit (s), and indicates whether or not the number is

negative. The second field is the exponent (e) and the third field is the mantissa (m). A standard floating-point format is the IEEE 754 standard[11] for 32-bit single-precision numbers. In that format a number is stored as

$$X = (-1)^s (0.5 + f) 2^{p-126} = (-1)^s (m) 2^e \qquad (2.17)$$

where s is the sign bit, f is a 23-b unsigned fraction in the range between 0 and 0.5, and p is an 8-b unsigned integer in the range between 0 and 255, m is the mantissa, and e is the exponent.

Altera has some floating-point megafunctions [13] that use a slightly different representation than IEEE 754. In these megafunctions

$$X = (-1)^{1-s} (m) 2^{(p-2^{n-1})} \qquad (2.18)$$

The mantissa is a positive number less than 1. A 0 is implied to the left of the binary point. After normalization, the most significant bit (MSB) is always 1. The exponent is represented in excess $2^{(n-1)}$ notation, where n is the number of bits in the exponent. For example, the binary representation of the number 1.5 is shown below. The example assumes 8 bits for the mantissa and 7 bits for the exponent. s represents the sign bit.

s=1, m=.110000000, p=1000001

Similarly, the binary representation of the number –0.3125 is :

s=0, m=.101000000, p=0111111

Because Altera megafunctions are used in the research, this format is used to represent of floating-point numbers.

Chapter III

COMPUTER SIMULATION

In most computers floating-point numbers are represented in either IEEE single precision format using 24 bits of mantissa and 8 bits of exponent or double precision which uses 48 bits of mantissa and 16 bits of exponent. Unfortunately, such formats would require excessive logic resources to implement in an FPGA. Also, in real-time processing, typically 8 bits, 12 bits or 16 bits are used to represent the input and output signals, so it is not necessary to use double or even single precision in the internal computation. Therefore, floating-point formats with fewer bits of mantissa and exponent than single or double precision could be acceptable. However, it remains to determine just how many exponent and mantissa bits are really needed.

3.1 Roundoff Noise Computation

To reduce the FPGA resources, as small a floating-point representation as possible should be used. Therefore, it is necessary to analyze the roundoff noise to decide how many mantissa and exponent bits are needed in the small floating-point representation.

The roundoff noise can be computed using computer programs to simulate the FPGA implementation. Programs were developed to compute the roundoff noise of the state-space structure of the second-order IIR filter.

We assume that there is negligible round-off noise if the internal computations are done using double precision representation. This is then compared to the result using the small floating-point format to represent the floating-point number. A Gaussan random

signal with zero mean and unit variance is chosen as the input. The input is quantized to the small floating-point format in both the double precision and the small precision computation to get rid of the quantization error and the coefficients are quantized to small floating-point format in both cases to discard the coefficient error. After getting the filter output of the double precision and small precision, the data are compared and the roundoff noise variance is computed.

In order to compare the roundoff noise of the state-space structure to the direct form structure, the direct form roundoff noise variance is also computed. All the programs are listed in Appendix A1. The functions of the listed programs are given in Table 3.1.

Table 3.1 Simulation programs description

Program Name	Function description
state_space.cc	Using state-space structure, choosing double precision to represent internal computation. The output is assumed no roundoff noise.
small_state_space.cc	Using state-space structure, choosing small precision to represent internal computation. The output has roundoff noise.
direct.cc	Using direct form structure, choosing double precision to represent internal computation. The output is assumed no roundoff noise.
small_direct.cc	Using direct form structure, choosing double precision to represent internal computation. The output has roundoff noise .
variance.cc	Input the no-roundoff-noise filter result file and the round-off-noise filter result file of the same structure and compute the roundoff noise variance.

3.2 Result Analysis

Two randomly chosen second-order IIR filter transfer functions are used as examples to compute the roundoff noise. One is the Butterworth high-pass filter with a passband edge frequency at one percent of the foldover frequency used in [4]. The transfer function of this filter can be written

$$H(z) = 0.97803048 + \frac{-0.04344583z^{-1} + 0.04250161z^{-2}}{1 - 1.95557824z^{-1} + 0.95654368z^{-2}} \qquad (3.1)$$

and will be called function H1. Another is one section of the parallel realization of a sixth-order Chebyshev high-pass filter with 1-dB passband ripple and a passband edge at 2% of the foldover frequency used in [3]. For this section

$$H(z) = \frac{-0.23151z^{-1} + 0.199235z^{-2}}{1 - 1.763888z^{-1} + 0.792049z^{-2}} \qquad (3.2)$$

which will be called function H2.

The number of exponent bits determines the dynamic range of the small floating-point numbers. The dynamic range (absolute value) of a normalized floating-point number X with n bit exponent (except 0) is:

$$2^{-2^{n-1}} \leq X < 2^{(2^{n-1}-1)} \qquad (3.3)$$

In real -time processing, typically 8 bit, 12 bit or 16 bit fixed-point numbers are used to represent the input and output signals. The fixed decimal point is generally placed to the

right of the sign bit so the input and output signals span the range from –1 to +1. Thus, the smallest (absolute value) number that occurs for 16 bits is

$$2^{-15} \approx 3.05 \times 10^{-5}$$

The number of exponent bits used should at least cover the total range of input and output with normalized floating-point values. From (3.3), n=4 gives a smallest normalized number of

$$2^{-9} \approx 1.95 \times 10^{-3}$$

while n=5 gives

$$2^{-16} \approx 1.53 \times 10^{-5}$$

so n=5 is selected.

Using the state-space structure, the roundoff noise of H1 and H2 was computed using functions in Table 3.1 with a different number of mantissa bits (from 8 bits to 22 bits) of the small floating-point representation. The results are listed in Table 3.2. The roundoff noise for the direct form structure is also given in Table 3.2. Comparing the roundoff noise of the two kinds of realization we can see that the state-space structure produces a smaller roundoff noise for the same number of mantissa bits.

The number of mantissa bits needed in the realization depends on the number of bits of input precision. There would be little or no advantage in making the roundoff noise variance lower than the input quantization noise variance. In [2] the quantization variance is given by

Table 3.2: Roundoff noise variance of state-space and direct-form structure

Mantissa bits (including a sign bit)	Roundoff noise variance of H1		Roundoff noise variance of H2	
	state-space structure	direct form structure	state-space structure	direct-form structure
8	6.3039E-05	2.53045E-02	2.92337E-05	1.22083E-04
9	2.40985E-05	4.88545E-02	9.07938E-06	5.00610E-05
10	6.72145E-06	4.04196E-04	2.51201E-06	1.50350E-05
11	1.97356E-06	3.08152E-04	6.56507E-07	4.12776E-06
12	5.16057E-07	1.45275E-04	7.11904E-07	1.13272E-06
13	1.38756E-07	1.25903E-05	4.15996E-08	2.77123E-07
14	3.46750E-08	3.36372E-06	1.05520E-08	7.00174E-08
15	8.61270E-09	8.26125E-07	2.61236E-09	1.70071E-08
16	2.17600E-09	2.13232E-07	6.50243E-10	4.49218E-09
17	5.44317E-10	4.88226E-08	1.65287E-10	1.09815E-09
18	1.35793E-10	1.44584E-08	4.06431E-11	2.74165E-10
19	3.33635E-11	3.89485E-09	1.02875E-11	6.53998E-11
20	8.21505E-12	8.34378E-10	2.51779E-12	1.79114E-11
21	2.13542E-12	2.07915E-10	6.76372E-13	5.11819E-12
22	6.49970E-13	5.09586E-11	2.18093E-13	2.35134E-12

$$\sigma_q^2 = \frac{q^2}{12} = \frac{(2^{-(B-1)})^2}{12} = \frac{2^{-2B}}{3} \qquad (3.4)$$

where q is the quantization step size and B is the number of input bits. Typical inputs have either 8, 12 or 16 bits of precision giving the quantization noise variance listed in Table 3.3.

Keeping the round-off noise variance in Table3.2 below the input quantization noise in Table 3.3, we can find how many bits of mantissa are needed in small floating-point format to implement function H1 and H2 using the state-space structure as well as direct-from structure. The result is summarized in Table 3.4

Table 3.3 Quantization noise variance of input

Input bits	8	12	16
Quantization noise variance	5.08626E-06	1.98682E-08	7.76102E-11

Table 3.4 Mantissa bits needed for 8,12 or 16 bits input

Input bits	Mantissa bits of state-space structure		Mantissa bits of direct form structure	
	H1	H2	H1	H2
8	10	9	13	11
12	14	13	17	15
16	18	17	21	19

These results show that different transfer functions may need different mantissa bits because their roundoff noise is different. The state-space structure requires fewer mantissa bits than the direct form structure to keep the round-off noise below the input quantization error. Thus, the state-space structure is chosen in the realization of the IIR filter. In the following chapters the realization of the IIR filter using state-space structure will be discussed in detail and H1 will be used as an example.

Chapter IV

FLOATING-POINT MULTIPLIER, ADDER DESIGN AND NORMALIZER DESIGN

Since the logic resources and speed of the filters are mainly determined by the floating-point multiplier and adder used in the filter implementation, it is necessary to first study their implementation. Thus, the floating-point arithmetic (multiplier and adder) used in filter design will be discussed in detail in this chapter.

4.1 Floating-Point Multiplier Realization

Floating-point multiplication algorithm is comparatively easy. The multiplication formula is:

$$(SA \times MA \times 2^{EA}) \times (SB \times MB \times 2^{EB}) = (SA \times SB) \times (MA \times MB) \times 2^{EA+EB} \qquad (4.1)$$

where SA and SB are sign bits, MA and MB represent mantissas and EA and EB are exponents.

4.1.1 Altera bit-parallel floating-point multiplier function

The Altera fp_mult function is written in AHDL, and implements a high-speed floating-point multiplier with parameterized input widths. This function uses sign-mantissa-exponent notation with parameterized mantissa and exponent bit widths as shown in Figure 4.1.

Figure 4.2 shows the fp_mult block diagram. To multiply the floating-point numbers, the mantissas are first multiplied. (In fp_mult a bit-parallel fixed-point LPM multiplier function is used. We call this multiplier the bit-parallel multiplier.) Then, the exponents are added, and the excess value $2^{(n-1)}$ is subtracted from the result.

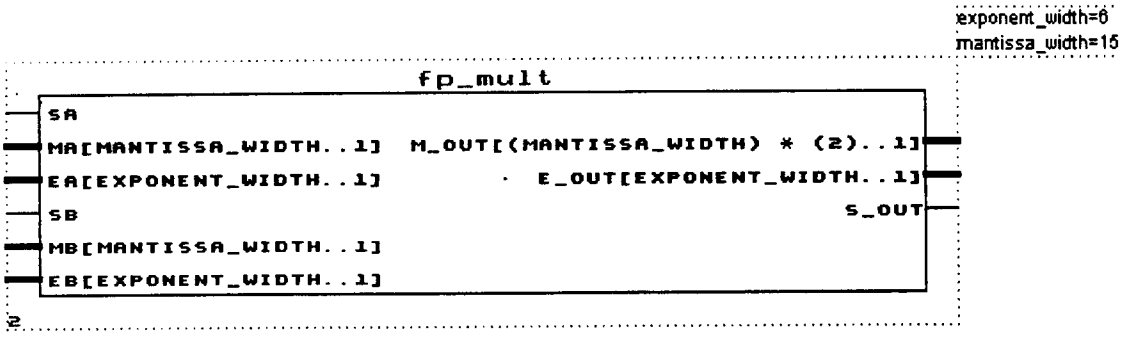


Figure 4.1 Schematic interface of fp_mult

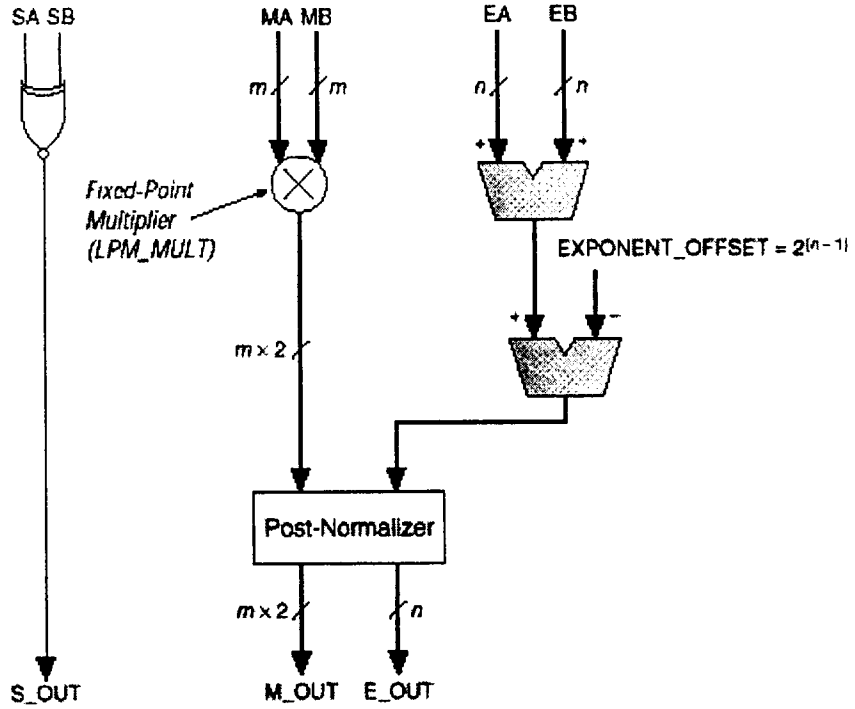


Figure 4.2 Block diagram of fp_mult

The sign of the output (S_OUT) is the XNOR of the signs of the inputs (SA and SB). After multiplication has taken place, the post-normalizer normalizes the result, if necessary, by adjusting the mantissa and exponent of the result to ensure that the MSB of the mantissa is 1.

Compiling this function using MAX+PLUS II can determine the logic resources required and the speed achieved. The FAST MODE synthesis style is used and the EPF10K50RC240-3 device is chosen in the fitting used to determine speed. The number of exponent bit is chosen to be 5 and the number of mantissa bits varies from 9 to 19 bits. The results are shown in Table 4.1.

Table 4.1 Logic cells used and speed of computation of fp_mult (5 bit expoent)

Mantissa bits	Logic cells	Mult/sec. (M)
9	258	13.07
10	312	12.40
11	367	11.62
12	430	10.69
13	494	10.81
14	563	10.36
15	635	10.22
16	712	9.15
17	800	8.58
18	886	8.26
19	979	8.50

Using the fp_mult function in a design produces a "double precision" output(the mantissa in the result has twice the number of bits of either input mantissa). Therefore, the result does not lose precision and does not require rounding. But in the filter design the result will be truncated to the number of input mantissa bits so it can be used by the floating-point adder in subsequent computations.

The fp_mult function does not check for the error conditions of overflow and underflow because this would add significant complexity and delays to the circuit. In filter implementations overflow will virtually never happen because with a 5 bit exponent, the maximum number is 65536, which is unlikely to ever be reached in filter computations when the input is less than 1. However underflow can happen if the exponent of the result is less than 15 which can happen for small or zero inputs. Therefore this megafunction must be modified to handle underflow.

4.1.2 Improved bit-parallel floating-point multiplier

Normally the result will be set to 0 if underflow occurs in a DSP computation. In our case underflow occurs when the MSB of the exponents of two multiplying numbers are both negative, but the MSB of the exponent of the result after post-normalization is positive. A control signal *underflow* is added to fp_mult function:

underflow=!(MSB of EA) and !(MSB of EB) and (MSB of postnormalization correction)

After post-normalization (see Figure 4.2) *underflow* signal is checked. If *underflow* is 1, then the result will be set to 0.

After carefully studying the Altera fp_mult function, it is found that the post-normalizer part in Figure 4.2 can be improved to make the multiplier more efficient in the filter implementation. In the fp_mult function, after getting the multiplication result of two mantissas, a double-length normalizer is used to normalize the result since a double length output mantissa is produced. This double-length normalizer uses many logic cells and reduces the speed of computation. Also, the normalizer complexity is increased since denormalized multiplier inputs are allowed. If denormalized representations are not allowed, the two mantissas are between 0.5 and 1 (except 0), so the result of the multiplication is between 0.25 and 1(except 0). This means that the normalizer can only check the MSB of the result. If the first MSB is 1, we do not need to adjust the exponent. If the first MSB is 0, the output mantissa is left shifted one place and the exponent is reduced by 1.

The improved code, written in AHDL, is in Appendix A2. We call this function fp1_mult. It only outputs the single-length MSBs of the result. Figure 4.3 shows the schematic design of fp1_mult. The logic cells and speed of computation are list in Table 4.2 using the same fitting choice as Section 4.1.1.

Compared with Table 4.1, we can see the improved bit-parallel floating-point multiplier significantly decreases the use of logic cells and increases the speed of computation. However, the number of logic cells it yields is still very large because of using the bit-parallel multiplier.

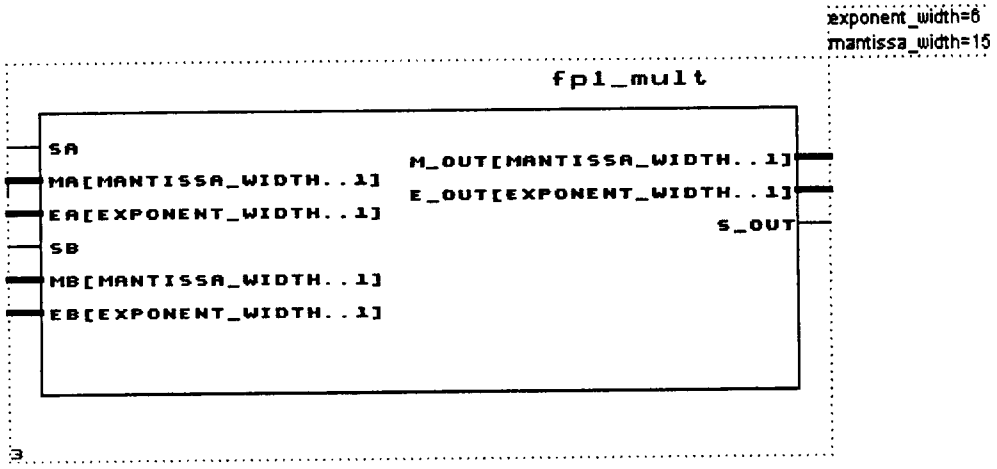


Figure 4.3 Schematic interface of fp1_mult

Table 4.2 Logic cells used and speed of computation of fp1_mult

Mantissa bits	Logic cells	Mult/second (M)
9	195	18.79
10	236	18.83
11	280	18.38
12	332	18.48
13	382	17.73
14	442	17.73
15	502	17.48
16	570	16.12
17	635	14.51
18	708	12.98
19	784	13.47

4.1.3 Bit-Serial Floating-Point Multiplier

The bit-parallel multiplier can increase the speed but uses a large number of logic cells. If the use of logic resources is a bigger concern than speed, a bit-serial multiplier can be used instead of bit-parallel. The fixed-point bit-serial approach applies one fixed-point number to the multiplier in parallel form and another in serial form, LSB first. The product is output serially, also with the LSB first. It will take twice as many clock cycles to finish a multiplication as there are input mantissa bits. For example, a 16 by 16 bit-serial multiplier will take 32 clock cycles. After 16 clock cycles, the 16 LSBs of the product are out and 16 more are needed to clock out the MSBs.

In [5] the fixed-point bit-serial multiplier algorithm and implementation in FLEX FPGAs were described in detail. Although it takes several clock cycles to get the result, the bit-serial method is not proportionately slower because the clock rate can be very high. The clock rate of a bit-serial two's complement multiplier using Booth's algorithm, optimized for the FLEX 10K architecture can reach 125MHz due to the simplicity of the bit-serial hardware. This fixed-point bit-serial multiplier can replace the LPM_MULT in Figure 4.2 to generate the bit-serial floating-point multiplier by adding some control signals, a shift-register to shift the serial data in, and a shift-register to hold the serial-out data.

Figure 4.4 gives a brief block diagram to show part of the design. The mult_ctrl block is a state machine that controls the process. The input signal *mult_enable* begins the multiplication. The state machine generates control signals *load_in, load_out, shift* and *mult_done*. The serial and parallel registers get the two multiplying numbers at the

rising edge of *load_in*. The outputs of the two registers go to the bit-serial multiplier. The *shift* signal controls the bit computation. A shift register catches the output bits of the bit-serial multiplier. It outputs the result when *load_out* is high. This part can replace the LPM_MULT in Figure 4.2 to generate the bit-serial floating-point multiplier which we call bs_mult. The total schematic design and the VHDL codes of the component are in given Appendix 3.

The interface of the bit-serial multiplier is like that of the bit-parallel multiplier except that the bit-serial multiplier has a *mult_enable* signal to trigger the beginning of multiplication and a *mult_done* signal to tell when the multiplication is finished. The schematic interface is shown in Figure 4.5.

The logic resources used and speed achieved by the bit-serial floating-point multiplier are listed in Table 4.3. For the fitting design the clock rate can reach 63.69Mhz, which is slower than 125 MHz because of the control signals and the shift registers as shown in Figure 4.4. In Table 4.3 the speed is the frequency of one total multiplication. From the data we can see that with the increment of one mantissa bit, the number of logic cells only increases 7 because of the increase of bits of shift registers. This saves a significant numbers of logic cells compared with the bit-parallel multiplier, especially when the number of mantissa bits is large. The speed of total computation is over 1 MHz.

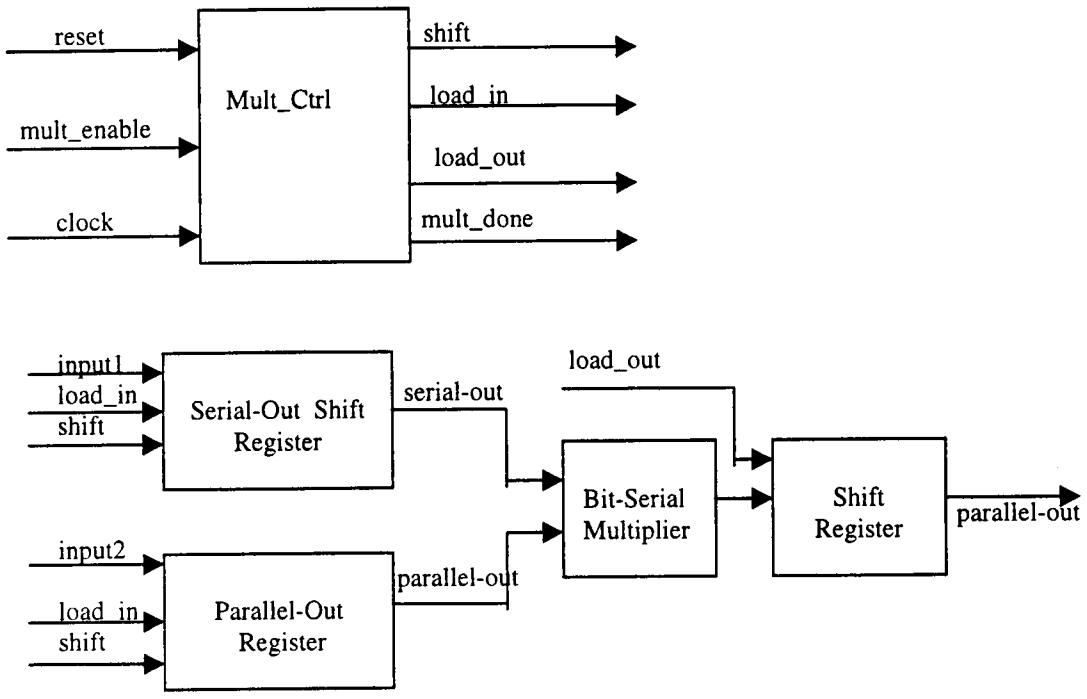


Figure 4.4 Block diagram of bit-serial multiplier

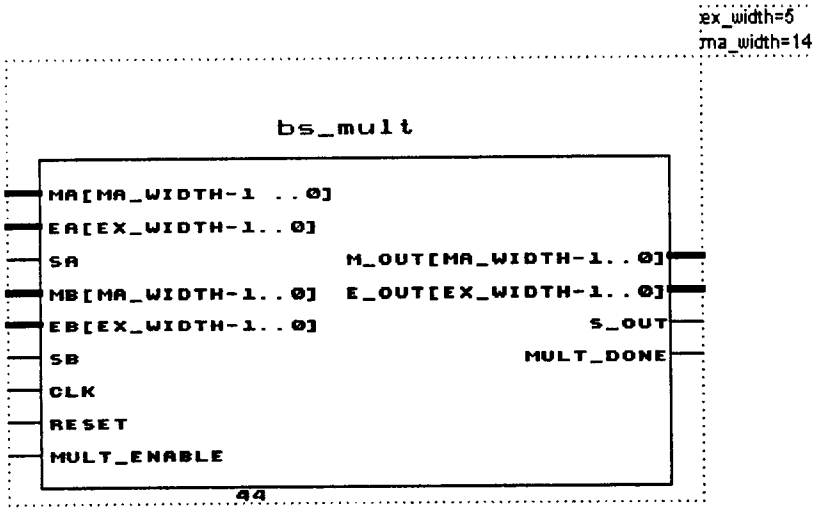


Figure 4.5 Schematic interface of bs_mult

Table 4.3 Logic cells used and speed of computation of bs_mult

Mantissa bits	Logic cells	Clock Cycles	Mult/sec (M)
9	126	24	2.68
10	133	26	2.34
11	140	28	2.26
12	147	30	1.99
13	154	32	1.86
14	161	34	1.84
15	168	36	1.73
16	175	38	1.64
17	182	40	1.50
18	189	42	1.48
19	196	44	1.44

4.2 Floating-point Adder Realization

Floating-point addition is more complex than multiplication because, depending on the signs of the operands, it may actually be a subtraction. If it is an addition, there can be carry-out on the left. If it is subtraction, there can be cancellation. The detailed algorithm can be found in [11].

Altera has a megafunction called fp_add_sub which implements a floating-point adder/substractor with parameterized input widths. The function uses sign-mantissa-exponent notation with parameterized mantissa and exponent widths as shown in Figure 4.6. In our case we only need addition, so the ADD_SUB can be set to 1.

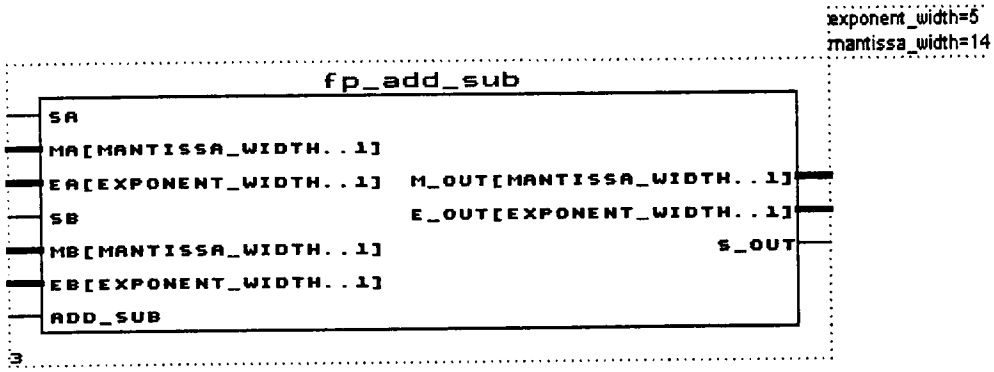


Figure 4.6 Schematic interface of fp_add_sub

To add or subtract two floating-point numbers, the mantissas must be aligned. The relative values of the exponents are checked by subtracting one exponent from the other as shown in Figure 4.7. The mantissa with the larger exponent is retained, and the mantissa with the smaller exponent is right-shifted until the radix point is properly aligned (i.e., until the exponents are equal). If the exponents differ by more than the number of bits in the mantissa, the smaller number becomes insignificant. The shifting is performed by the LPM function lpm_clshift.

After the mantissas pass through the shifters, an unsigned integer add/subtractor performs an operation that is determined by the sign of the inputs (SA and SB) and the add_sub port. The result of the adder is then passed through a programmable inverter, which is controlled by the sign decision logic. This process ensures that the mantissa has the proper sign. After the addition or subtraction has taken place, the post-normalizer normalizes the result, if necessary, by adjusting the mantissa and exponent of the result so that the MSB of the mantissa is 1. The block diagram is shown in Figure 4.7.

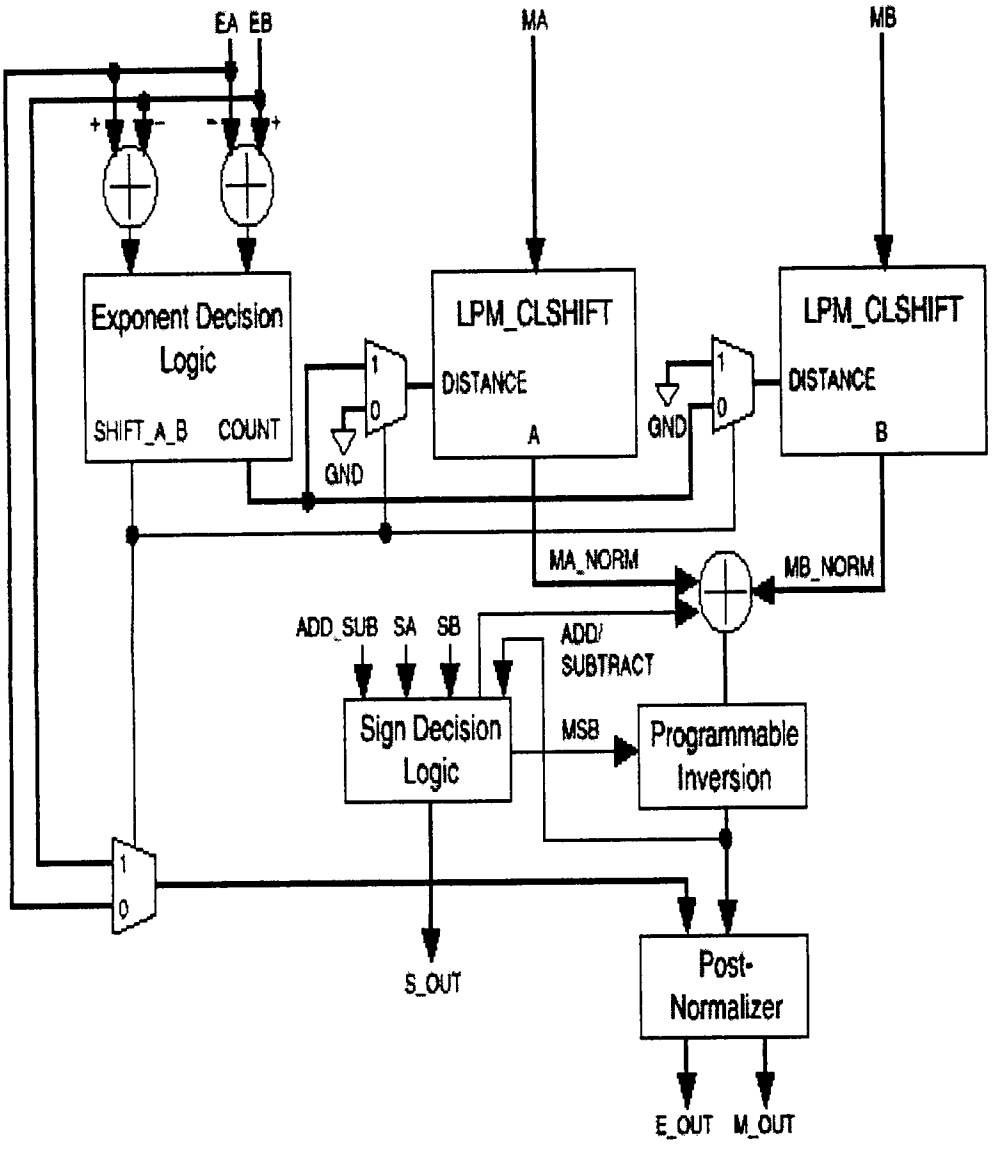


Figure 4.7 Block diagram of fp_add_sub

The logic resources and speed of computation can be determined by MAXPLUS II. Using the same synthesis style and device mentioned in Section 4.1.1, the result is in Table 4.4. From the result we can see that although the algorithm is complex, floating-point addition actually uses much fewer logic cells than floating point bit-parallel multiplication. This megafunction will be used in the filter design in the next chapter.

4.3 Simplifying the Floating-Point Multiply-Add Operation

Due to the complexity of the floating-point addition algorithm, a method for using a large fixed-point adder in place of the floating-point adder was considered. This seemed reasonable because one logic cell in FPGA can implement one-bit of addition using the fast carry chain in the FPGA, so the fixed-point addition of a large number of bits is not cells consuming. If one very large fixed-point adder is used, normalization of the floating-point multiplier output and denormalization of the floating-point adder input might be avoided. This will reduce the complexity of implementation. However, after a trial this method was found not to be practical. The large adder also needs a large normalizer to normalize its output, which uses many logic cells and reduces the speed.

4.4 Normalizer Design

The multipliers discussed above need normalized input numbers. From Figure 2.4 we know that there will be a multiplication of the input and the coefficient d, so the input needs to be normalized before it goes into a multiplier. An Altera megafunction fp_normalizer is used to normalize the input; it shifts the fixed-point number left until the MSB is 1. A counter is used to record the shifts required, which will be adjusted to be the exponent part. The logic cells required are listed in Table 4.5 (exponent bits is 5).

Table 4.4 Logic cells used and speed of computation of fp_add_sub

Mantissa bits	Logic cells	Add/sec (M)
9	141	22.37
10	153	20.20
11	168	19.8
12	182	20.66
13	197	20.36
14	208	18.48
15	224	19.84
16	237	16.89
17	265	17.85
18	281	16.42
19	298	15.79

Table 4.5 Logic cells used in input normalization

Input bits	8	12	16
Logic Cells	32	50	66

Chapter V

SECOND-ORDER FLOATING-POINT IIR FILTER IMPLEMENTATION

5.1 Filter Design Using Floating-Point Bit-Parallel Multiplier

First bit-parallel floating-point multiplier fpl_mult and floating-point adder fp_add_sub will be used to implement a second-order IIR filter. To save logic resources, only one multiplier and one adder are used. Thus, it takes several clock cycles to finish the computations required for each filter input sample.

The block diagram of the filter implementation is shown in Figure 5.1. Besides the multiplier and adder, several registers, multiplexes, a control state machine, and a coefficient part are used. The description of the components is listed in Table 5.1.

In one complete filter computation seven multiplications and six additions are needed. In one clock cycle the diagram in Figure 5.1 executes one multiplication and one addition. This will be called the unpipelined structure. When overhead for data movement is taken into account, nine clock cycles are required to finish a computation. The behavior of the components in every clock cycle is listed in Table 5.2.

The control state machine generates the signals to control the behavior of registers, multiplexes, and coefficient selection and is shown in Figure 5.2. The registers and multiplexers are designed with parameterized mantissa and exponent bit widths like the multiplier and adder. By adjusting these parameters and the coefficients, operations with different precisions can be easily implemented. The schematic design of Figure 5.1 and the component design in VHDL are included in Appendix A4.

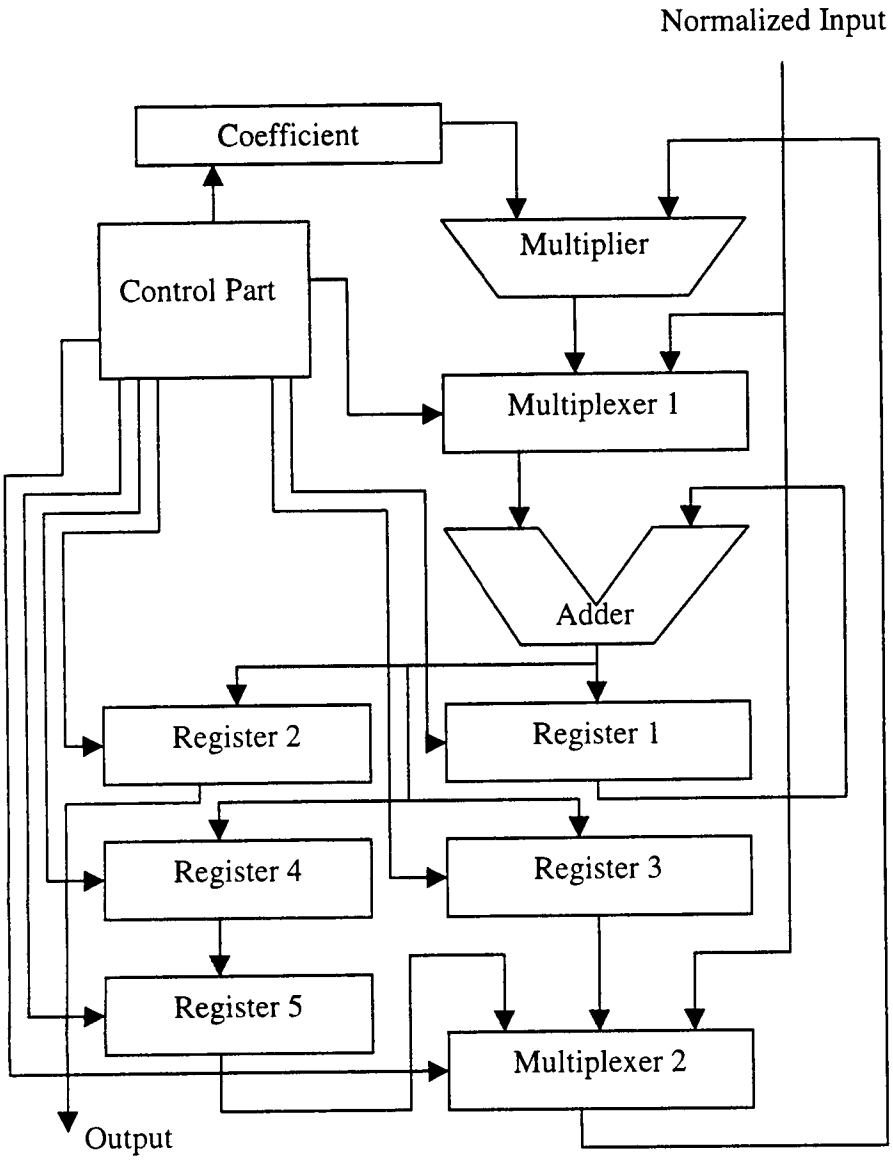


Figure 5.1 Block diagram of unpipelined second-order IIR filter

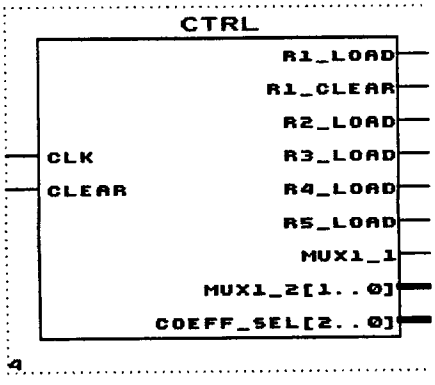


Figure 5.2 Schematic interface of control part using bit-parallel multiplier

Table 5.1 Function description of filter component

Component	Function Description
Multiplier	Realize bit-parallel floating-point multiplication
Adder	Realize floating-point addition
Register1	Hold the addition result
Register2	Hold the filter output data
Register3	Hold state variable $x_1[k]$ or $x_2[k+1]$
Register4	Hold state variable $x_1[k+1]$
Register5	Hold state variable $x_2[k]$
Multiplexer1	Choose from input or multiplication result
Multiplexer2	Choose state variable $x_1[k]$ or $x_2[k]$ or input
Coefficient	Store the floating-point coefficients.
Control state machine	Control the total process, including register loading and clearing, multiplexer controlling , and coefficient choosing

Table 5.2 Unpipelined filter component behaviors in different clock cycles

Clock Cycle	Coeff	R1	R2	R3	R4	R5	Mux1	Mux2
1	c_1	clear	hold	load	hold	hold	multiplier	$x_1[k]$
2	c_2	load	hold	hold	hold	hold	multiplier	$x_2[k]$
3	d	load	hold	hold	hold	hold	multiplier	input
4	a_{11}	clear	load	hold	hold	hold	multiplier	$x_1[k]$
5	a_{12}	load	hold	hold	hold	hold	multiplier	$x_2[k]$
6	--	load	hold	hold	hold	hold	input	--
7	a_{21}	clear	hold	hold	load	hold	multiplier	$x_1[k]$
8	a_{22}	load	hold	hold	hold	load	multiplier	$x_2[k]$
9	--	load	hold	hold	hold	hold	input	--

If a register is added between multiplier and multiplexer1, multiplication and addition can be pipelined. This improves the clock rate. The pipelined filter is shown in Figure 5.3 and takes ten clock cycles to finish a computation. The behavior of components in different clock cycles is shown in Table 5.3. The detailed design appears in Appendix A5.

5.2 Filter Design Using Floating-Point Bit-Serial Multiplier

If speed is not a big concern, the bit-serial multiplier designed in Chapter IV can be used in place of the bit-parallel multiplier. There should be two clocks in this case. One clock is used in the control state machine of the bit-serial multiplier, which can reach 63.69 MHz (see Chapter IV). The other is used in the total computation control state machine, which can reach over 10 Mhz. The block diagram is the same as using the pipelined bit-parallel multiplier. The control part should add a control signal to trigger the multiplier to begin computation. After computation is done the multiplier sends a signal mult_done back to the control part to begin the next step. The control part schematic interface is shown in Figure 5.4.

Appendix A6 shows the schematic design of the IIR filter using the bit-serial multiplier as well as the control part designed in VHDL.

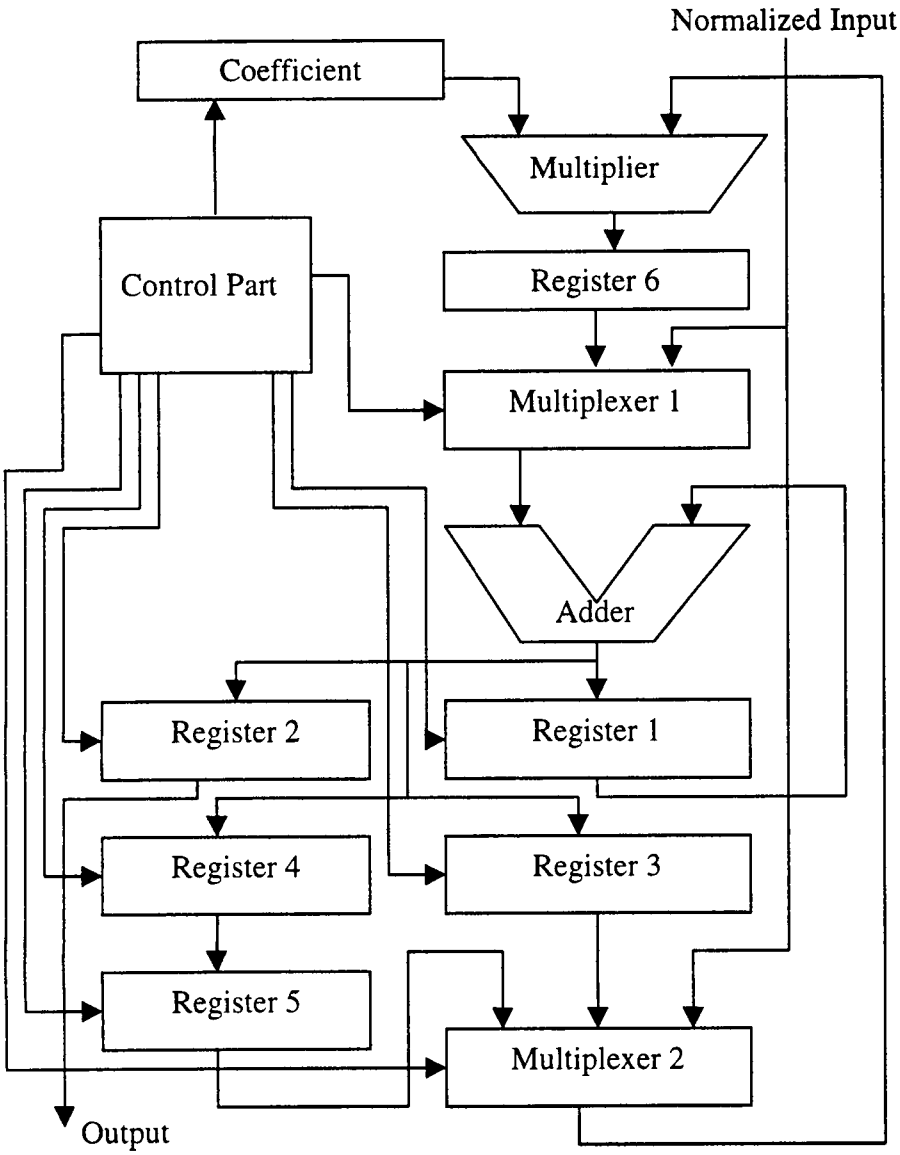


Figure 5.3 Block diagram of pipelined second-order IIR filter

Clock Cycle	Coeff	R1	R2	R3	R4	R5	R6	Mux1	Mux2
1	c_1	clear	hold	load	hold	hold	--	--	$x_1[k]$
2	c_2	clear	hold	hold	hold	hold	load	multiplier	$x_2[k]$
3	d	load	hold	hold	hold	hold	load	multiplier	input
4	--	load	hold	hold	hold	hold	load	multiplier	--
5	a_{11}	clear	load	hold	hold	hold	load	input	$x_1[k]$
6	a_{12}	load	hold	hold	hold	hold	load	multiplier	$x_2[k]$
7	--	load	hold	hold	hold	hold	--	multiplier	--
8	a_{21}	clear	hold	hold	load	hold	--	input	$x_1[k]$
9	a_{22}	load	hold	hold	hold	hold	load	multiplier	$x_2[k]$
10	--	load	hold	hold	hold	load	load	multiplier	--

Table 5.3 Pipelined filter component behaviors in different clock cycles

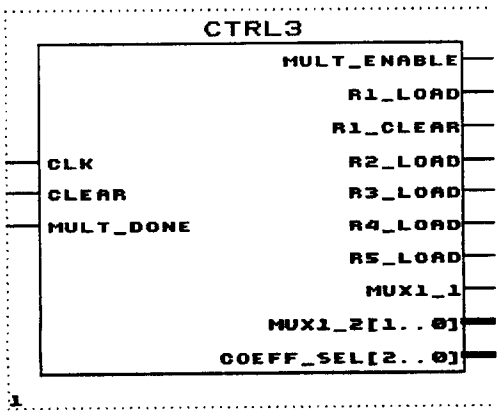


Figure 5.4 Schematic interface of control part using bit-serial multiplier

5.3 Result Analysis

The transfer function H1 (see Chapter III) was chosen to be implemented in the FPGA. The input precision is taken to be 12 bits. From Table 3.5 it is found that at least 14 bits of mantissa should be used to keep the roundoff noise below the input quantization error. The state-space coefficients can be computed using the program float_state_space.cc (see Chapter III). The result is shown in Table 5.4.

Adjusting the mantissa_width and exponent_width parameters and the coefficients of the design gives the implementation of the IIR filter with transfer function H1. The logic cells and computation speed with different implementations is shown in Table 5.5. From the results we can see the pipelined bit-parallel design uses almost the same logic resources but is 1.7 times faster compared with unpipelined bit-parallel design. The bit-serial design used less logic cells but the speed is low.

The impulse response output of the designed filter is given in Table 5.6; it also lists the result attained using double precision floating-point arithmetic. Only 40 outputs are listed above. From the result we see that the output of designed IIR filter is very close to the double precision result.

Table 5.4 State-space structure coefficient of H1

Coefficient	Binary Representation			Floating-pint Representation
	sign	mantissa	Exponent	
a_{11}	1	11111010010100	10000	0.9777890
a_{12}	0	10110101111101	01011	-0.0222111
a_{21}	1	10101110001000	01011	0.2125590
a_{22}	1	11111010010100	10000	0.9777890
c_1	0	10110001111101	01011	-0.0217229
c_2	0	10110001111101	01011	-0.0217229
d	1	11111010011000	10000	0.9780273

Table 5.5 Logic cells and speed in different implementation of H1

Method	Logic Cells	Samples/second
Unpipelined, Bit-Parallel	886	6.24M/9=693K
Pipelined, Bit-Parallel	887	10.70M/10=1.07 M
Bit-Serial	675	≈200 K

Table 5.6 Impulse response of H1

Output Number	Output (by FPGA)	Output(by computer)	Difference(%)
1	0.9780273	0.9780305	0.0003
2	-0.0433655	-0.0434456	0.1844
3	-0.0424500	-0.0424596	0.0226
4	-0.0414467	-0.0414755	0.0694
5	-0.0404358	-0.0404941	0.1439
6	-0.0394592	-0.0395162	0.1442
7	-0.0384827	-0.0385427	0.1556
8	-0.0375366	-0.0375743	0.1003
9	-0.0365295	-0.0366170	0.2389
10	-0.0356140	-0.0356556	0.1166
11	-0.0346680	-0.0347066	0.1112
12	-0.0336914	-0.0337654	0.2191
13	-0.0327759	-0.0328325	0.1723
14	-0.0318604	-0.0319084	0.1504
15	-0.0309753	-0.0309936	0.0590
16	-0.0300293	-0.0300887	0.1974
17	-0.0291138	-0.0291941	0.2750
18	-0.0282898	-0.0283102	0.0720
19	-0.0273743	-0.0274373	0.2296
20	-0.0264893	-0.0265759	0.3258
21	-0.0256956	-0.0257263	0.1193
22	-0.0248108	-0.0248887	0.3129
23	-0.0239868	-0.0240636	0.3191
24	-0.0231628	-0.0232510	0.3793
25	-0.0224000	-0.0224514	0.2289
26	-0.021606	-0.0216648	0.2714
27	-0.0208435	-0.0208914	0.2292
28	-0.0200500	-0.0201316	0.4053
29	-0.0193176	-0.0193853	0.3492
30	-0.0185852	-0.0186527	0.3618
31	-0.0178529	-0.0179340	0.4522
32	-0.0171814	-0.0172291	0.2768
33	-0.0164490	-0.0165383	0.5399
34	-0.0157776	-0.0158616	0.5295
35	-0.0151062	-0.0151989	0.1296
36	-0.0144653	-0.0145504	0.5848
37	-0.0139161	-0.0139160	0.0007
38	-0.0133056	-0.0132957	0.0744
39	-0.0126953	-0.0126896	0.0449
40	-0.0120850	-0.0120975	0.1033

Chapter VI

CONCLUSION AND RECOMMENDATIONS

In this thesis the implementation of second-order floating-point IIR filters in current FPGAs was investigated. The state-space structure was used because of its low roundoff noise. A small word length floating-point representation was chosen to keep the roundoff noise below the input quantization error while also keeping the logic resources required as low as possible. A 5-bit exponent was found to be adequate with the number of mantissa bits dependent on the transfer function realized. Two kinds of floating-point multiplier were analyzed: bit-parallel and bit-serial. The bit-parallel approach was found to achieve high speed but used substantial logic resources, especially with the increase of mantissa bits. The bit-serial approach was found to use fewer logic cells but has much lower speed.

One floating-point multiplier and one adder were used to implement the IIR filter. Using the bit-parallel floating-point multiplier, implementation of the IIR filter takes less than 900 logic cells and the sample rate can be over 1 MHz if a 14-bit mantissa and a 5-bit exponent are chosen. This sample rate is close to that of DSP chip. For example, the sample rate is between 1 MHz to 2 MHz if a Texas Instrument C30x or C40x chip is used (also using a state-space structure) [12]. The advantage of using an FPGA is that other components like interfaces or control circuits can also be implemented in the same chip.

Logic resources can be saved by using the bit-serial multiplier, especially for a large number of mantissa bits, but the speed is comparatively lower. From the

investigation it is found that second-order floating-point IIR filters can be implemented in current FPGAs.

Because higher-order IIR filters can be realized using a parallel or cascade connection of second-order IIR filters, one second-order IIR filter structure like that discussed in the thesis can be used repeatedly to implement higher-order IIR filters. The coefficients of second-order filters can be stored in EAB which acts as a ROM. The control state machine can choose different groups of coefficients when computing different second-order subfilters.

Further research can be done to improve the speed. Two or more multipliers and adders can be used to parallel the computations if using an FPGA like EPF10K40 or higher which has more than 2000 logic cells. This would reduce the clock cycles required for one computation. Another recommendation is to implement the floating-point multiplier and adder in several stages, which will further pipeline the computation to achieve a higher clock rate [6]. In this case, the block state structure [14] can be used for pipelined computation.

APPENDICES

A1

COMPUTER SIMULATION PROGRAM TO COMPUTE ROUNDOFF NOISE

```
/*****************************************************************************/
/*                          Program :  standard_state_space.cc                */
/* Using state_space structure to compute the second-order IIR filter result  The input and coefficient*/
/* are quantized to fixed bits of mantissa  and  exponent. The intermeadiate data are chosen to be    */
/* double precision. The  result is assumed to be no round-off noise.                                 */
/*****************************************************************************/
#include<stdio.h>
#include<iostream.h>
#include<math.h>
#include<fstream.h>
#include<stdlib.h>
#include<iomanip.h>
main()
{
  double alpha1,alpha2,beta1,beta2;
  double a11,a12,a21,a22,b1,b2,c1,c2,d;
  double A11,A12,A21,A22,C1,C2;
  double u,r,w,v,e,scale1,scale2;
  double sum1,sum2;
  char inputfilename[32],outputfilename[32];
  ifstream  readfile;
  ofstream  writefile;
  int  i,N_input,N_fix,N_ex;
  double input,xk1,xk2,yk;
  void normlize(double *f, int N_bits,int N_ex,int print);
  cout<<"Please enter the transfer function coefficient:"<<endl;
  cout<<"     alpha1="; cin>>alpha1; cout<<endl;
  cout<<"     alpha2="; cin>>alpha2; cout<<endl;
  cout<<"     beta1 ="; cin>>beta1 ; cout<<endl;
  cout<<"     beta2 ="; cin>>beta2 ; cout<<endl;
  cout<<"       d   ="; cin>>d     ; cout<<endl;

 //get the state-space coefficient
   a11=-beta1*0.5;
   a22=a11;
   u=sqrt((alpha2/alpha1)*(alpha2/alpha1)-(alpha2/alpha1)*beta1+beta2);
   r=alpha2/alpha1-u;
   w=alpha2/alpha1+u;
   v=(beta2-1)*((beta2+1)*(beta2+1)-beta1*beta1);
   e=(beta1/2)*(beta1/2)-beta2;
   b1=sqrt(v/(2.0*beta1*r-(beta2+1)*(1.0+r*r)));
   b2=sqrt(v/(2.0*beta1*w-(beta2+1)*(1.0+w*w)));
   a21=sqrt(((b2*b2+beta2-1)/(b1*b1+beta2-1))*e);
   a12=e/a21;
   c1=alpha1/(2.0*b1);
   c2=alpha1/(2.0*b2);

   cout<<"c1="<<c1<<"    "<<"c2="<<c2<<endl;
```

```
//scale to let b1=b2=1;
  a12=a12*b2/b1;
  a21=a21*b1/b2;
  c1=b1*c1;
  c2=b2*c2;
  A11=a11;
  A12=a12;
  A21=a21;
  A22=a22;
  C1=c1;
  C2=c2;

  cout<<"a11="<<a11<<endl;
  cout<<"a12="<<a12<<endl;
  cout<<"a22="<<a21<<endl;
  cout<<"a22="<<a22<<endl;
  cout<<"c1= "<<c1<<endl;
  cout<<"c2= "<<c2<<endl;

for(int m=0; m<20;m++) {

  a11=A11;
  a12=A12;
  a21=A21;
  a22=A22;
  c1=C1;
  c2=C2;
  cout<<"Please enter fixed bits:";    cin>>N_fix; cout<<endl;
  cout<<"Please enter exponent bits:"; cin>>N_ex;   cout<<endl;
  normlize(&a11,N_fix,N_ex,1);   cout<<"    "<<"a11="<<a11<<"     "<<endl;
  normlize(&a12,N_fix,N_ex,1);   cout<<"    "<<"a12="<<a12<<"     "<<endl;
  normlize(&a21,N_fix,N_ex,1);   cout<<"    "<<"a22="<<a21<<"     "<<endl;
  normlize(&a22,N_fix,N_ex,1);   cout<<"    "<<"a22="<<a22<<"     "<<endl;
  normlize(&c1,N_fix,N_ex,1);    cout<<"    "<<"c1= "<<c1<<"      "<<endl;
  normlize(&c2,N_fix,N_ex,1);    cout<<"    "<<"c2= "<<c2<<"      "<<endl;

  cout<<"Please enter input data file name:"; cin>>inputfilename; cout<<endl;
  readfile.open(inputfilename,ios::in);
  readfile>>N_input;
  cout<<"Please enter output file name   :"; cin>>outputfilename;cout<<endl;
  writefile.open(outputfilename,ios::out);
  sum1=0.0;
  sum2=0.0;
  for(i=0;i<N_input;i++)
  {
     xk1=sum1;
     xk2=sum2;
```

```
    readfile>>input;
    normlize(&input,N_fix,N_ex,0);
    sum1=xk1*a11+xk2*a12+input;
    sum2=xk1*a21+xk2*a22+input;
    yk=c1*xk1+c2*xk2+input*d;
    //normlize(&yk,N_bits,N_ex,0);
    writefile<<i<<"          "<<yk<<endl;
  }

  readfile.close();
  writefile.close();
 }


}

/* Function normlize change the double precsion floating-point to different bits
  of mantissa and exponent according to the parameter N_bits and N_ex.N_bits is
  the number of bits of mantissa. N_ex is the number of bits of exponent */
void normlize(double *f, int N_bits,int N_ex,int print=0)
{
  int f_bit_array[50];
  float mantissa, step, new_mantissa,ff;
  int i,j, exponent, sign, round;
  int  f_to_int;

//get the mantissa and exponent in floating-point
  mantissa=(float) frexp(*f, &exponent);

//get the N_bits assigned fixed-point mantissa
  if(mantissa<0) {
     mantissa=-mantissa;
     sign=1; }
  else
     sign=0;
  step=1.0;

  j=N_bits-1;

  for(i=0;i<j;i++) {
    step=step*0.5;
    if(mantissa>=step) {
       mantissa=mantissa-step;
       f_bit_array[i]=1; }
    else
       f_bit_array[i]=0;
```

```
      if(print==1) cout<<f_bit_array[i];
 }

 new_mantissa=0.0;
 step=1.0;
 for(i=0;i<j;i++)
 {
    step=step*0.5;
    new_mantissa=new_mantissa+f_bit_array[i]*step;
 }
 if(sign==1) new_mantissa=-new_mantissa;
 ff=new_mantissa*pow(2, exponent);
 if(print==1) cout<<"     "<<"exponent="<<exponent<<endl;

 if(abs(exponent)>=pow(2,(N_ex-1)))
   *f=0.0;
 else
   *f=ff;
}
```

```
/*****************************************************************************/
/*                          Program : small_state_space.cc                   */
/* Using state_space structure to compute the second-order IIR filter result  The input and coefficient */
/* are quantized to fixed bits of  mantissa  and  exponent. The intermeadiate data are also quantised to*/
/* fixed  bits  of  mantissa and expoent. This simulates the result in real FPGA design.The  result has*/
/* round-off noise.                                                          */
/*****************************************************************************/



#include<stdio.h>
#include<iostream.h>
#include<math.h>
#include<fstream.h>
#include<stdlib.h>

main()
{
  double alpha1,alpha2,beta1,beta2;
  double a11,a12,a21,a22,b1,b2,c1,c2,d;
  double u,r,w,v,e,scale1,scale2;
  double sum1,sum2,mantissa;
  double temp1,temp2,temp3,temp4,temp5,temp6,temp7;
  char inputfilename[32],outputfilename[32];
  ifstream  readfile;
  ofstream  writefile;
  int  i,j,N_input,N_fix,N_ex, exponent;
  double input,xk1,xk2,yk;
  void normlize(double *,int,int,int);
  float multiply(double *,double,double,int,int);
  cout<<"Please enter the transfer function coefficient:"<<endl;
  cout<<"     alpha1="; cin>>alpha1; cout<<endl;
  cout<<"     alpha2="; cin>>alpha2; cout<<endl;
  cout<<"     beta1 ="; cin>>beta1 ; cout<<endl;
  cout<<"     beta2 ="; cin>>beta2 ; cout<<endl;
  cout<<"       d   ="; cin>>d     ; cout<<endl;


for(int m=0;m<20;m++) {
  cout<<"Please enter fixed bits you want to use:";   cin>>N_fix;cout<<endl;
  cout<<"Please enter exponent bits you want to use:";cin>>N_ex; cout<<endl;

//get the state-space coefficient
  a11=-beta1*0.5;
  a22=a11;
  u=sqrt((alpha2/alpha1)*(alpha2/alpha1)-(alpha2/alpha1)*beta1+beta2);
  r=alpha2/alpha1-u;
  w=alpha2/alpha1+u;
```

```
  v=(beta2-1)*((beta2+1)*(beta2+1)-beta1*beta1);
  e=(beta1/2)*(beta1/2)-beta2;
  b1=sqrt(v/(2.0*beta1*r-(beta2+1)*(1.0+r*r)));
  b2=sqrt(v/(2.0*beta1*w-(beta2+1)*(1.0+w*w)));
  a21=sqrt(((b2*b2+beta2-1)/(b1*b1+beta2-1))*e);
  a12=e/a21;
  c1=alpha1/(2.0*b1);
  c2=alpha1/(2.0*b2);

//scale to let b1=b2=1;
  a12=a12*b2/b1;
  a21=a21*b1/b2;
  c1=b1*c1;
  c2=b2*c2;

  cout<<"   "<<"a11="<<a11<<"   "<<endl;
  cout<<"   "<<"a12="<<a12<<"   "<<endl;
  cout<<"   "<<"a22="<<a21<<"   "<<endl;
  cout<<"   "<<"a22="<<a22<<"   "<<endl;
  cout<<"   "<<"c1= "<<c1<<"   "<<endl;
  cout<<"   "<<"c2= "<<c2<<"   "<<endl;

  normlize(&a11,N_fix,N_ex,1);  cout<<"   "<<"a11="<<a11<<"   "<<endl;
  normlize(&a12,N_fix,N_ex,1);  cout<<"   "<<"a12="<<a12<<"   "<<endl;
  normlize(&a21,N_fix,N_ex,1);  cout<<"   "<<"a22="<<a21<<"   "<<endl;
  normlize(&a22,N_fix,N_ex,1);  cout<<"   "<<"a22="<<a22<<"   "<<endl;
  normlize(&c1,N_fix,N_ex,1);   cout<<"   "<<"c1= "<<c1<<"   "<<endl;
  normlize(&c2,N_fix,N_ex,1);   cout<<"   "<<"c2= "<<c2<<"   "<<endl;


  cout<<"Please enter input data file name:"; cin>>inputfilename; cout<<endl;
  readfile.open(inputfilename,ios::in);
  readfile>>N_input;
  cout<<"Please enter output file name   :"; cin>>outputfilename;cout<<endl;
  writefile.open(outputfilename,ios::out);
  sum1=0.0;
  sum2=0.0;
  for(i=0;i<N_input;i++)
  {
     xk1=sum1;
     xk2=sum2;
     readfile>>input;
     normlize(&input,N_fix,N_ex,0);
     multiply(&temp1,xk1,a11,N_fix,N_ex);
     multiply(&temp2,xk2,a12,N_fix,N_ex);
     multiply(&temp3,xk1,a21,N_fix,N_ex);
     multiply(&temp4,xk2,a22,N_fix,N_ex);
```

```
    multiply(&temp5,xk1,c1,N_fix,N_ex);
    multiply(&temp6,xk2,c2,N_fix,N_ex);
    multiply(&temp7,input,d,N_fix,N_ex);
   /*  sum1=xk1*a11+xk2*a12+input;
    sum2=xk1*a21+xk2*a22+input;
    yk=c1*xk1+c2*xk2+input*d;  */

    sum1=temp1+temp2+input;
    sum2=temp3+temp4+input;
    yk=temp5+temp6+temp7;

    normlize(&yk,N_fix,N_ex,0);
    writefile<<i<<"  "<<yk<<endl;


/* sum can possibly be N_bit+2*pow(2,(N_fix-1))-1 bits mantissa, when it changes
  to state variable it become N_bit mantissa. */

    mantissa=frexp(sum1, &exponent);
    if(abs(exponent)>pow(2,(N_ex-1)))
       sum1=0.0;
    else
       normlize(&sum1,N_fix,N_ex,0);
  //   writefile<<i<<"  "<<sum1<<endl;
     mantissa=frexp(sum2, &exponent);
    if(abs(exponent)>pow(2,(N_ex-1)))
       sum2=0.0;
    else
       normlize(&sum2,N_fix,N_ex,0);
  }


  readfile.close();
  writefile.close();
 }
}
```

```
/*****************************************************************************/
/*                          Program : standard_direct.cc                     */
/* Using direct structure to compute the second-order IIR filter result .The input and coefficient are */
/* quantized to fixed bits of mantissa and exponent. The intermeadiate data are chosen to be double*/
/* precision. The result is assumed to be no round-off noise                 */
/*****************************************************************************/

#include<stdio.h>
#include<iostream.h>
#include<math.h>
#include<fstream.h>
#include<stdlib.h>
main()
{
  double alpha1,alpha2,beta1,beta2,d;
  double a1,a2,b1,b2,d1;
  char inputfilename[32],outputfilename[32];
  ifstream readfile;
  ofstream writefile;
  int i,N_input,N_fix,N_ex;
  double input,x0,x1,x2,y1,y2,yk;
  void normlize(double *,int,int,int);

  //enter the coefficient
  cout<<"Please enter the transfer function coefficient:"<<endl;
  cout<<"     alpha1="; cin>>a1; cout<<endl;
  cout<<"     alpha2="; cin>>a2; cout<<endl;
  cout<<"     beta1 ="; cin>>b1 ; cout<<endl;
  cout<<"     beta2 ="; cin>>b2 ; cout<<endl;
  cout<<"       d   ="; cin>>d1    ; cout<<endl;

for(int m=0; m<15; m++) {
  cout<<"Please enter fixed bits:";    cin>>N_fix; cout<<endl;
  cout<<"Please enter exponent bits:"; cin>>N_ex;   cout<<endl;
  alpha1=a1;
  alpha2=a2;
  beta1=b1;
  beta2=b2;
  d=d1;

 //normalize the coefficient to small floating-point representation
  normlize(&alpha1,N_fix,N_ex,1); cout<<"    "<<"alpha1="<<alpha1<<endl;
  normlize(&alpha2,N_fix,N_ex,1); cout<<"    "<<"alpha2="<<alpha2<<endl;
  normlize(&beta1,N_fix,N_ex,1);  cout<<"    "<<"beta1="<<beta1<<endl;
  normlize(&beta2,N_fix,N_ex,1);  cout<<"    "<<"beta2="<<beta2<<endl;
  normlize(&d,N_fix,N_ex,1);     cout<<"    "<<"d="<<d<<endl;

  cout<<"Please enter input data file name:"; cin>>inputfilename; cout<<endl;
  readfile.open(inputfilename,ios::in);
  readfile>>N_input;
```

```
 cout<<"Please enter output file name   :"; cin>>outputfilename;cout<<endl;
 writefile.open(outputfilename,ios::out);

 for(i=0;i<N_input;i++)
 {
   readfile>>input;
   if(i==0)
    {
      x1=0.0;
      x2=0.0;
      y1=0.0;
      y2=0.0;
    }
   x0=input;
   normlize(&x0,N_fix,N_ex,0);
   yk=x0*d+(d*beta1+alpha1)*x1+(d*beta2+alpha2)*x2-beta1*y1-beta2*y2;
   writefile<<i<<"          "<<yk<<endl;
   x2=x1;
   x1=x0;
   y2=y1;
   y1=yk;
 }
 readfile.close();
 writefile.close();

}
}

// the normalize function is the same as  that in float_state_space.cc
```

```
/*****************************************************************************/
/*                          Program : float_direct.cc                        */
/*  Using direct structure to compute the second-order IIR filter result The input and coefficient are */
/*quantized to fixed bits of mantissa and exponent. The intermeadiate data are also quantized to */
/*fixed bits of mantissa and expoent. This simulates the result in real FPGA design.The result */
/* has round-off noise.                                                      */
/*****************************************************************************/

#include<stdio.h>
#include<iostream.h>
#include<math.h>
#include<fstream.h>
#include<stdlib.h>
main()
{
  double alpha1,alpha2,beta1,beta2,d;
  double a1,a2,b1,b2,d1;
  char inputfilename[32],outputfilename[32];
  ifstream readfile;
  ofstream writefile;
  int i,N_input,N_fix,N_ex,exponent;
  double input,x0,x1,x2,y1,y2,yk;
  void normlize(double *,int,int,int);
  float multiply(double *,double,double,int,int);
  cout<<"Please enter the transfer function coefficient:"<<endl;
  cout<<"    alpha1="; cin>>a1; cout<<endl;
  cout<<"    alpha2="; cin>>a2; cout<<endl;
  cout<<"    beta1 ="; cin>>b1 ; cout<<endl;
  cout<<"    beta2 ="; cin>>b2 ; cout<<endl;
  cout<<"      d   ="; cin>>d1    ; cout<<endl;
 for(int m=0; m<20;m++){
  cout<<"Please enter fixed bits you want to use:";  cin>>N_fix;cout<<endl;
  cout<<"Please enter exponent bits you want to use:";cin>>N_ex; cout<<endl;
  alpha1=a1;
  alpha2=a2;
  beta1=b1;
  beta2=b2;
  d=d1;
  normlize(&alpha1,N_fix,N_ex,1); cout<<"    "<<"alpha1="<<alpha1<<endl;
  normlize(&alpha2,N_fix,N_ex,1); cout<<"    "<<"alpha2="<<alpha2<<endl;
  normlize(&beta1,N_fix,N_ex,1);  cout<<"    "<<"beta1="<<beta1<<endl;
  normlize(&beta2,N_fix,N_ex,1);  cout<<"    "<<"beta2="<<beta2<<endl;
  normlize(&d,N_fix,N_ex,1);    cout<<"    "<<"d="<<d<<endl;
  cout<<"Please enter input data file name:"; cin>>inputfilename; cout<<endl;
  readfile.open(inputfilename,ios::in);
  readfile>>N_input;
```

```
  cout<<"Please enter output file name   :"; cin>>outputfilename;cout<<endl;
  writefile.open(outputfilename,ios::out);
  for(i=0;i<N_input;i++)
  {
     readfile>>input;
     if(i==0)
      {
         x1=0.0;
         x2=0.0;
         y1=0.0;
         y2=0.0;
       }
     x0=input;
     normlize(&x0,N_fix,N_ex,0);
     normlize(&x1,N_fix,N_ex,0);
     normlize(&x2,N_fix,N_ex,0);
     normlize(&y1,N_fix,N_ex,0);
     normlize(&y2,N_fix,N_ex,0);
     yk=x0*d+(d*beta1+alpha1)*x1+(d*beta2+alpha2)*x2-beta1*y1-beta2*y2;
     normlize(&yk,N_fix,N_ex,0);
     writefile<<i<<"          "<<yk<<endl;
     x2=x1;
     x1=x0;
     y2=y1;
     y1=yk;
  }
  readfile.close();
  writefile.close();
 }

}

//The normalize function is the same as that in state_space.cc.
```

A2

IMPROVED BIT-PARALLEL FLOATING-POINT MULTIPLIER DESIGN

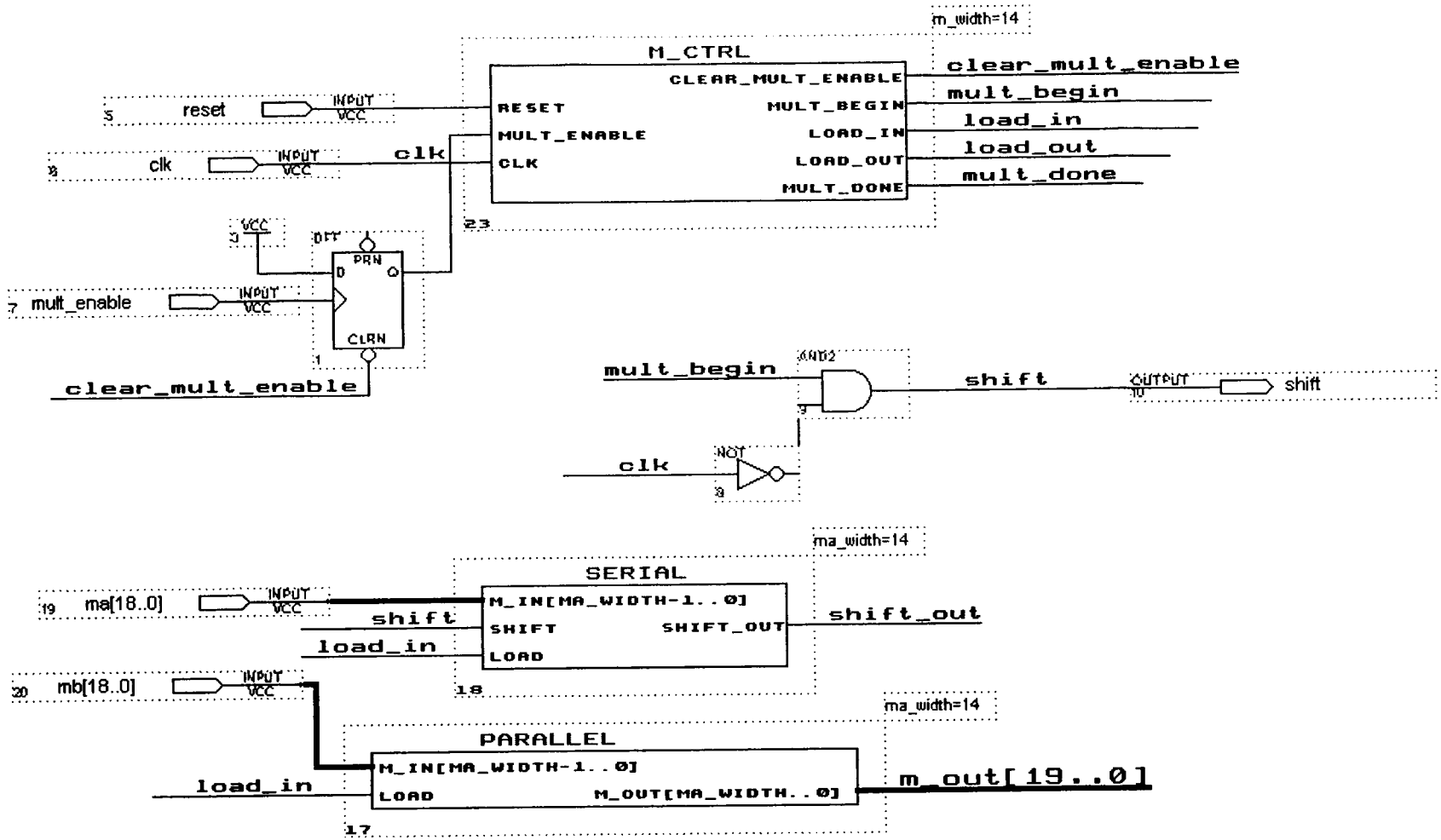
```
--------------------------------------------------------------------------------------------------------------
--                                          fp1_mult.tdf                                                    --
--The improved bit-parallel floating-point multiplier design based on Altera megafunciton fp_mult.          --
--------------------------------------------------------------------------------------------------------------


include "lpm_mult";
include "fp_normalizer";

parameters
(
        mantissa_width = 14,
        exponent_width = 5
);
constant exponent_offset = 2^(exponent_width-1);

subdesign fp1_mult
(
        sa:input;                                                       -- Sign of input a
        ma[mantissa_width..1]:input;            -- mantissa of input a
        ea[exponent_width..1]:input;            -- exponent of input a
        sb:input;                                                       -- sign of input b
        mb[mantissa_width..1]:input;            -- mantissa of input b
        eb[exponent_width..1]:input;            -- exponent of input b
        m_out[mantissa_width..1]:output;  -- mantissa of output
        e_out[exponent_width..1]:output;  -- exponent output
        s_out:output;                                                   -- sign output.
)

variable
        -- declare the multiplier
        mult:lpm_mult with
                (
                        lpm_widtha = mantissa_width,
                        lpm_widthb = mantissa_width,
                        lpm_widthp = 2 * mantissa_width,
                        lpm_widths = 2 * mantissa_width,
                        lpm_representation = "UNSIGNED"
                );
                e_temp[exponent_width..1]:node; -- temporary exponent -- the sum of the 2 exponents
    underflow           :node; --check the underflow;
begin
        -- multiply the mantissas
        mult.dataa[] = ma[];
        mult.datab[] = mb[];

        -- add the exponents and subtract 2^mantissa_width-1
        e_temp[] = ea[] + eb[] - exponent_offset;

  if mult.result[2*mantissa_width-1]==GND then
    e_temp[]=e_temp[]-1;
        end if;
  underflow=(not ea[exponent_width]) and (not eb[exponent_width]) and e_temp[exponent_width];
```

```
--if underflow happens, result set to 0
if underflow==VCC then
  m_out[]=GND;
  e_out[exponent_width]=VCC;
  e_out[exponent_width-1..1]=GND;
  s_out=VCC;
else
  e_out[]=e_temp[];
  if mult.result[2*mantissa_width-1]==GND then
    m_out[]=mult.result[2*mantissa_width-2..mantissa_width-1];
  else
    m_out[]=mult.result[2*mantissa_width-1..mantissa_width];
  end if;
  s_out = sa xnor sb;
end if;
end;
```

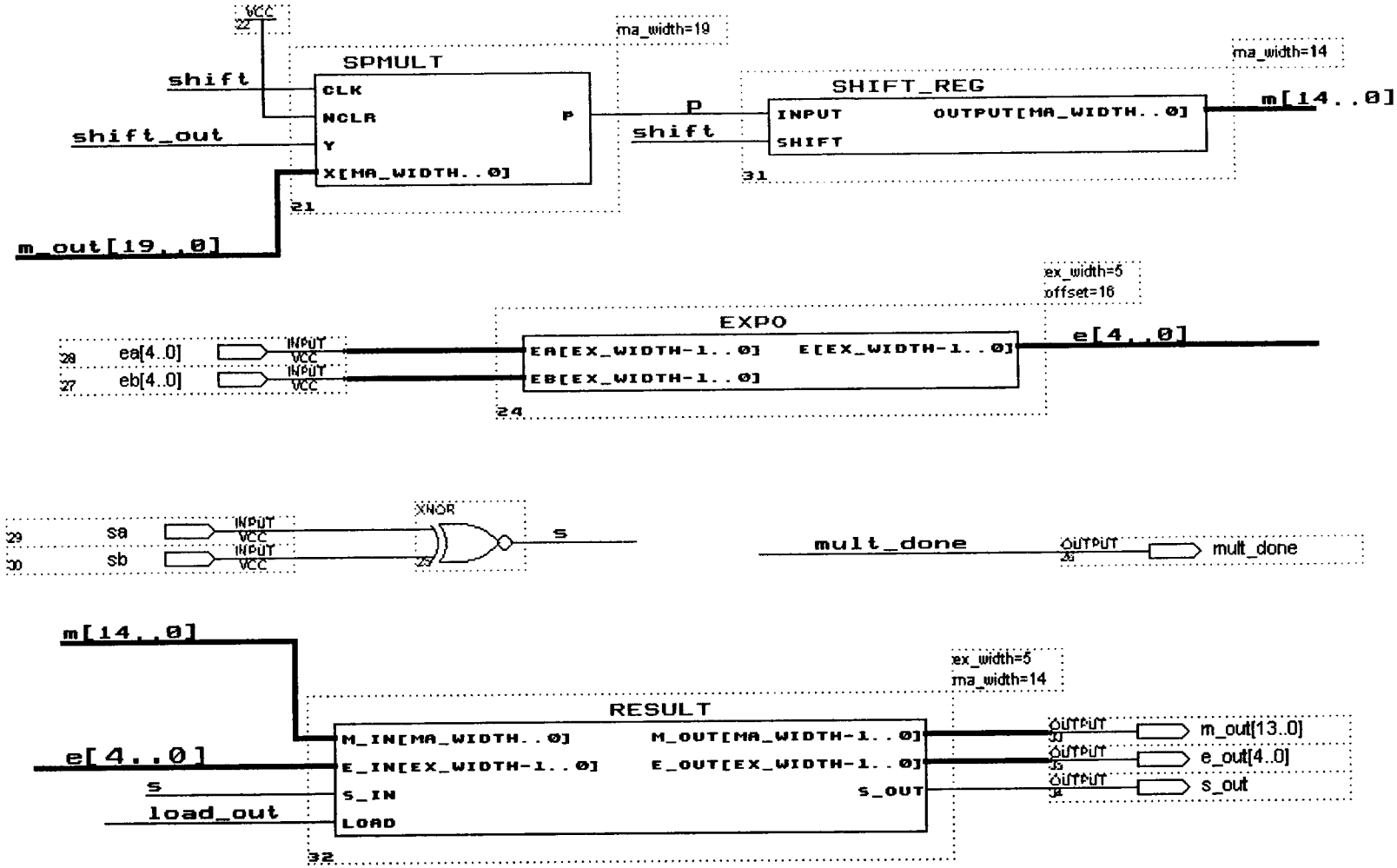
A3

BIT-SERIAL FLOATING-POINT MULTIPLIER DESIGN

Schematic Design of Bit-Serial Multiplier(Page 1)



Schematic Design of Bit-Serial Multiplier(Page 2)



```
--------------------------------------------------------------------------
--                          m_ctrl.vhd                                  --
--Control the bit-serial floating-point multiplier computation  --
--------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity m_ctrl is
 generic(m_width  : integer  :=15);
 port (  reset          : in std_logic;
        mult_enable     : in std_logic;
        clk             : in std_logic;
        clear_mult_enable: out std_logic;
        mult_begin      : out std_logic;
        load_in         : out std_logic;
        load_out        : out std_logic;
        mult_done       : out std_logic);
end m_ctrl;

architecture dataflow of m_ctrl is
signal state : std_logic_vector(2 downto 0);
begin
  process(clk,reset)
    variable cnt  : integer range 0 to (m_width+m_width-2);
  begin
    if reset='0' then
      clear_mult_enable<='0';
      mult_begin<='0';
      load_in<='0';
      load_out<='0';
      state<="000";
      cnt:=0;
    elsif clk'event and clk='1' then
      case state is
         when "000" => state<="001";
         when "001" =>
             clear_mult_enable<='1';
             state<="010";
         when "010" =>
             if mult_enable='1' then
                clear_mult_enable<='0';
                load_in<='1';
                mult_done<='0';
                state<="011";
             end if;
         when "011" =>
             load_in<='0';
             mult_begin<='1';
             state<="100";
         when "100" =>
             if cnt=m_width+m_width-1 then
                mult_begin<='0';
                load_out<='1';
```

```
            state<="101";
          else
            cnt:=cnt+1;
          end if;
        when "101" =>
           mult_done<='1';
           load_out<='0';
           state<="001";
        when others =>state<="000";
    end case;
  end if;
 end process;
end dataflow;
```

```
--------------------------------------------------------------------------
--                          serial.vhd                                  --
--A parallel input serial output register used in bit-serial floating--
--poit multiplier.                                                     --
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity serial is
  generic(ma_width  : integer :=14);
  port (
       m_in      : in std_logic_vector(ma_width-1 downto 0);
       shift     : in std_logic;
       load      : in std_logic;
       shift_out  : out std_logic);
end serial;

architecture dataflow of serial is
begin
  process(load, shift)
  variable temp : std_logic_vector(ma_width-1 downto 0);
  begin
   if load='1' then
     temp:=m_in;
   elsif shift'event and shift='1' then
     temp:='0' & temp(ma_width-1 downto 1);
   end if;
    shift_out<=temp(0);
  end  process;
end dataflow;
```

```
--------------------------------------------------------------------------------
--                                parallel.vhd                                --
--A parallel-in and parallel-out register used in bit-serial multiplier--
--design.
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity parallel is
  generic(ma_width  : integer :=14);
  port (
        m_in      : in std_logic_vector(ma_width-1 downto 0);
        load      : in std_logic;
        m_out     : out std_logic_vector(ma_width downto 0));
end parallel;

architecture dataflow of parallel is
begin
  process(load)
  begin
    if load='1' then
      m_out<='0' & m_in;
    end if;
  end process;
end dataflow;



--------------------------------------------------------------------------------
--                                sp_mult.vhd                                 --
--Serial-parallel two's complement multiplier optimized for FLEX 10K--
--architecture. Serial output product Uses Booth's algorithm for two's --
--complement multiplication.                                                 -
--------------------------------------------------------------------------------

  LIBRARY ieee;
  USE ieee.std_logic_1164.all;

  ENTITY sp_mult IS
    GENERIC(ma_width   : INTEGER :=14);
    PORT(clk, nclr, y : IN  STD_LOGIC;
           x    : IN  STD_LOGIC_VECTOR(ma_width DOWNTO 0);
           p    : OUT STD_LOGIC);
  END sp_mult;

  ARCHITECTURE boolean OF sp_mult IS
    SIGNAL cq, sq, xq : STD_LOGIC_VECTOR(x'HIGH DOWNTO 0);
    SIGNAL yq         : STD_LOGIC;
  BEGIN
    PROCESS(clk, nclr)
    BEGIN
      IF nclr = '0' THEN
```

```
      yq <= '0';
      sq <= (OTHERS => '0');
      cq <= (OTHERS => '0');
      xq <= (OTHERS => '0');
    ELSIF clk'EVENT AND clk = '1' THEN
      sq(x'HIGH) <= xq(x'HIGH) XOR cq(x'HIGH);
      cq(x'HIGH) <= (NOT yq) AND (xq(x'HIGH) XOR cq(x'HIGH));
      xq(x'HIGH) <= x(x'HIGH) AND ( y XOR yq );
      FOR i IN x'HIGH - 1 DOWNTO 0 LOOP
        sq(i) <= xq(i) XOR cq(i) XOR sq(i+1);
        cq(i) <= (yq XOR sq(i+1)) AND (xq(i) XOR cq(i));
        xq(i) <= x(i) AND ( y XOR yq );
      END LOOP;
      yq <=y;
    END IF;
  END PROCESS;
 p <= sq(0);
 END boolean;
```

```
--------------------------------------------------------------------------
--                    expo.vhd                                          --
--Exponent computation part in bit-serial multipiler design  --
--------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity expo is
 generic(ex_width : integer := 5;
       offset  : integer := 16);
 port( ea, eb  : in std_logic_vector(ex_width-1 downto 0);
     e      : out std_logic_vector(ex_width-1 downto 0));
end expo;

architecture dataflow of expo is
begin
    e<=ea+eb-offset;
end dataflow;
```

```
------------------------------------------------------------------------------
--                                Result.vhd                                --
--Post normolizer in bit-serial multiplier design. It check the           --
--MSB and second MSB to adjust the exponent and mantissa output--
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

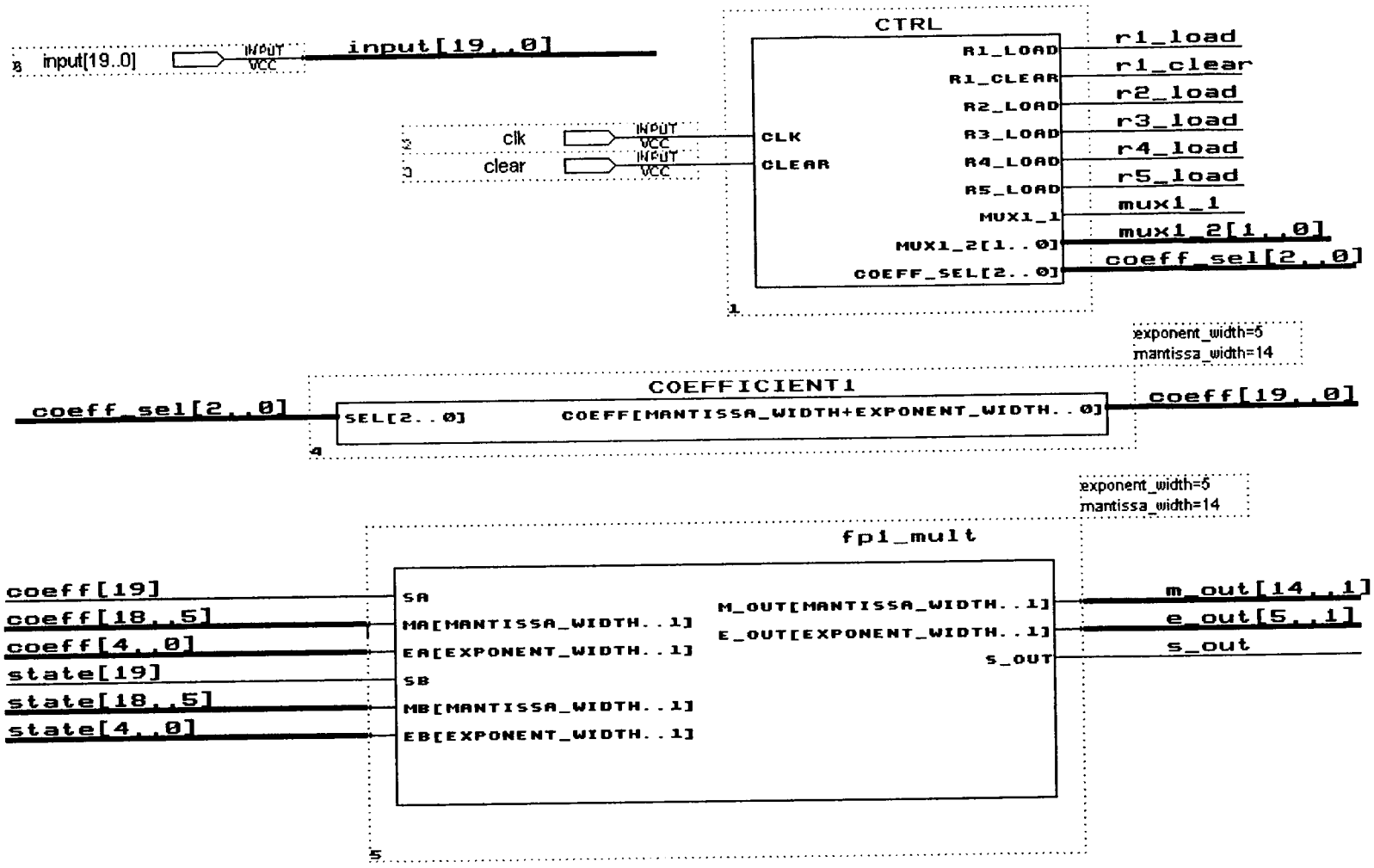
```
entity result is
 generic(ma_width : integer :=14;
       ex_width : integer :=5);
 port (
       m_in   : in std_logic_vector(ma_width downto 0);
       e_in   : in std_logic_vector(ex_width-1 downto 0);
       s_in   : in std_logic;
       load   : in std_logic;
       m_out  : out std_logic_vector(ma_width-1 downto 0);
       e_out  : out std_logic_vector(ex_width-1 downto 0);
       s_out  : out std_logic);
end result;

architecture dataflow of result is
begin
  process( load)
  begin
    if load='1' then
     if m_in(ma_width)='1' then
        e_out<=e_in;
        m_out<=m_in(ma_width downto 1);
      else
        e_out<=e_in-1;
        m_out<=m_in(ma_width-1 downto 0);
      end if;
      s_out<=s_in;
    end if;
   end process;
end dataflow;


--------------------------------------------------------------------
--                          shift_reg                             --
--A right shift register used in bit-serial multiplier design--
--to hold the output data                                        --
--------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity shift_reg  is
 generic(ma_width : integer := 14);
 port ( input   : in std_logic;
       shift   : in std_logic;
       output  : out std_logic_vector(ma_width downto 0));
end shift_reg;

architecture dataflow of shift_reg is
begin
  process(shift)
   variable temp : std_logic_vector(ma_width downto 0);
  begin
   if shift'event and shift='1' then
```

```
      temp:=input & temp(ma_width  downto 1);
    end if;
    output<=temp;
  end process;
end dataflow;
```

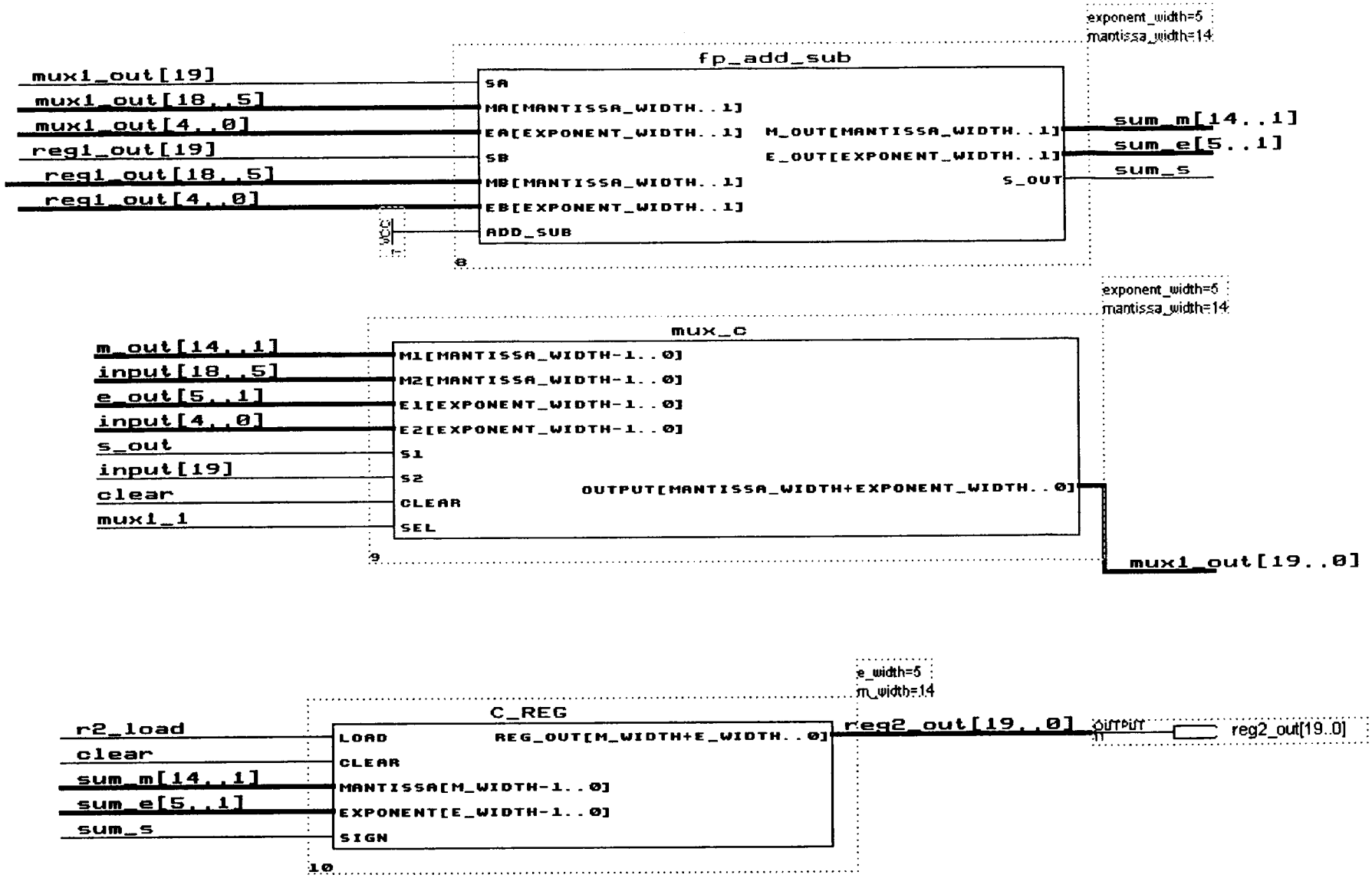
A4

UNPIPELINED BIT-PARALLEL FLOATING-POINT IIR FILTER DESGIN

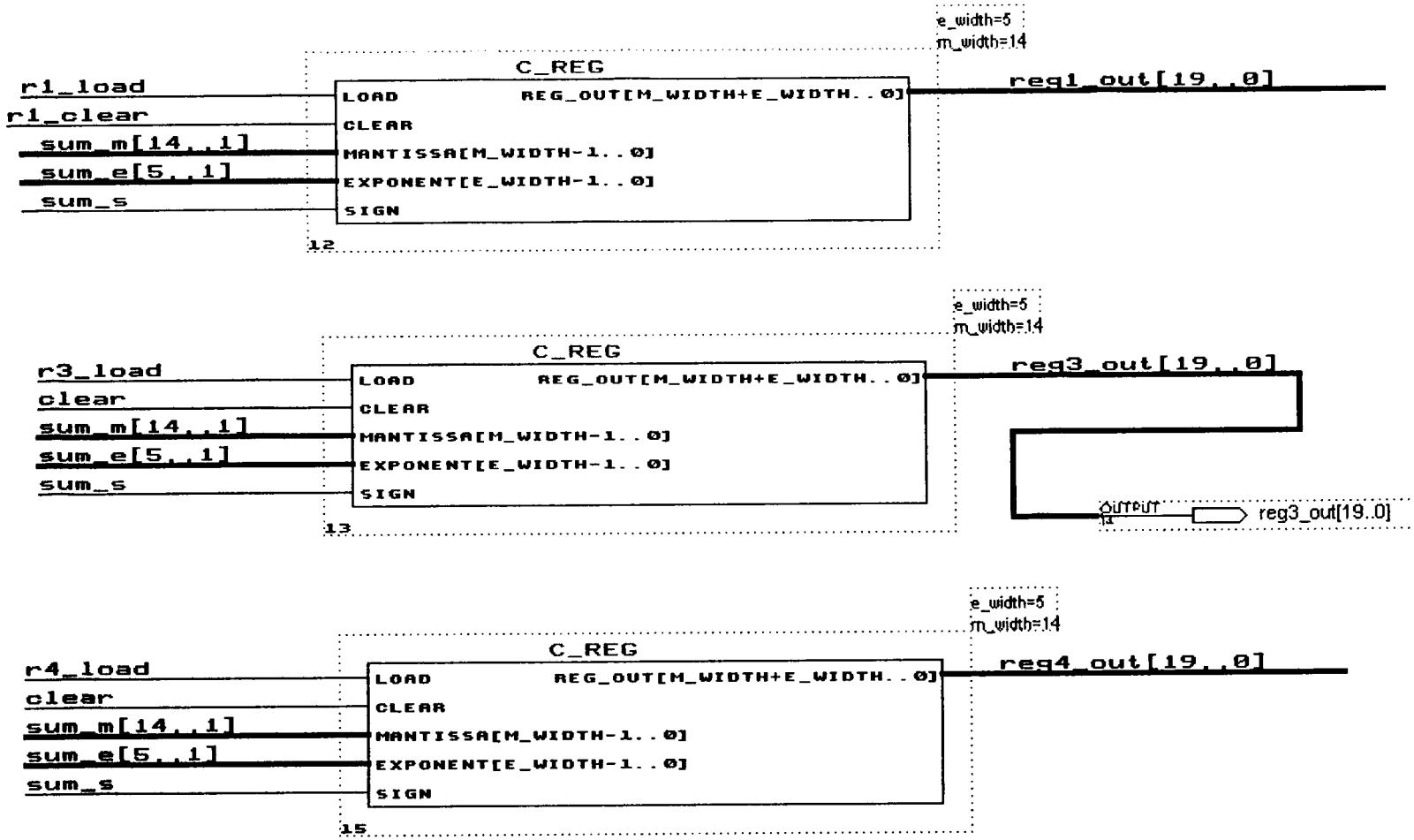
Unpipelined second-order Floating-point IIR filter (page 1)



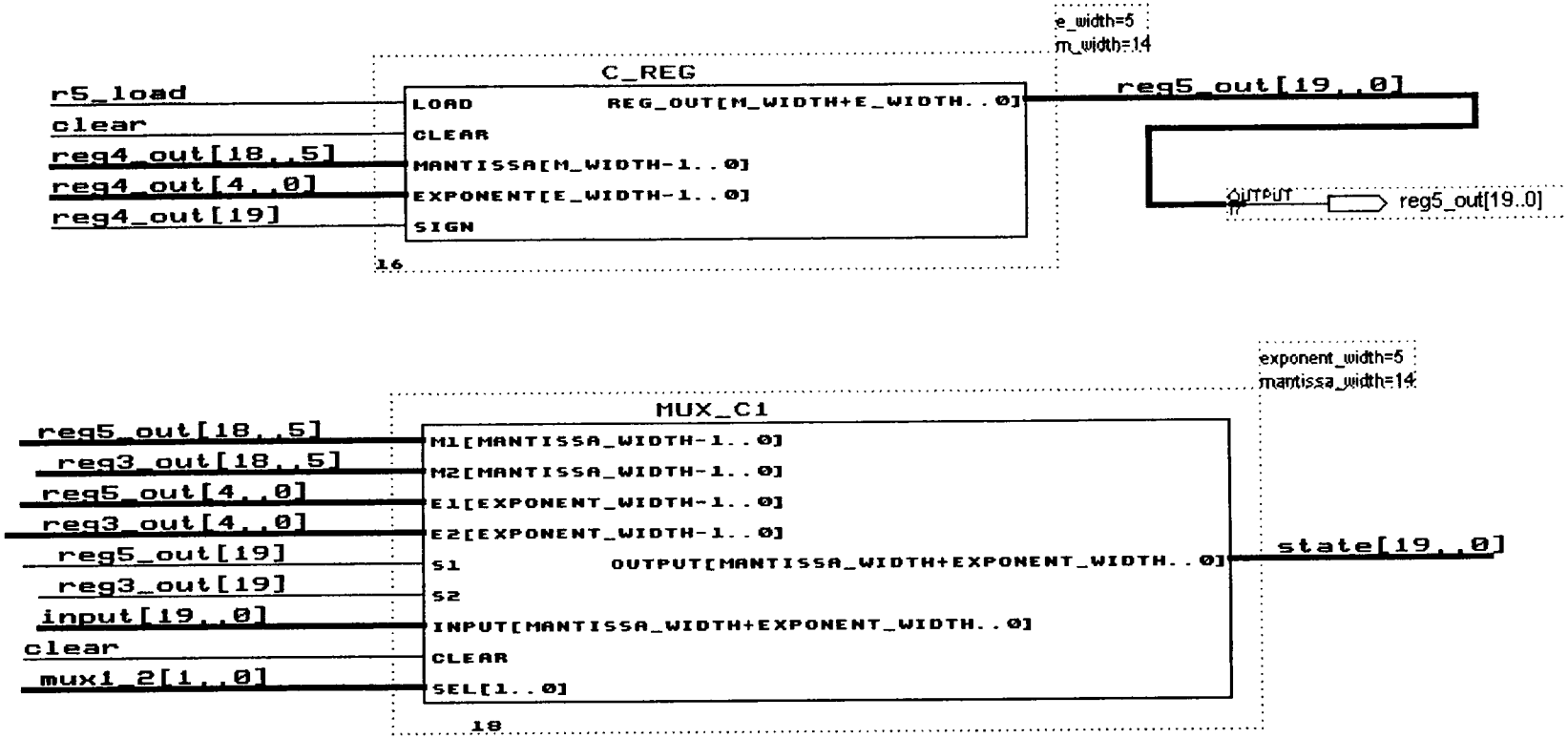
Unpipelined second-order Floating-point IIR filter (page 2)



Unpipelined second-order Floating-point IIR filter (page 3)



Unpipelined second-order Floating-point IIR filter (page 4)



```
--------------------------------------------------------------------------------
--                                ctrl.vhd                                    --
--Unpipelined second-order floating-point IIR  filter  control  part.  --
--It control the registers, multiplexes and coefficient. The total       --
--computation use 9 clock cycles                                         --
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ctrl is
 port ( clk        : in std_logic;
      clear       : in std_logic;
       r1_load,r1_clear    : out std_logic;
       r2_load           : out std_logic;
       r3_load           : out std_logic;
       r4_load           : out std_logic;
       r5_load           : out std_logic;
       mux1_1             : out std_logic;
       mux1_2             : out std_logic_vector(1 downto 0);
       coeff_sel          : out std_logic_vector(2 downto 0));

end ctrl;

architecture behavoiral of ctrl is
  signal state    : std_logic_vector(10 downto 0);
  --reset
  constant state0  : std_logic_vector(10 downto 0):="00000000000";
  --c1*x1[k]
  constant state1  : std_logic_vector(10 downto 0):="00100000001";
  --c1*x1[k]+c2*x2[k]
  constant state2  : std_logic_vector(10 downto 0):="10000001010";
  --input*d+c1*x1[k]+c2*x2[k]
  constant state3  : std_logic_vector(10 downto 0):="10000010011";
  --a11*x1[k]
  constant state4  : std_logic_vector(10 downto 0):="01000000100";
  --a11*x1[k]+a12*x2[k]
  constant state5  : std_logic_vector(10 downto 0):="10000001101";
  --input+a11*x1[k]+a12*x2[k]
  constant state6  : std_logic_vector(10 downto 0):="10000100110";
  --a12*x1[k]
  constant state7  : std_logic_vector(10 downto 0):="00010000110";
  --a22*x2[k]+a12*x1[k]
  constant state8  : std_logic_vector(10 downto 0):="10001001111";
  --input+a22*x2[k]+a21*x1[k];
  constant state9  : std_logic_vector(10 downto 0):="10000100000";

  begin
  process(clk,clear)
  begin
   if clear='0' then
      state<=state0;
   elsif clk'event and clk='1' then
      case state is
```

```
          when state0 => state<=state1;
          when state1 => state<=state2;
          when state2 => state<=state3;
          when state3 => state<=state4;
          when state4 => state<=state5;
          when state5 => state<=state6;
          when state6 => state<=state7;
          when state7 => state<=state8;
          when state8 => state<=state9;
          when state9 => state<=state1;
          when others => state<=state0;
        end case;
    end if;
   end process;
   with state select
     r1_load<=state(10) and clk   when   state2 |state5 |state8,
           state(10)          when   others;
     r1_clear<=state(10);
     r2_load<=state(9) ;
     r3_load<=state(8)  ;
     r4_load<=state(7)  ;
     r5_load<=state(6)  ;
     mux1_1<= state(5)  ;   mux1_2 <=state(4 downto 3) ;
     coeff_sel<=state(2 downto 0);
end behavoiral;



--------------------------------------------------------------------------------
--                    coefficient1.vhd                                        --
--This part store the coefficient of state-space structure c1,c2,d,a11  --
--a12,a21,a22. The control part control the coefficient address.        --
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity coefficient1 is
  generic(mantissa_width : integer :=14;
        exponent_width : integer :=5);
  port ( sel   : in  std_logic_vector(2 downto 0);
       coeff :  out std_logic_vector(mantissa_width+exponent_width downto 0));
end coefficient1;

architecture behavioral of coefficient1 is
begin
   process(sel)
   begin
     case sel is
        when "001" =>
            coeff <= "01011000111110101011" ; --c1
        when "010" =>
            coeff <= "01011000111110101011" ; --c2
        when "011" =>
            coeff <= "11111101001100010000" ; --d
```

```
        when "100" =>
            coeff <= "11111101001010010000" ; --a11
        when "101"  =>
            coeff <= "01011010111110101011" ; --a12
        when "110"  =>
            coeff <= "11010111000100001011" ; --a21
        when "111" =>
            coeff <= "11111101001010010000" ; --a22
        when others =>
            coeff <= "10000000000000000100000" ;
      end case;
  end process;
end behavioral;
```

```
--------------------------------------------------------------------------------
--                                mux_c.vhd                                   --
--A 2 to 1 multiplexe, which is used in all the filter design in the         --
--thesis. It represents multiplexer1 in the block diagram in chapterV        --
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity mux_c is
 generic(mantissa_width : integer :=14;
       exponent_width : integer :=5);
 port( m1,m2     : in std_logic_vector(mantissa_width-1 downto 0);
      e1,e2     : in std_logic_vector(exponent_width-1 downto 0);
      s1,s2     : in std_logic;
      clear     : in std_logic;
      sel       : in std_logic;
      output    : out std_logic_vector(mantissa_width+exponent_width
                          downto 0));
end mux_c;

architecture dataflow of mux_c is
begin
  process(sel,clear)
  begin
    if clear='0' then --reset to floating-point representation of 0.
      output(mantissa_width+exponent_width)<='1';
      output(mantissa_width+exponent_width-1 downto
                          exponent_width)<="00000000000000";
      output(exponent_width-1)<='1';
      output(exponent_width-2 downto 0)<="0000";
    else
      case sel is
        when '0' =>
          output(mantissa_width+exponent_width)<=s1;
          output(mantissa_width+exponent_width-1 downto
                                    exponent_width) <=m1;
          output(exponent_width-1 downto 0)  <=e1;
        when others =>
          output(mantissa_width+exponent_width)<=s2;
```

```
        output(mantissa_width+exponent_width-1 downto
                                exponent_width) <=m2;
        output(exponent_width-1 downto 0) <=e2;
      end case;
    end if;
  end process;
end dataflow;
```

```
--------------------------------------------------------------------------------------------------
--      fp_add_sub Parameterized Reference Design                                               --
--                                                                                              --
--      Copyright (c) 1995 by Altera Corporation.  All rights reserved.  This text              --
--      contains proprietary, confidential information of Altera Corporation, and               --
--      may be used, copied, and/or disclosed only pursuant to the terms of a                   --
--      valid software license agreement with Altera Corporation.  This copyright               --
--      notice must be retained as part of this text at all times.                              --
--------------------------------------------------------------------------------------------------


include "lpm_add_sub";
include "fp_2_input_normalizer";
include "fp_normalizer";

parameters
(
        mantissa_width = 8,
        exponent_width = 7
);

subdesign fp_add_sub
(
        sa:input;                                               -- Sign of input a
        ma[mantissa_width..1]:input;            -- mantissa of input a
        ea[exponent_width..1]:input;            -- exponent of input a
        sb:input;                                               -- sign of input b
        mb[mantissa_width..1]:input;            -- mantissa of input b
        eb[exponent_width..1]:input;            -- exponent of input b
        add_sub:input;                                          -- 1 == ADD, 0 = SUB
        m_out[mantissa_width..1]:output;  -- mantissa of output
        e_out[exponent_width..1]:output;  -- exponent output
        s_out:output;                                           -- sign output.
)
variable
        op:node; -- tells what operation to send to the actual adder.  See below for truth table
        norm_2:fp_2_input_normalizer with
                (
                        mantissa_width = mantissa_width,
                        exponent_width = exponent_width
                );
        e_norm[exponent_width..1]:node;         -- normalized exponent
        ma_norm[mantissa_width..1]:node;-- normalized a mantissa
        mb_norm[mantissa_width..1]:node;-- normalized b mantissa
```

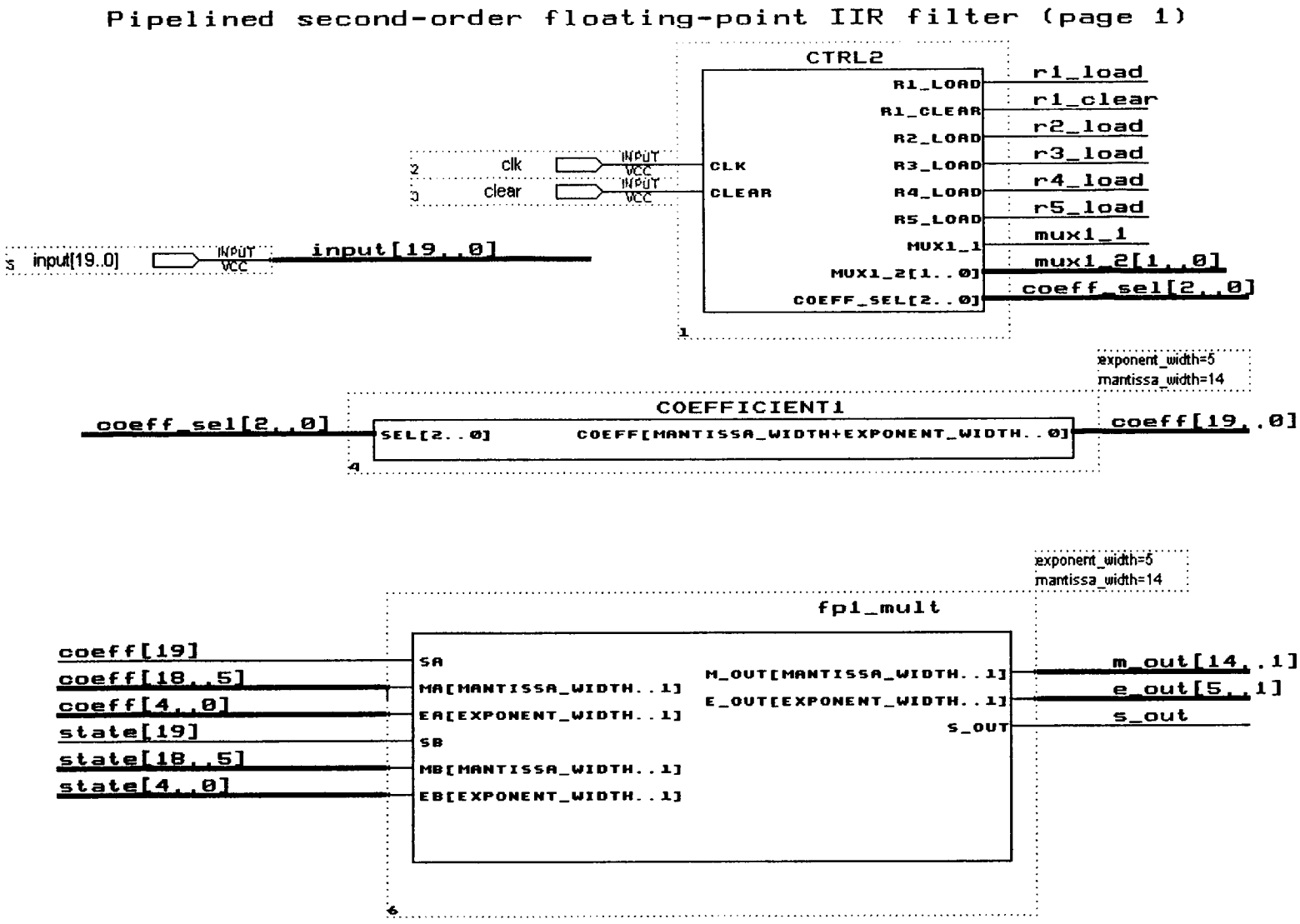
```
        mantissa_adder:lpm_add_sub with
                (
                        lpm_representation = "UNSIGNED",
                        lpm_width = mantissa_width+1
                );
        e_out_pn[exponent_width..1]:node;-- output exponent before final normalization
        m_out_pn[mantissa_width+1..1]:node;     -- output mantissa before final normalization
        output_normalizer:fp_normalizer with
                (
                        mantissa_width = mantissa_width+1,
                        exponent_width = exponent_width
                );
begin
        --
        -- normalize the inputs so they have the same exponent
        --
        norm_2.ma[] = ma[];
        norm_2.ea[] = ea[];
        norm_2.mb[] = mb[];
        norm_2.eb[] = eb[];
        e_norm[] = norm_2.e_out[];
        ma_norm[]= norm_2.ma_out[];
        mb_norm[]= norm_2.mb_out[];

        --
        -- add or subtract the mantissas
        mantissa_adder.dataa[] = (GND, ma_norm[]);
        mantissa_adder.datab[] = (GND, mb_norm[]);
        -- sign table:
        --      sa sb add_sub |                 | op | negate result
        --       0 0   0   | (-a-(-b)) = -(a-b) | - | yes
        --       0 0   1   | (-a+(-b)) = -(a+b) | + | yes
        --       0 1   0   | (-a-(+b)) = -(a+b) | + | yes
        --       0 1   1   | (-a+(+b)) = -(a-b) | - | yes
        --       1 0   0   | (+a-(-b)) = +(a+b) | + | no
        --       1 0   1   | (+a+(-b)) = +(a-b) | - | no
        --       1 1   0   | (+a-(+b)) = +(a-b) | - | no
        --       1 1   1   | (+a+(+b)) = +(a+b) | + | no
        -- This can be minimized:
        --  op = !(sb xor add_sub) xor !sa
        --  negate = !sa
        op = !(sb xor add_sub) xor !sa;
        mantissa_adder.add_sub = op;

        if (op == 0) and (mantissa_adder.result[mantissa_width]) then
                -- if there was a subtraction,  and the result was negative, invert it
                m_out_pn[] = 0 - mantissa_adder.result[];
                s_out = !sa;
        else
                m_out_pn[] = mantissa_adder.result[];
                s_out = sa;
        end if;
        e_out_pn[] = e_norm[];
        output_normalizer.mantissa[] = m_out_pn[];
```

```
        output_normalizer.exponent[] = e_out_pn[];
        m_out[] = output_normalizer.mantissa_norm[mantissa_width+1..2];
        e_out[] = output_normalizer.exponent_norm[]+1;


end;




--------------------------------------------------------------------------------------------
--                                    c_reg.vhd                                           --
--a register with clear and load control. It is used in all the filter design            --
--in chapter V. The parameter m_width, e_width can be changed for different bit--
--of small floating-point representation                                                 --
--------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity c_reg is
 generic(
       m_width   : integer :=15;
       e_width   : integer :=6);
  port(load      : in  std_logic;
      clear     : in  std_logic;
      mantissa  : in  std_logic_vector(m_width-1 downto 0);
      exponent  : in  std_logic_vector(e_width-1 downto 0);
      sign      : in  std_logic;
      reg_out   : out  std_logic_vector(m_width+e_width downto 0));
end c_reg;

architecture dataflow of c_reg is
begin
  process(load,clear)
  begin
   if clear='0' then
      reg_out(m_width+e_width)<='1';
      reg_out(m_width+e_width-1 downto e_width)<="00000000000000";
      reg_out(e_width-1)<='1';
      reg_out(e_width-2 downto 0)<="0000";
   elsif (load'event and load='1') then
      reg_out(m_width+e_width)<=sign;
      reg_out(m_width+e_width-1 downto e_width)<=mantissa(m_width-1 downto 0);
      reg_out(e_width-1 downto 0)<=exponent(e_width-1 downto 0);
   end if;
  end process;
end dataflow;




--------------------------------------------------------------------------------------------
--                                    mux_c1.vhd                                          --
--A 1 from 3 multiplexer which is used in all the filter design in chapter V. It         --
--represents multiplexer2 in the filter block diagram in chapter V. The parameter      --
```

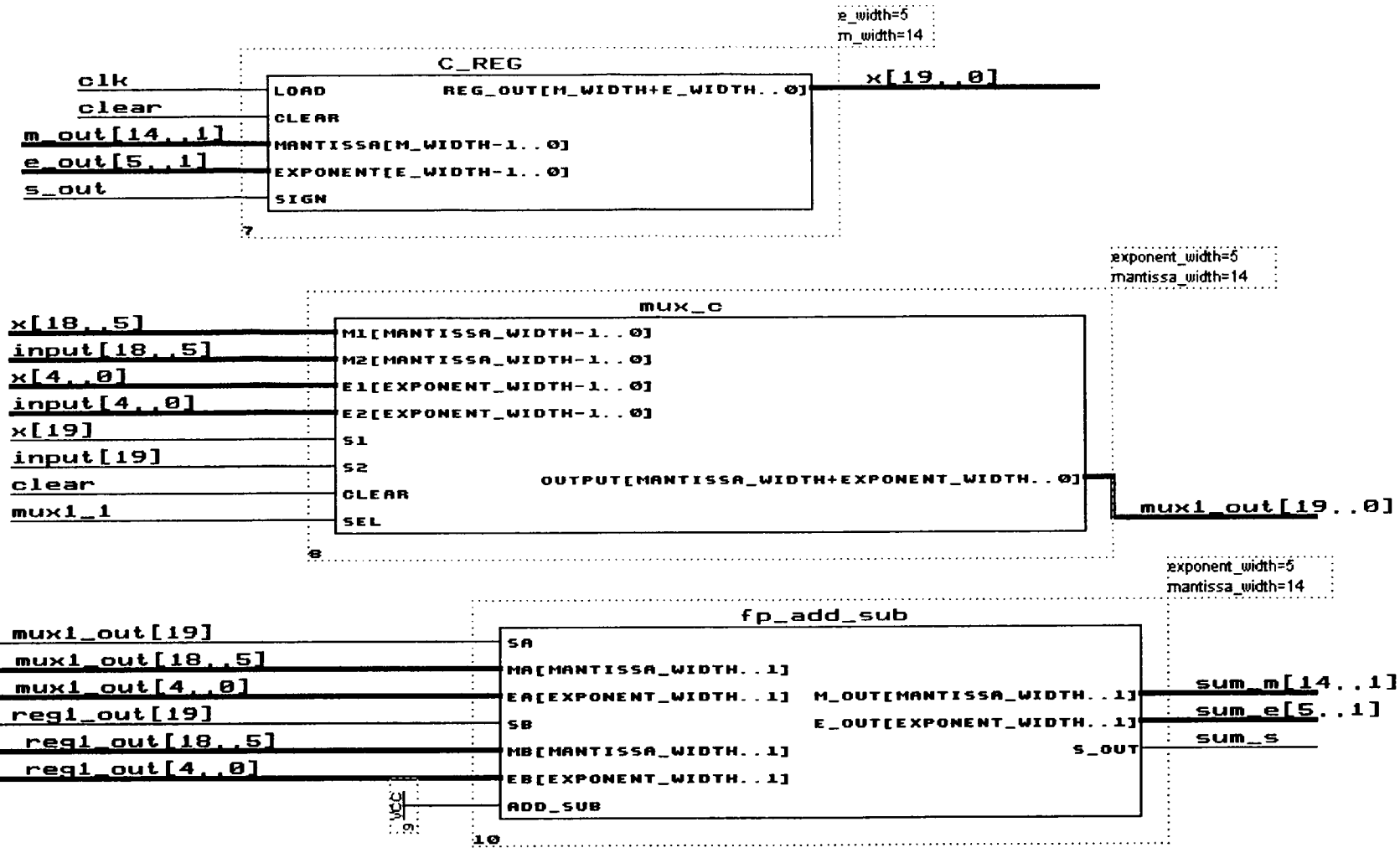
```
--mantissa_width and expoent_width can be changed to implement different small   --
--floating-point representation of numbers                                       --
-----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity mux_c1 is
 generic(mantissa_width : integer :=14;
       exponent_width : integer :=5);
 port( m1,m2  : in std_logic_vector(mantissa_width-1 downto 0);
      e1,e2  : in std_logic_vector(exponent_width-1 downto 0);
      s1,s2  : in std_logic;
      input  : in std_logic_vector(mantissa_width+exponent_width downto 0);
      clear    : in std_logic;
      sel      : in std_logic_vector(1 downto 0);
      output   : out std_logic_vector(mantissa_width+exponent_width downto 0));
end mux_c1;

architecture dataflow of mux_c1 is
begin
  process(sel,clear)
  begin
    if clear='0' then
      --output<=(others=>'0');
     output(mantissa_width+exponent_width)<='1';
     output(mantissa_width+exponent_width-1 downto exponent_width)<="00000000000000";
     output(exponent_width-1)<='1';
     output(exponent_width-2 downto 0)<="0000";
    else
      case sel is
        when "00" =>
          output(mantissa_width+exponent_width)<=s1;
          output(mantissa_width+exponent_width-1 downto exponent_width) <=m1;
          output(exponent_width-1 downto 0)  <=e1;
        when "01" =>
          output(mantissa_width+exponent_width)<=s2;
          output(mantissa_width+exponent_width-1 downto exponent_width) <=m2;
          output(exponent_width-1 downto 0) <=e2;
        when others=>
          output<=input;
      end case;
    end if;
  end process;
end dataflow;
```

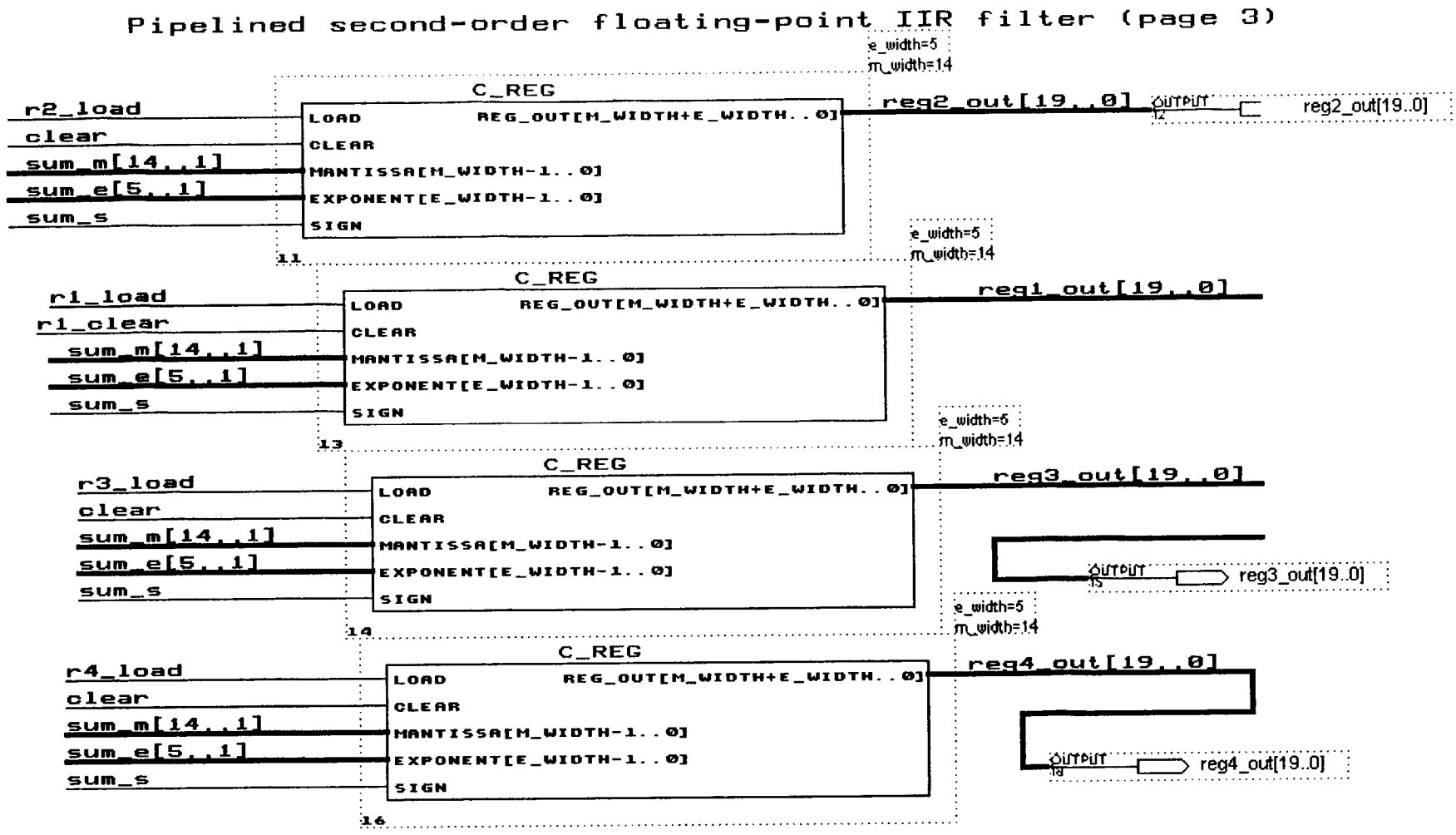
A5

PIPELINED BIT-PARALLEL FLOATING-POINT IIR FILTER DESIGN

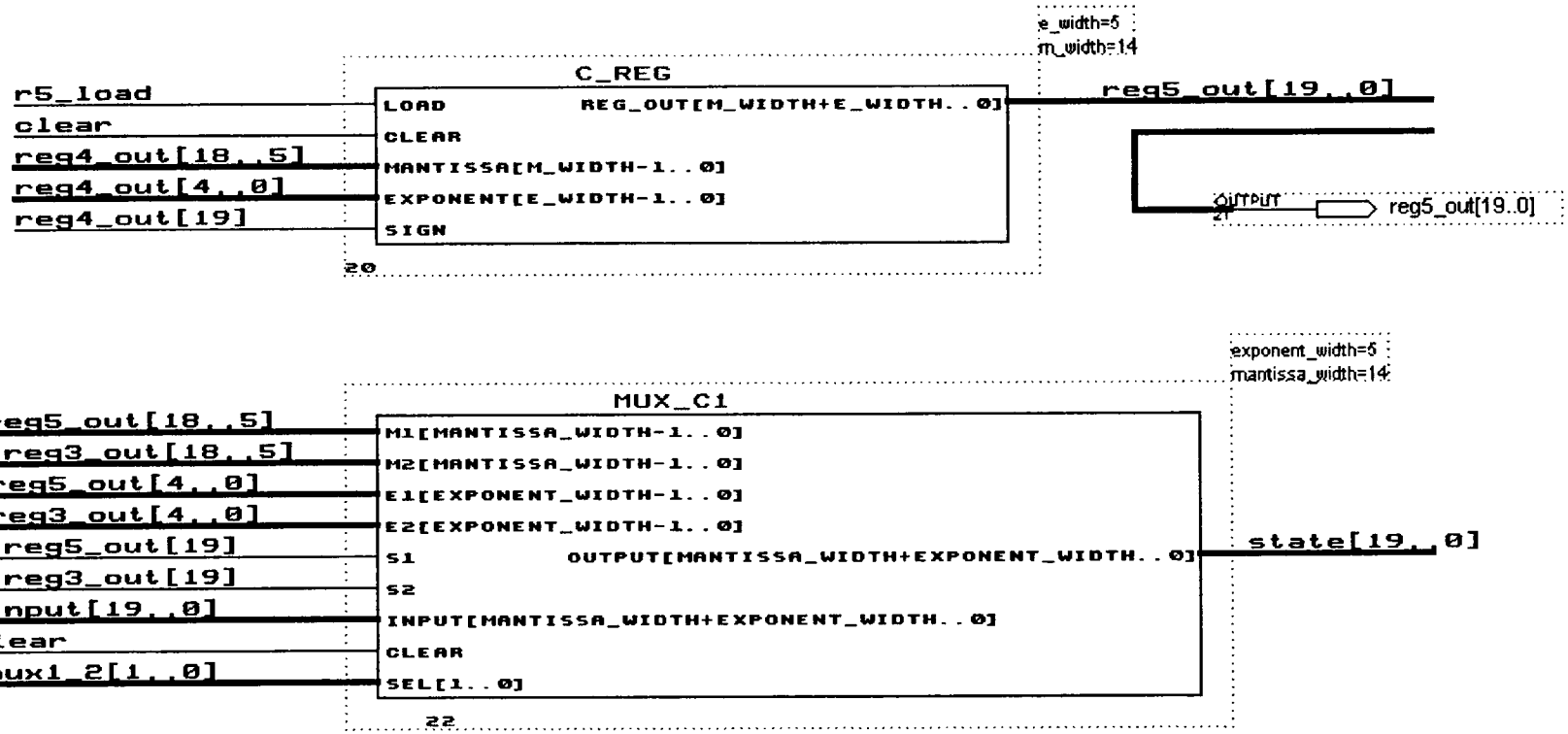

Pipelined second-order floating-point IIR filter (page 1)
CTRL2
CLK
CLEAR
R1_LOAD
R1_CLEAR
R2_LOAD
R3_LOAD
R4_LOAD
R5_LOAD
MUX1_1
MUX1_2[1..0]
COEFF_SEL[2..0]
r1_load
r1_clear
r2_load
r3_load
r4_load
r5_load
mux1_1
mux1_2[1..0]
coeff_sel[2..0]
clk
clear
input[19..0]
input[19..0]
INPUT
VCC
exponent_width=5
mantissa_width=14
COEFFICIENT1
SEL[2..0]
COEFF[MANTISSA_WIDTH+EXPONENT_WIDTH..0]
coeff_sel[2..0]
coeff[19..0]
fp1_mult
SA
MA[MANTISSA_WIDTH..1]
EA[EXPONENT_WIDTH..1]
SB
MB[MANTISSA_WIDTH..1]
EB[EXPONENT_WIDTH..1]
M_OUT[MANTISSA_WIDTH..1]
E_OUT[EXPONENT_WIDTH..1]
S_OUT
coeff[19]
coeff[18..5]
coeff[4..0]
state[19]
state[18..5]
state[4..0]
m_out[14..1]
e_out[5..1]
s_out

Pipelined second-order floating-point IIR filter (page 2)




Pipelined second-order floating-point IIR filter (page 3)
e_width=5
m_width=14
C_REG
r2_load
clear
sum_m[14..1]
sum_e[5..1]
sum_s
LOAD
CLEAR
MANTISSA[M_WIDTH-1..0]
EXPONENT[E_WIDTH-1..0]
SIGN
REG_OUT[M_WIDTH+E_WIDTH..0]
reg2_out[19..0]
OUTPUT
reg2_out[19..0]
11
e_width=5
m_width=14
C_REG
r1_load
r1_clear
sum_m[14..1]
sum_e[5..1]
sum_s
reg1_out[19..0]
13
e_width=5
m_width=14
C_REG
r3_load
clear
sum_m[14..1]
sum_e[5..1]
sum_s
reg3_out[19..0]
OUTPUT
reg3_out[19..0]
14
e_width=5
m_width=14
C_REG
r4_load
clear
sum_m[14..1]
sum_e[5..1]
sum_s
reg4_out[19..0]
OUTPUT
reg4_out[19..0]
16

Pipelined second-order floating-point IIR filter (page 4)



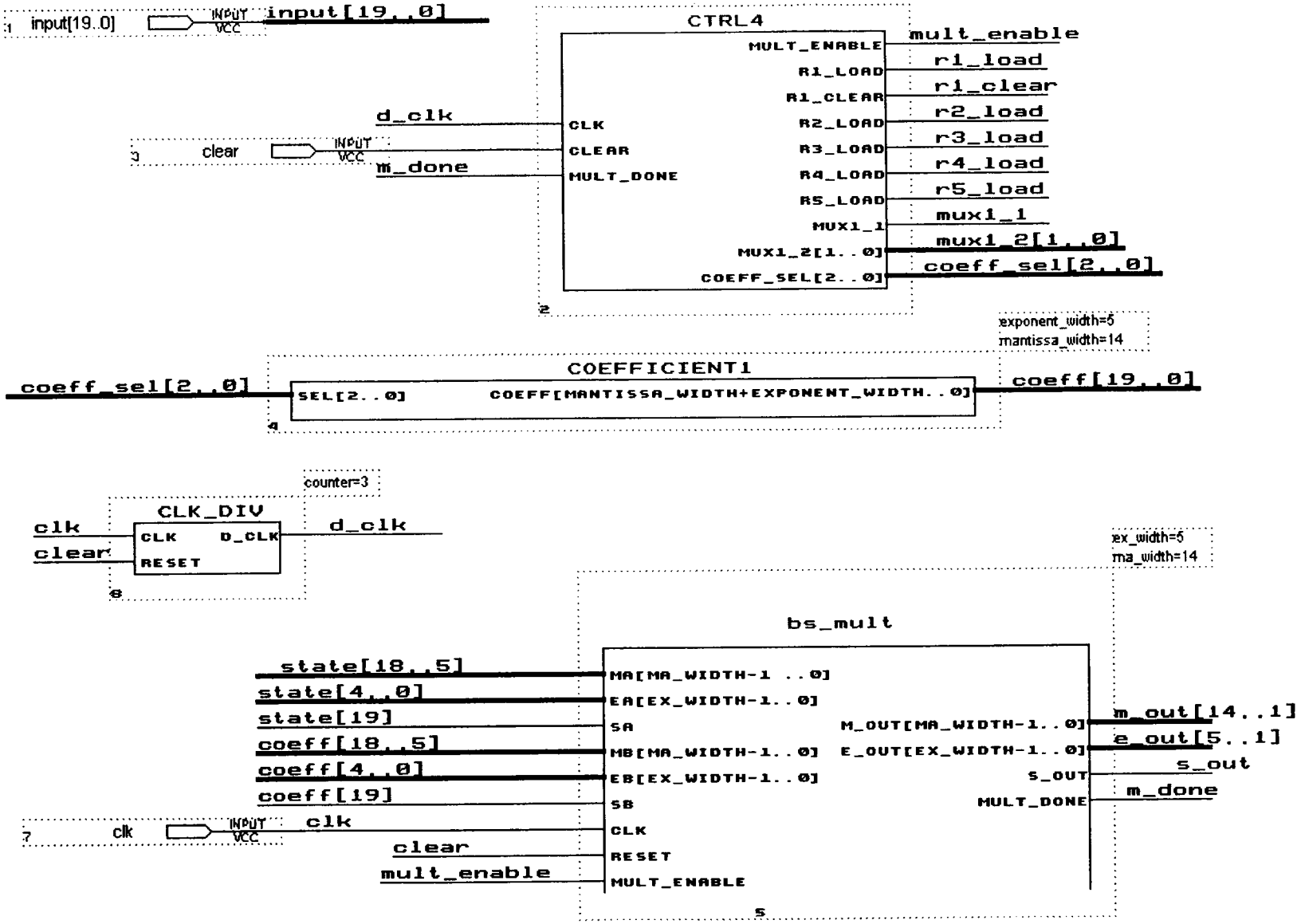
```
--------------------------------------------------------------------------------
--                        ctrl2.vhd                                           --
--Pipelined second-order floating-point IIR filter control part.  --
--It control the registers, multiplexes and coefficient. The total    --
--computation use 10 clock cycles                                    --
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ctrl2 is
 port ( clk        : in std_logic;
      clear       : in std_logic;
       r1_load,r1_clear    : out std_logic;
       r2_load             : out std_logic;
       r3_load             : out std_logic;
       r4_load             : out std_logic;
       r5_load             : out std_logic;
       mux1_1               : out std_logic;
       mux1_2               : out std_logic_vector(1 downto 0);
       coeff_sel           : out std_logic_vector(2 downto 0));

end ctrl2;

architecture behavoiral of ctrl2 is
  signal state    : std_logic_vector(10 downto 0);
  constant state0  : std_logic_vector(10 downto 0):="00000000000";
  --c1*x1[k]
  constant state1  : std_logic_vector(10 downto 0):="00100000001";
  --R6 load c1*x1[k], c2*x2[k] begin
  constant state2  : std_logic_vector(10 downto 0):="00000001010";
  --R6 load c2*x2[k], d*input begin
  constant state3  : std_logic_vector(10 downto 0):="10000010011";
  --R6 load d*input,
  constant state4  : std_logic_vector(10 downto 0):="10000000100";
  --a11*x1[k] begin, R2 load output,input is added
  constant state5  : std_logic_vector(10 downto 0):="01000100100";
  --R6 load a11*x1[k], a12*x2[k] begin
  constant state6  : std_logic_vector(10 downto 0):="10000001101";
  --R6 load a12*x2[k]
  constant state7  : std_logic_vector(10 downto 0):="10000000110";
  --a21*x1[k] begin, input added
  constant state8  : std_logic_vector(10 downto 0):="00010100110";
  --R6 load a21*x1[k], a22*x2[k] begin;
  constant state9  : std_logic_vector(10 downto 0):="10000001111";
  --R6 load a22*x2[k]
  constant state10 : std_logic_vector(10 downto 0):="10001000001";
  begin
  process(clk,clear)
  begin
   if clear='0' then
      state<=state0;
   elsif clk'event and clk='1' then
      case state is
         when state0 => state<=state1;
```

```
          when state1 => state<=state2;
          when state2 => state<=state3;
          when state3 => state<=state4;
          when state4 => state<=state5;
          when state5 => state<=state6;
          when state6 => state<=state7;
          when state7 => state<=state8;
          when state8 => state<=state9;
          when state9 => state<=state10;
          when state10 => state<=state1;
          when others => state<=state0;
        end case;
    end if;
   end process;
   with state select
     r1_load<=state(10) and clk   when   state3 |state6 |state9,
           state(10)          when   others;
     r1_clear<=state(10);
     r2_load<=state(9) ;
     r3_load<=state(8)  ;
     r4_load<=state(7)  ;
     r5_load<=state(6)  ;
     mux1_1<= state(5)  ;  mux1_2 <=state(4 downto 3) ;
     coeff_sel<=state(2 downto 0);
end behavoiral;
```

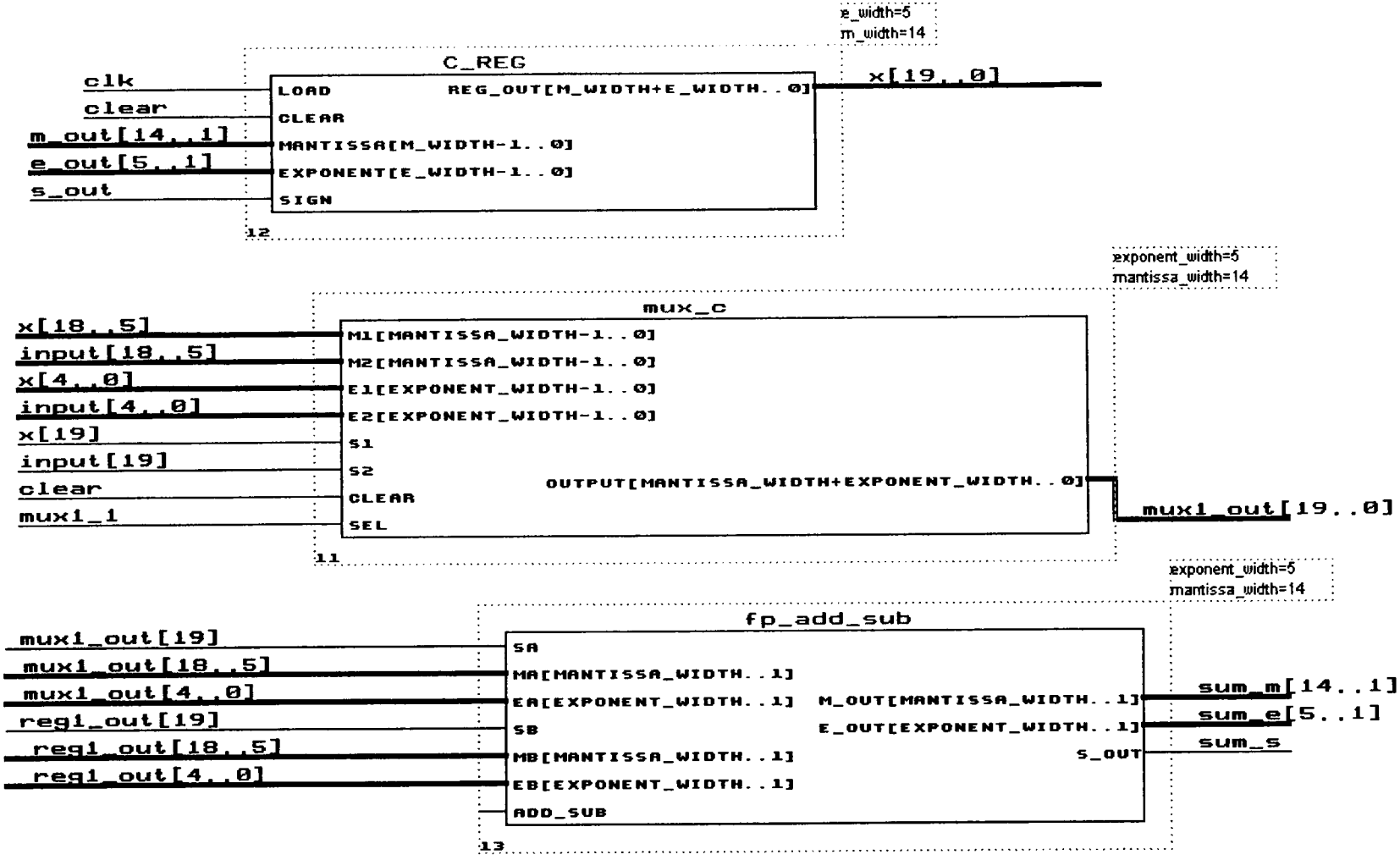
A6

BIT-SERIAL FLOATING-POINT IIR FILTER DESIGN

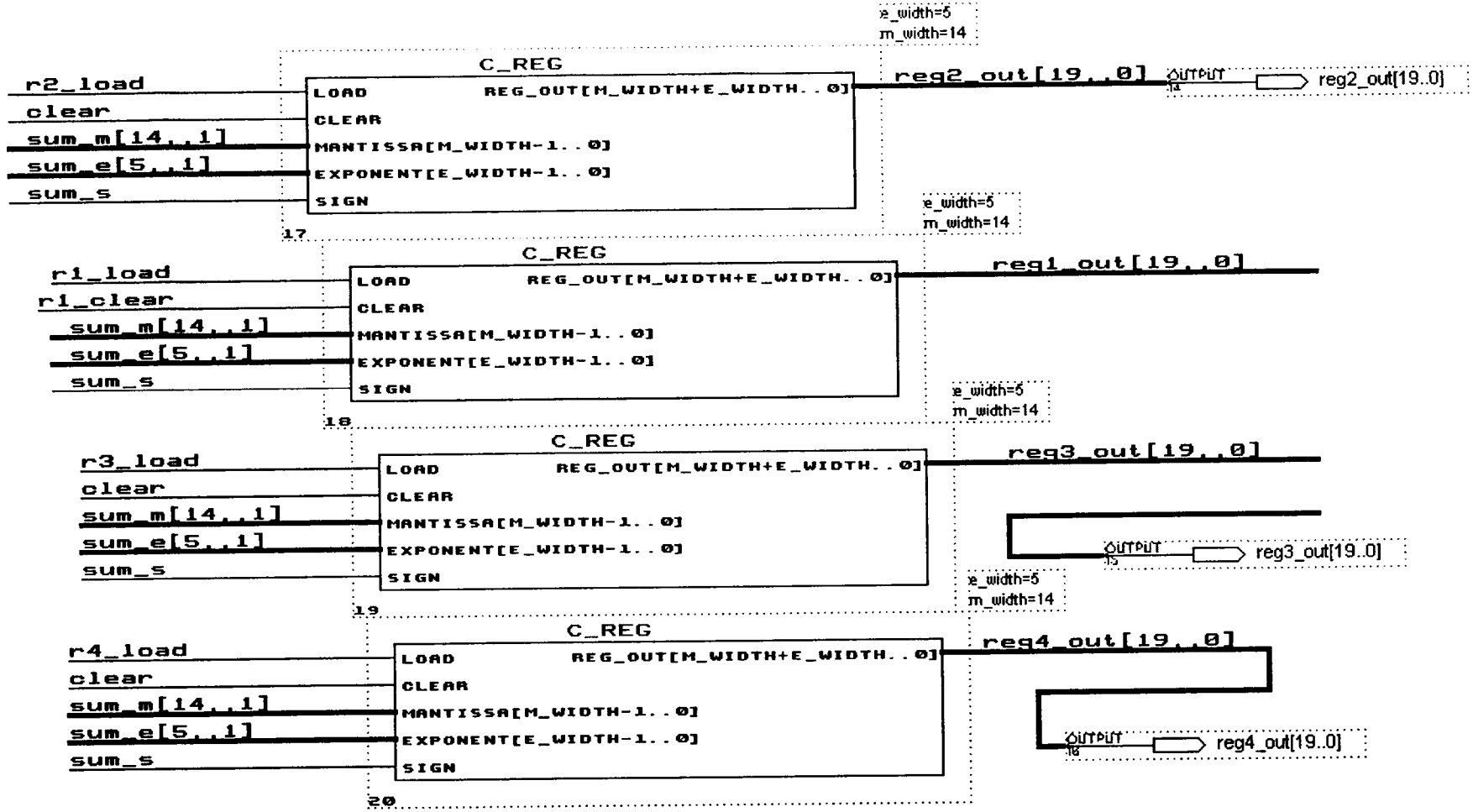
Second-order IIR filter design using bit-serial Multiplier(page 1)



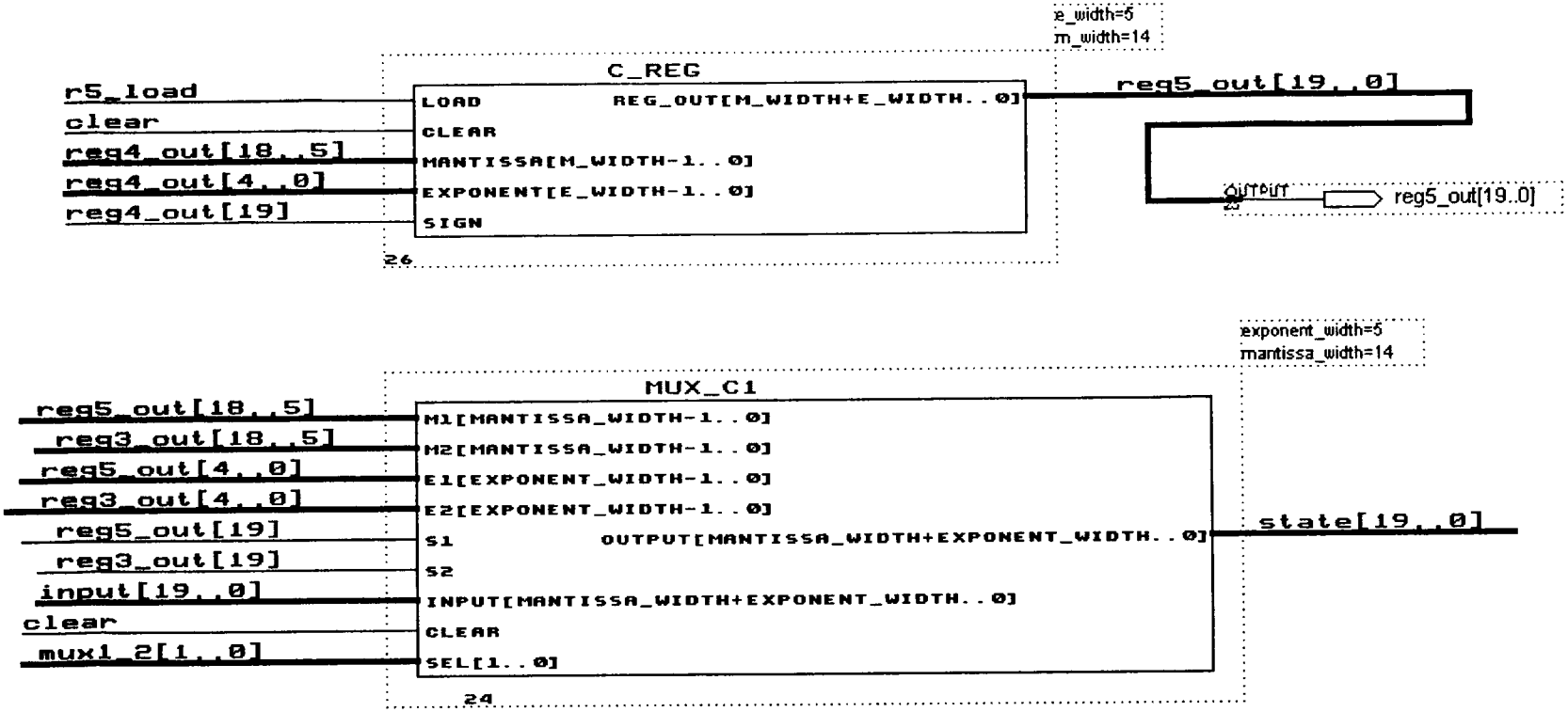
Second-order IIR filter design using bit-serial Multiplier(page 2)



Second-order IIR filter design using bit-serial Multiplier(page 3)



Second-order IIR filter design using bit-serial Multiplier(page 4)



```
--------------------------------------------------------------------------------
--                                ctrl4.vhd                                   --
--Pipelined bit-serial floating-point IIR  filter  control  part.             --
--It control the registers, multiplexes and coefficient. The total            --
--computation use 10 clock cycles                                             --
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ctrl4 is
 port ( clk              : in std_logic;
        clear            : in std_logic;
        mult_done        : in std_logic;
        mult_enable      : out std_logic;
        r1_load,r1_clear : out std_logic;
        r2_load          : out std_logic;
        r3_load          : out std_logic;
        r4_load          : out std_logic;
        r5_load          : out std_logic;
        mux1_1           : out std_logic;
        mux1_2           : out std_logic_vector(1 downto 0);
        coeff_sel        : out std_logic_vector(2 downto 0));

end ctrl4;

architecture behavoiral of ctrl4 is
  signal state    : std_logic_vector(10 downto 0);
  constant state0  : std_logic_vector(10 downto 0):="00000000000";
  --c1*x1[k]
  constant state1  : std_logic_vector(10 downto 0):="00100000001";
  --R6 load c1*x1[k], c2*x2[k] begin
  constant state2  : std_logic_vector(10 downto 0):="00000001010";
  --R6 load c2*x2[k], d*input begin
   constant state3  : std_logic_vector(10 downto 0):="10000010011";
  --R6 load d*input,
  constant state4  : std_logic_vector(10 downto 0):="10000000100";
  --a11*x1[k] begin, R2 load output,input is added
  constant state5  : std_logic_vector(10 downto 0):="01000100100";
  --R6 load a11*x1[k], a12*x2[k] begin
  constant state6  : std_logic_vector(10 downto 0):="10000001101";
  --R6 load a12*x2[k]
  constant state7  : std_logic_vector(10 downto 0):="10000000110";
  --a21*x1[k] begin, input added
  constant state8  : std_logic_vector(10 downto 0):="00010100110";
  --R6 load a21*x1[k], a22*x2[k] begin;
  constant state9  : std_logic_vector(10 downto 0):="10000001111";
  --R6 load a22*x2[k]
 constant state10 : std_logic_vector(10 downto 0):="10001000001";
 begin
  process(clk,clear)
  begin
   if clear='0' then
      state<=state0;
   elsif clk'event and clk='1' then
```

```
  case state is
    when state0 => state<=state1;
    when state1 =>
          mult_enable<='1';
          if mult_done='1' then
            state<=state2;
            mult_enable<='0';
          end if;
    when state2 =>
          mult_enable<='1';
          if mult_done='1' then
            state<=state3;
            mult_enable<='0';
          end if;
    when state3 =>
          mult_enable<='1';
          if mult_done='1' then
            state<=state4;
            mult_enable<='0';
          end if;

    when state4 => state<=state5;
    when state5 =>
          mult_enable<='1';
          if mult_done='1' then
            state<=state6;
            mult_enable<='0';
          end if;

    when state6 =>
          mult_enable<='1';
          if mult_done='1' then
            state<=state7;
            mult_enable<='0';
          end if;

    when state7 => state<=state8;
    when state8 =>
          mult_enable<='1';
          if mult_done='1' then
            state<=state9;
            mult_enable<='0';
          end if;

    when state9 =>
          mult_enable<='1';
          if mult_done='1' then
            state<=state10;
            mult_enable<='0';
          end if;

    when state10 => state<=state1;
    when others => state<=state0;
  end case;
end if;
```

```
  end process;

    r1_load<=state(10) and clk  when  state3 |state6 |state9,
          state(10)        when  others;
    r1_clear<=state(10);
    r2_load<=state(9) ;
    r3_load<=state(8)  ;
    r4_load<=state(7)  ;
    r5_load<=state(6)  ;
    mux1_1<= state(5)  ;  mux1_2 <=state(4 downto 3) ;
    coeff_sel<=state(2 downto 0);
end behavoiral;
```

REFERENCE

REFERENCE

[1]B. W. Bomar, "Finite Wordlength Effects," The Circuits and Filters Handbook, CRC Press, Boca Raton, 1995.

[2]E. C. Ifeachor and B. W. Jervis, Digital Signal Processing: A Practical Approach, Addison-Wesley, Great Britain, 1993, p. 416—419.

[3]B. W. Bomar, "On the Design of Second-Order State-Space Digital Filter Sections," IEEE Transactions On Circuits And Analysis, Vol. 36, No. 4, April 1989.

[4]B. W. Bomar, "New Second-Order State-Space Structures for Realizing Low Roundoff Noise," IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-33, No. 1, February 1985.

[5]R. N. Givhan, "A Comparison of Methods for Computing Sum-of-Products in Programmable Logic Devices," Master Thesis, The University of Tennessee Space Institute, August, 1997.

[6]N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," IEEE Symposium on FPGAs for Custom Computing Machines,Virginia Polytechnic Institute and State University, January 1995, p. 155—162 .

[7]L. M. Smith, B. W. Bomar and R. D. Joseph, "Floating-Point Roundoff Noise Analysis of Second-Order State-Space Digital Filter Structures," IEEE Transactions on Circuits and Systems—II : Analog and Digital Signal Processing, Vol. 39, No. 2, February 1992.

[8]L. Louca, T. A. Cook and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," IEEE Symposium on FPGAs for Custom Computing Machines, Rutgers University, 1996, p. 107—116 .

[9]B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," IEEE Transactions on Very Large Integration (VLSI) Systems, Vol. 2 , No. 3, September 1994.

[10]Z. Salcic and A. Smaliagic, Digital System Design And Prototyping Using Field Programmable Logic, Kluwer Academic Publishers, Boston, 1997, p. 49—97.

[11]D. A. Patterson, J. L. Hennessy, Computer Architecture—A Quantitative Approach, Morgan Kaufmann Publishers, California, 1996, A-23—A-25.

[12]B. W. Bomar, L. M. Smith, R. D. Joseph, "Roundoff Noise Analysis of State-Space Digital Filters Implemented on Floating-Point Digital Signal Processor," IEEE Transaction on Circuits and Systems—II Analog and Digital Signal Processing, Vol. 44, No. 11, November 1997.

[13]Altera Company, "Floating-point Megafunctions", Digital Library CD-ROM 1996,Altera, San Jose, 1996.

[14]Y. Jiang, S. P. Kim, "Block Digital Filter Structures and Their Finite Precision Response," IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing, VOL. 43. NO. 7, July 1996.

VITA

Bo Su was born in QingDao, China on January 23, 1970. He attended elementary school and high school in QingDao until 1987. In 1987 he entered Tsinghua Univeristy after a very competitive examination. During his time in Tsinghua University he majored in Electrical Engineering and ranked in the top 10% in the department of 200 students. He received his Bachelor's degree in 1992. From 1992 to 1996 he worked in Beijing, China as an electrical engineer. In Fall 1996 he was awarded a graduate scholarship in Lamar University in Texas and came to the United States to pursue a Master's Degree. He transferred to The University of Tennessee Space Institute in Spring 1997 majoring in Electrical Engineering. He completed the masters program at UTSI in Summer 1998. The master's degree was received in August 1998.

He will now be taking a position at Scientific Atlanta, a telecommunication company in Atlanta, Georgia.