Xilinx FPGA Implementation of Overflow Correction in a Wave Digital Filter Starting from a C-based Description

Reinder Nouta and Huibert J. Lincklaen Arriëns
CAS section, Department of Micro-electronics
Faculty of ITS, Delft University of Technology
Mekelweg 4, 2628 CD Delft, the Netherlands.
E-mail: R.Nouta@ITS.TUDelft.nl

Abstract— This paper describes that, under certain conditions, the area cost of parallel addition number overflow detection and correction can be zero when implemented in Xilinx FPGA's. A simple wave digital filter is shown as an example. A C-based description language can be used very efficiently.

Keywords— **FPGA; Digital Filter**

I. INTRODUCTION

In this contribution we discuss the overflow detection and correction in carry ripple parallel addition and its implementation cost using 4-input LookUpTables (LUT's) as being used in Xilinx FPGA chips.

In experiments we used a modern C-based description language (A|RT-Library + A|RT-C)[1] for description, Microsoft Visual C++[2] for compilation and simulation, translation to VHDL using A|RT-Builder[1], ModelSim[4] for VHDL simulation, Synplify[3] for VHDL synthesis, Alliance[5] for Xilinx placement and routing and finally a Nallatech[6] PCI board with a Xilinx Spartan-2[7] chip for implementation.

We show that under certain conditions the correction circuitry can be included in 4-input LUT's causing zero extra cost.

II. OVERFLOW CHARACTERISTICS AND NUMBER RANGES

Figure 1 shows as an example the number range of an 8-bits 2's complement binary word. It ranges from +127 downto −128. We show 4 ways overflow in this (example) 8 bit system can be handled. The choice of methods A, B and C from figure 1 can be specified in A|RT-C directly together with the addition involved, in one line of code.

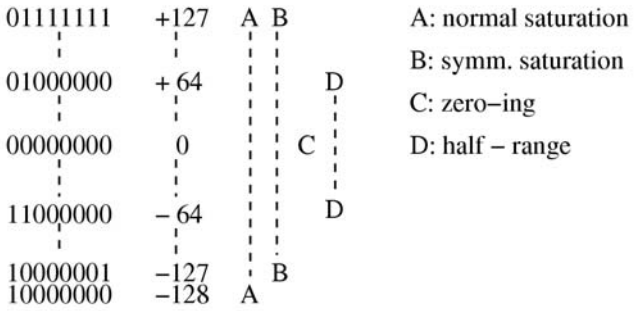

Figure 1: 8 bits overflow number ranges.

A. Normal saturation

When an addition of two positive numbers is detected to grow above +127, we call it positive overflow, the addition result is fixed at the upper limit of +127. When the addition of two negative numbers is detected to grow below −128, we call it negative overflow, the addition result is fixed at −128. Detection has to reveal which of the two limits has to be used.

B. Symmetrical saturation

The upper and lower limits used for overflow correction are now chosen to be symmetric: +127 and −127. In this case also: detection has to reveal which limit has to be used. We can take two different positions here:

1. We choose to use the whole available number space, only if an addition result is being detected to result in a number that will be too large for that number

space, the corresponding number limit is the addition result instead. This means in the example of figure 1 that the number −128 is a valid number. Only if the result will be larger than +127 or −128, the overflow correction will instead use the numbers +127 and −127.

2. We choose to use only the number space between +127 and −127. This means that a result of −128 means a negative overflow. This case means that not only detection has to reveal that a result will be too large for the available 8 bits number space (for figure 1) but also the value of −128 needs to be detected for overflow. Only this possibility is implemented in A|RT-Library.

C. Zero-ing

In this case, whenever overflow, positive or negative, is detected, the result is set to zero.

D. Half-range

The proposed positive and negative correction value (symmetrical) is half the range. Also in this case we have the choice to use the whole number space or not. We only discuss the choice where only a result larger than +127 or −128 means overflow and will be replaced by +64 or −64 (figure 1 example). This means a symmetrical saturation at half the range. This overflow method has not been implemented in the A|RT-C Library.

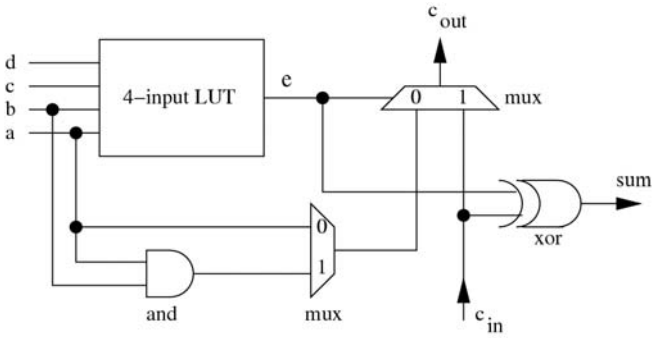

Figure 2: Xilinx architecture (simplified) showing 4-input LUT and fast carry path.

The basic building block of parallel adder circuits is the binary fulladder. This component has 3 inputs and 2 outputs. In terms of a 4-input LUT, that forms the base component in Xilinx FPGA logic cells, the fulladder can always be implemented using 2 of these LUT's. One input of these LUT's remains unused in this case. An *n*-bit carry ripple adder according to figure 3 will then need 2*n* LUT's.

For the Xilinx FPGA series 4000, Spartan, Spartan II, Virtex and Virtex II, Xilinx has created the possibility of using a fast carry line between 4-input LUT's for efficient implementation of carry ripple adders. This has the effect of using 1 LUT per fulladder instead of 2. It also means that only two inputs for each LUT are used instead of 3 and that placement is restricted to use the column approach of the implemented fast carry lines.

x	y	c_{in}	sum	c_{out}	e	mux_{0-in}
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	1	0	1	d
0	1	1	0	1	1	d
1	0	0	1	0	1	d
1	0	1	0	1	1	d
1	1	0	0	1	0	1
1	1	1	1	1	0	1

Table 1 LUT truth table for fulladder.

The (simplified) Xilinx logic cell architecture is shown in figure 2[8]. It shows a 4-input LUT and the fast carry path consisting of one MUX (multiplexer). It also shows the presence of an XOR for the sum output and an optional AND gate.

Table 1 shows how a fulladder having x, y and c_{in} as inputs and sum and c_{out} as outputs, is implemented using inputs a and b and the fast carry path.

The truth table shows that for $e = 0$, c_{out} needs to be a copy of input x (or y), implicating that the AND gate is not used.

A VHDL synthesizer like Synplify will use this fast carry line if a parallel adder is defined in VHDL as $sum = a + b$. This will need n LUT's for an n-bit parallel adder. When the adder is defined in terms of a hierarchical adder built from fulladders, the synthesizer will use 2n LUT's.

III. The C-based software used

We use A|RT-Library, A|RT-C and A|RT-Builder for description and translation to VHDL and Microsoft Visual C++ for compilation and simulation and testing of the C-descriptions.

A|RT-Library is a C++ linkable software library that allows the user to add fixed-point arithmetic to algorithms coded in a subset of C, including a number of overflow and truncation algorithms.

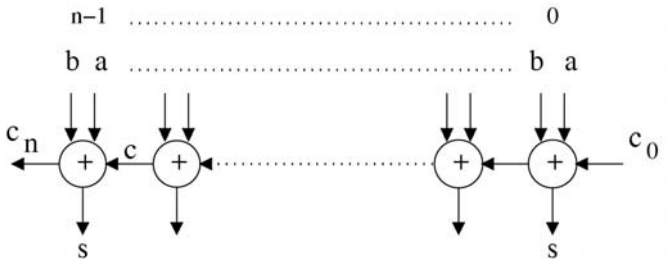

Figure 3: *n* bits carry ripple adder.

A|RT-Builder is a software tool that translates a C-based functional specification of an algorithm or function into an RTL HDL description, in our case VHDL. The input to A|RT-Builder is a subset of C, optionally enhanced with fixed-point classes as provided by A|RT-Library or SystemC.

The translation process to VHDL has the following important properties:

- C function definitions and properties will be interpreted as structure and translated into VHDL component definitions and instantiations,
- Sequential statements which are not function calls will be interpreted as behavior and translated into sequential statements inside a single VHDL process.

The translations process from A|RT-C to VHDL keeps the hierarchical description intact, which means that a description in A|RT-C for a parallel addition using the + operator, finally needs *n* LUT's in its implementation. Table 2 shows the number of LUT's needed when the addition is combined with one of the 3 possible overflow correction methods in A|RT-Library. It becomes clear that the addition with symmetrical saturation needs more than 2*n* LUT's. The reason for this is, that also the most negative number has to be detected (–128 in figure 1). This takes the extra LUT's.

	8 bits	16 bits	24 bits	32 bits
adder	8	16	24	32
adder + sat	16	32	48	64
adder + zero-ing	17	33	49	65
adder + symm. sat	18	38	58	75

Table 2 Number of LUT's needed.

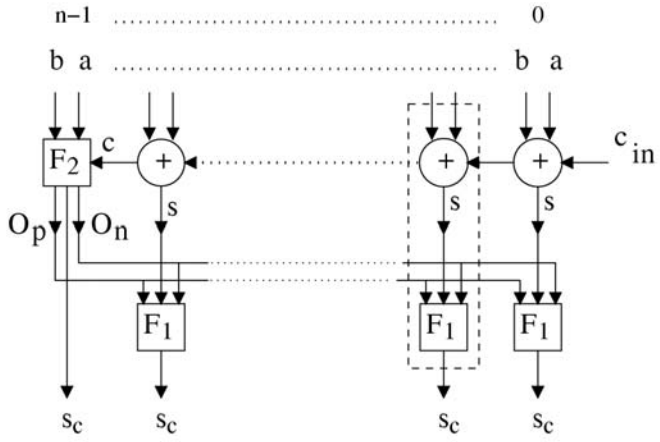

Figure 4: *n* bits carry ripple adder with general overflow detection and correction, method 1.

IV. Carry Ripple Adder Overflow

In figure 4 a block diagram is given. Overflow is detected at the most significant adder position.

A. Method 1:

Positive overflow will be detected there when $a = 0$, $b = 0$ and $c = 1$. This means that $s = 1$ will occur: the result word will be negative. Positive overflow will have been detected when $O_{pos} = 1$ and negative overflow will have been detected when $O_{neg} = 1$, with:

$$O_{pos} = \overline{a}\,\overline{b}\,c$$

$$O_{neg} = a\,b\,\overline{c}$$

B. Method 2:

Overflow detection can be done by considering c_n and s_{n-1}. Overflow is detected when these bits differ in value:

$$O = O_{pos} + O_{neg} = \overline{c}_n s_{n-1} + c_n \overline{s}_{n-1}$$

C. Method 3:

A fulladder at position *n* is added. With the proper sign extensions used, this converts c_n into s_n. Overflow can now be detected from here.

D. Method 4:

A|RT-C allows the user to effectively reduce the number of bits used, do overflow detection and correction as well as a certain truncation, all in one line of code. This method has in fact been used in the example.

Each bit position in figure 4 will need an extra LUT to implement the overflow correction. The total number is then 3*n*. If we choose to use the fast carry lines, the number will be 2*n*.

In the case we use method C or D from figure 1, we can use the simpler version given in figure 5. We now only need to know the existence of overflow ($O = O_{pos} + O_{neg} = 1$). Each bit position in the parallel adder together with overflow correction logic needed in these two cases can be written in VHDL as an entity-architecture combination having 4 instead of 5 inputs and 2 outputs leading to a 2*n* LUT implementation for a hierarchical VHDL description of the ***n***-bit parallel addition. The overflow correction needs no extra LUT space in these cases.

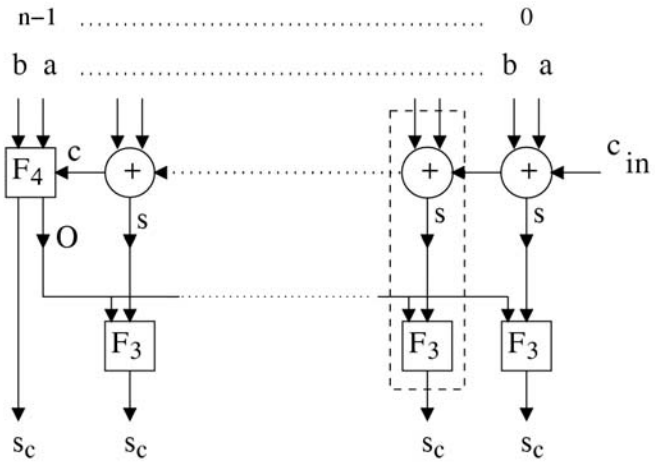

Figure 5: *n* bits carry ripple adder with special overflow detection and correction, method 1.

Important: if the fulladder with overflow correction circuit is being described in A|RT-C in the same way as in VHDL, simulation of the C-description may lead to wrong results because the C-description is sequential and the VHDL description is concurrent!

Formulas for the detection and correction circuits F_1, F_2, F_3 and F_4 can easily be found and are very simple.

V. Implementations of Overflow Corrections Using the Fast Carry Lines

As said before, Figure 2 shows a detailed picture of the logic cell including the 4-input LUT and the fast carry line for the Xilinx chips, while table 1 shows the fulladder implementation.

The question naturally arises whether the overflow logic can be included in this LUT so that the result would be that overflow has no cost when implemented in 4-input LUT's (for a moment we forget what synthesis software can give us, we simply ask the question whether it can be done anyhow).

Normally the overflow circuitry follows at the output of the adder, in this case following the sum output of the logic cell. It follows directly that extra circuitry cannot be included in the cell because the only place where the user can influence the content of the circuit is the programming of the LUT and the choice of using the AND gate or not.

If, however, we have the situation where the overflow corrected output number flows directly into another adder or subtractor, we may try to add the overflow correction circuitry into the next LUT involved with the next adder or subtractor. This, in fact can be done.

We have found the following for the various overflow characteristics involved:

A. Normal saturation (method A, figure 1)

Saturation (method A, figure 1) circuitry can be shifted into one input of a next adder LUT. We then need two extra inputs for that LUT: O_{pos} and O_{neg}. This means that when $O_{pos} = 1$, input *a* of the LUT has the effect of being 1, in the case $O_{neg} = 1$, input *a* has the effect of being 0 (this is for the case we describe a bit position not being the MSB).

The next possibility is when the next LUT is meant to implement a subtractor instead of an adder. In this case we need to distinguish between the adding and the subtracting input. It may also be needed to invert the definition of overflow: $O_{pos} = 0$ might be needed to define positive overflow instead of $O_{pos} = 1$. The same goes for negative overflow O_{neg}.

Under these conditions we have found that this saturation overflow logic is being absorbed completely in the next LUT (by the synthesis tool used) resulting in no extra cost of implementation.

If two adders with overflow logic at the output are both inputs to the same next adder, this can of course not being absorbed into the next 4-input LUT. We would need 6 inputs in this case.

B. Symmetrical saturation (method B, figure 1)

This case has not been investigated.

C. Zero-ing (method C, figure 1)

We have found that the overflow logic can always be absorbed into the next LUT, whether it is an adder or a subtractor. In this case we need only one bit for each overflow signaling. Also in the case that two inputs to the next LUT have overflow logic we have found that this logic will be absorbed in the LUT. It may again be needed to invert one (or both) of the definitions of overflow.

D. Half-range overflow (method D, figure 1).

This is being handled similar to the previous case of Zero-ing. Only one overflow bit is needed for each overflow corrected number which means that we can have two inputs together with their corresponding overflow bit connected to a 4-input LUT.

VI. WAVE DIGITAL FILTER EXAMPLE[9]

In figure 6 we show the second order damped resonator circuit as an example. Figure 6d shows the resulting recursive arithmetical structure. The coefficient α_1 controls the Q-factor of the circuit, the coefficient α_2 controls the resonant frequency value. From wave digital theory it follows that we should only detect and correct overflow at outputs of adapters. That is: no overflow should happen internally in the adapter arithmetic. This means for instance that an addition of 2 16-bits words needs a 17-bit adder. Accordingly, we have to reduce the number of bits at the outputs of adapters again to the starting 16 bits. In figure 6d we have restricted us to outputs of adapters that are in a recursive path: c_1, c_2 and c_3 are the places where bit reduction and overflow detection and correction has to occur (method 4). Correction logic c_1 can be absorbed by the LUT's of adder 1. The correction logic c_2 can be shifted through delay T_1 and absorbed by adder 4. The delay T_1 is then implemented by the flipflops at the outputs of the LUT's forming adder 3. Delay T_2 cannot be combined with adder LUT's and thus must be realized separately. The correction logic c_3 can be combined with it. When the correction logic is implemented as described, only one input of each adder involved, is used. This means that we have the choice of correction method, either saturation, zeroing, or half-range method can be implemented in this case. If we had chosen to absorb correction logic

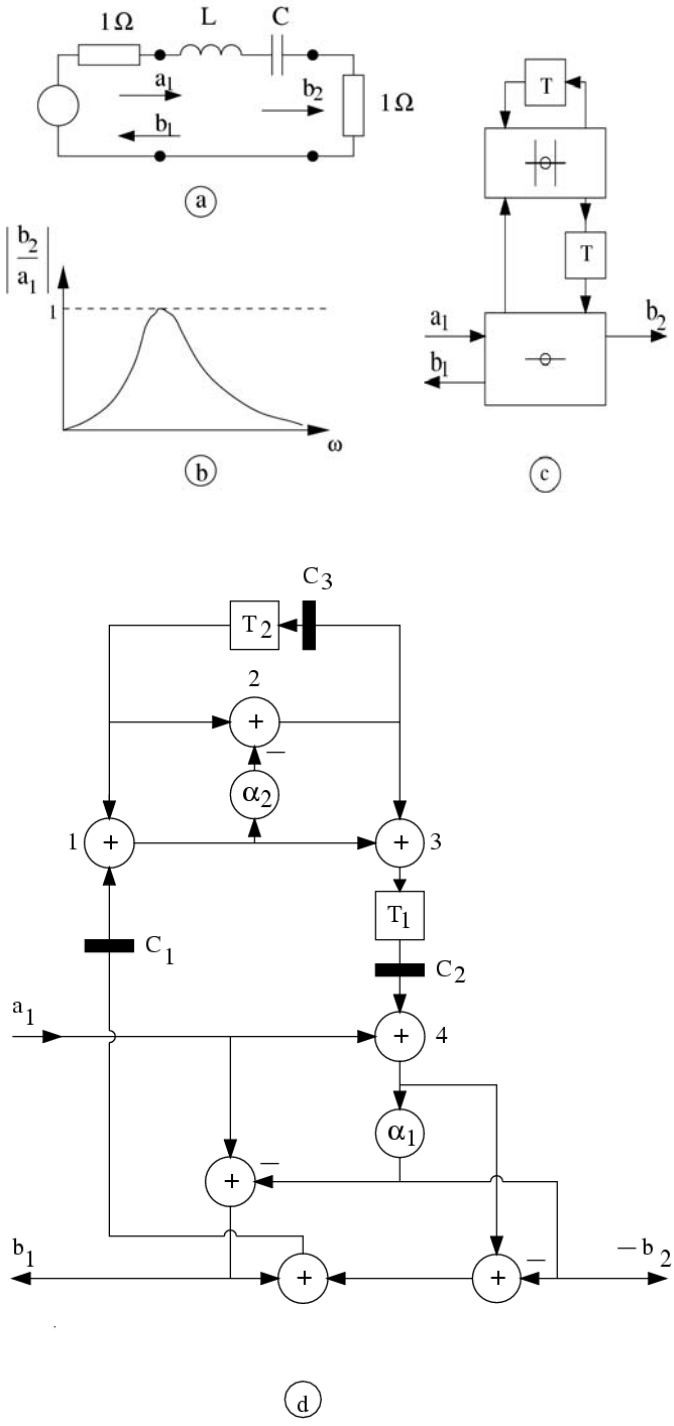

Figure 6: Wave Digital Filter: prototype circuit and digital structure.

c_3 into adders 1 and 2, we had been forced to choose between the zero-ing or half-range methods since adder 1 now having both inputs to absorb correction logic.

When the coefficients in figure 6*d* are chosen to be negative powers of 2 (resulting in just right shifts), the FPGA layout reveals that exactly 7 fast carry lines are

being used (the number of adders implemented is 7).

An A|RT-C description of this filter is given below.

```
// description of the wdf version of a
// second order damped resonator circuit
// alpha_1 and alpha_2 are restricted here
// to negative powers of 2,
// while alpha_2 = 1/4   ==>  fc/Fs = 0.115,
// and  alpha_1 = 1/16  ==>  Bandwidth =
// 0.0212

#include <fxp.h>
fxpOqc withSaturation(TRUNCATED,SATURATED);

#define BW 16

// definition and initialization of delay
// elements
Int<BW>    T2 = 0;
Int<BW+4>  T1 = 0;

void wdfx( const Int<BW>  a1,
                 Int<BW>& b1, Int<BW>& b2  )

{
    #pragma OUT b1 b2

    Int<BW>    u0, C1, C2;
    Int<BW+1>  s1, s4;
    Int<BW+2>  u1, u2;
    Int<BW+3>  s2;
    Int<BW+4>  s3;

    C2 = oqc( T1, &withSaturation);      // C2
    s4 = a1 + C2;
    u0 = s4 >> Uint<3>(4);          // alpha_1
    b2 = -u0;
    u1 = s4 - u0;
    b1 = a1 - u0;
    C1 = oqc( u1+b1, &withSaturation);   // C1
    s1 = C1 + T2;
    u2 = s1 >> Uint<2>(2);          // alpha_2
    s2 = T2 - u2;
    s3 = s1 + s2;
    // update delay elements
    T2 = oqc( s2, &withSaturation);      // C3
    T1 = s3;                        // delayed s3
}
```

VII. Conclusions

We have shown that overflow detection and correction circuitry can, under certain conditions, completely be absorbed in the 4-input LUT's that are used for parallel addition with fast carry lines, when the choice of implementation is a Xilinx FPGA. The result is zero extra area cost.

This can be obtained from a description in A|RT-C, which means that this high level description method does not introduce an area penalty when used together with a VHDL synthesizer for automatic generation of the Xilinx netlist.

The A|RT-C description is powerful and compact and the A|RT-Builder tool does its job reliably.

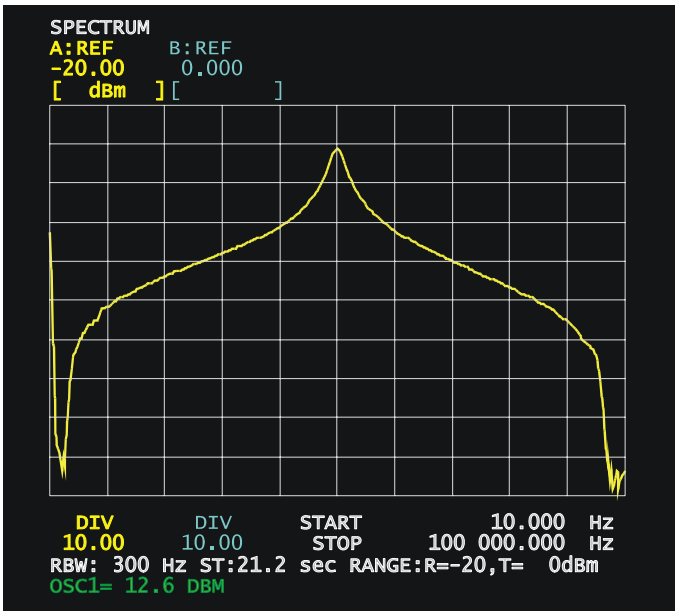

Figure 7: spectrum plot of the example Wave Digital Filter when implemented on the Strathneuy PCI Card.

References

[1] A|RT-Library + A|RT-C + A|RT-Builder Version 2.2, Frontier Design Inc., now Adelante Technologies Inc., www.adelantetech.com.

[2] Microsoft Visual C++ 6.0 with Service Pack 5.

[3] Synplify Pro 7.0, Synplicity Inc.

[4] ModelSim SE Plus 5.5, Model Technology Inc.

[5] Alliance 3.3.08i, Xilinx Inc.

[6] Nallatech Ltd., Strathnuey PCI Card.

[7] Xilinx Spartan 2 XC2S150 chip.

[8] Xilinx Application Note: XAPP215(v1.0) June 28, 2000. Design tips for HDL implementation of Arithmetic functions, Steven Elzinga, Jeffrey Lin, Vinita Singhal.

[9] A.Fettweis, *Digital filter structures related to classical filter networks*, A.E.U., band 25, pp. 79 - 89, 1971.