A Reconfigurable Coprocessor for Redundant Radix-4 Arithmetic

Alodeep Sanyal[1], **Rajat Shuvra Ghoshal**[1], **Achintya Das**[1]
and **Susmita Sur-Kolay**[2]

[1]Department of ECE, Kalyani Govt. Engineering College, Kalyani
[2]Advanced Computing & Microelectronics Unit, ISI, Calcutta

Abstract

We present the implementation of a reconfigurable arithmetic coprocessor based on a fast parallel multiplication scheme proposed in [1]. In this coprocessor, we have implemented four basic arithmetic operations (viz. addition, subtraction, multiplication and complementation) and four primary logic operations (viz. AND, OR, EX-OR and NOT).

The coprocessor can be directly accessed from the PC by an interfacing software implemented in [19].

In this project, we have developed a complete set of VHDL modules, which through different stages of Xilinx Foundation Express 3.1i, finally give rise to the bitstream file which is downloaded from the PC to the FPGA board to configure the FPGA chip (target architecture: XC4010E^{TM}) as the desired arithmetic coprocessor.

Acknowledgement

In the beginning, we would like to pay our sincerest thanks and gratitude to Prof. Bhabani P. Sinha for allowing us to use his paper on “Fast Parallel multiplication using redundant quarternary number system” [1] for FPGA implementation. We would also like to acknowledge here the thesis work by Koushik Sinha [20] as the first step towards FPGA implementation of the above mentioned algorithm.

We would like to take this opportunity to express our special thanks to Prof. Miguel Ángel Aguirre Echánove [19] too, without whose kind cooperation this project work could not have been completed in its present perspective.

We would also like to pay our gratitude to Xilinx and XESS and their many kind people for providing all sorts of technical supports, especially to Mr. David E Vanden Bout whose on-line book titled “Pragmatic Logic Design” helped us a lot in learning the Xilinx Foundation Express software with all its utilities.

---

[*]*Communicating Author:* Alodeep Sanyal (*Email*: alodeep2k@yahoo.com)

Contents

Chapter 1

Introduction

1.1 Introduction

The giant strides humanity has taken in terms of technological progress, though unfathomable, has been punctuated by significant achievements. The hallmark among them being in the fields of VLSI Design. The greatest advantage in this field lies in the immense scope for future developments. Improved system organization and efficient computing algorithms have added to the reliability of high-speed processing. That too at a low and affordable price. As a result computers have circumvaletted human progress with a touch of finesse and efficiency and encroached into spheres like animation, electronic design, telecommunications, space research and innumerable other ones.

One key element which has enabled computers to meet such large volumes of real-time computations is the use of subsidiary processors alternatively termed coprocessors. The coprocessors are capable of computing a selected small subset of operations at a very high speed. The advantage of coprocessors is that they can be purely application specific. They improve the overall performance of the processor by executing instructions in parallel with the main processor. The hardware logic is designed specially keeping in mind the job it is required to perform and also the system specifications. As an example the 80287/80387 math coprocessor is designed to work in parallel with 80286/80386 processor. The coprocessor is designed to perform many powerful floating point operations. This frees the main processor from performing the floating point computations while the coprocessor simultaneously performs the numeric calculations. Besides numeric computations, coprocessors are also available for other specific computation – intensive application areas such as signal processing, graphics, image processing and many others.

In recent times, designing coprocessors for parallel fast multiplications of numbers has become an important field of research. Several algorithms have been developed for multiplication of binary numbers that are easily being implemented on a VLSI chip. For example, Nakamura in [2] proposed an algorithm for iterative array multiplication that requires O(n) time to multiply 2 n-bit binary numbers and can be implemented on a VLSI chip using an almost regular interconnection structure among the processing element. Takagi *et. al.* [3] proposed an O($\log n$) multiplication scheme using *redundant binary trees*. The authors in [4] reported two other parallel algorithms for multiplication of two n-bit numbers using O($\log n$) parallel addition technique proposed in [6] using the concept of *precarry*. One of these algorithms requires $n + \lceil \log_2 n \rceil$ time while the other requires approximately $\lceil 3 \log_2 n \rceil$ time. Authors in [5] have given a parallel multiplications scheme which requires $2\lceil \log_2 n \rceil + 2$ time in *balanced ternary number* system using the technique of *column compression* and also the concept of precarry. An

algorithm for multiplication based on combination of *redundant radix-4 (RR-4)* representation of numbers and the divide-and-conquer policy used in Karatsuba-Ofman algorithm [8] was developed by Mehlhorn and Preparata [7]. Their method required O(*log n*) time for multiplication of two n-bit numbers using O($n^{1.59}$) logical elements. Also fast parallel algorithms for radix-2 and radix-4 modular multiplication in (n+1) and *én/2ù* + 1 time respectively have been designed [9] for multiplication of two n-bit numbers.

1.2 Scope of the work

This project work deals with the actual FPGA implementation of a new fast parallel multiplication technique using the redundant quarternary or radix-4 number system (RR-4) as proposed in [1], which is faster than any of existing multiplication techniques. This high speed VLSI multiplication scheme can multiply two m-digit by m-digit radix-4 numbers in just $\lceil (1/2)\ log_2\ m \rceil + 1$ steps of addition of four radix-4 numbers. For m digit by m-digit radix-4 integer multiplication, we first generate m partial products, each of (m+1) radix-4 digits. These partial products are added four at a time by means of redundant quarternary adders. The parallel addition of four (m+1) radix-4 numbers can be performed in constant time, independent of m, without the generation and propagation of carry. The number of computational elements required is O(m^2). Because of the regular cellular array structure, it is suitable for VLSI implementation with O($m^2\ log_2\ m$) AT-value. We have implemented this multiplication algorithm on Xilinx FPGA board (XC4010E^{TM}). The initial FPGA implementation of this algorithm by Sinha, Roy and Sur-Kolay [20] addressed the implementation of the RR-4 multiplication scheme. The present authors have extended the arithmetic scheme over addition, subtraction and negation and incorporated the basic bit-wise logic operations (AND, OR, NOT and EX-OR) to make it a complete arithmetic co-processor.

FPGA was chosen as the implementation device because of the many advantages offered by the *Field Programmable Gate Arrays*. They have a very simple and regular structure and are scalable, making them extremely suitable for VLSI fabrication. Another advantage of using FPGA is that they are also easily reconfigurable, allowing the flexibility of implementing several functions on the same FPGA device, possibly simultaneously, at different parts of the FPGA device.

The rest of the project report is organized as follows. In *Chapter 2*, a detailed outline of the multiplication algorithm using RR-4 digits is provided. In *Chapter 3*, the implementation logic for the coprocessor given in [1] is discussed in details. *Chapter 4* provides a general conclusion pointing towards the future advancement in this aspect.

Chapter 2

Multiplication in Redundant Radix-4 Number System

2.1 Introduction

The multiplication technique proposed by De and Sinha [1] using radix-4 number representation uses one of the signed-digit (SD) number representation introduced by Avizienis[10] to multiply two m-digit numbers in RR-4 system.We will first discuss about the RR-4 number system.Next the algorithm for conversion from binary to RR-4 system is described, followed by multiplication algorithm as proposed in [1].

2.2 The RR-4 Number System

In RR-4 number system the radix used is 4 and individual digits belong to the set, S={-3,-2,-1,0,1,2,3}, An m-digit redundant radix – 4 integer $Y=[y_{m-1}....y_0]_{RR-4}$,where for all i, $y_i \in \{-3,-2,-1,0,1,2,3\}$ and has the value $\Sigma\, y_i . 4^i$ where i ranges from 0 to (m-1). There are more than one possible representation of the same integer in RR-4 number system. For example, $[0\ 3\ 1]_{RR-4}$, $[1\ -1\ 1]_{RR-4,}$ and $[1\ 0\ -3]_{RR-4}$, all represent the number $(13)_{10}$. This redundancy in number representation will be exploited to perform carry propagation – free addition, thereby allowing the parallel addition of four RR-4 numbers in *O(1)* time, independent of the length of the numbers.

Of the different possible representation of RR-4 digits, one possible way of writing the digits of set S using three binary bits for each digit are as follows, where the leftmost bit is 0(1) if the digit is positive(negative):

$(-3)_{RR-4} = 111$
$(-2)_{RR-4} = 110$
$(-1)_{RR-4} = 101$
$(0)_{RR-4} = 000$
$(1)_{RR-4} = 001$
$(2)_{RR-4} = 010$
$(3)_{RR-4} = 011$

The representation of any RR-4 number using binary bits can be visualized by a matrix of 0's and 1's as follows, where each column represents the respective RR-4 digit:

$$(2\ -1\ 0\ 3\ 1)_{RR-4} = \begin{pmatrix} 0\,1\,0\,0\,0 \\ 1\,0\,0\,1\,0 \\ 0\,1\,0\,1\,1 \end{pmatrix}$$

The topmost bit in each column indicates the sign of the digit, where 0 stands for positive and 1 stands for negative digit.

2.3 Binary to RR-4 Conversion

Given a binary number in sign-magnitude form, we can easily convert it to equivalent RR-4 representation by first grouping each pair of bits, starting fro the least significant bit (lsb) position of the magnitude part of the number and the attaching a sign bit to every such pair to obtain a representation similar to that shown in the example above. We might need to pad a 0 at the leftmost end, if necessary. We add a 0 bit to a pair of bits to indicate a positive RR-4 digit and a 1 to indicate negative RR-4 digit.

If the binary number is given in two's complement form, then we proceed as follows:

STEP 1: First check the most significant bit (msb) of the number. If it is 0,then proceed with the remaining bits in the same way as for a binary number in sign-magnitude form to obtain the equivalent RR-4 number. If however, msb is 1 then do the following steps,

STEP 2: Complement the remaining bits of the binary number and then group them pairwise starting from the lsb. A 0 may be padded at the leftmost end,if necessary.

STEP 3: If any group of bits is 11,then the corresponding RR-4 digit will be -1, along with the generation of an RR-4 carry digit of 1 to the next higher digit position.

STEP 4: Now collect the carry digits from each group of 11 bits to construct a carry vector in RR-4 system. The RR-4 digits for the other bit pairs will be obtained in the same way as discussed in Step 2 of the algorithm. Now the three RR-4 numbers:

i> the number obtained from the pair of bits
ii> the carry vector, and
iii> a carry of 1 at the least significant RR-4 digit position,

are added to get a new RR-4 number. This addition can be done in constant time using carry propagation-free addition described earlier.

STEP 5: The sign bit of each RR-4 digit is complemented to get the equivalent representation in RR-4 number system of the binary number.

2.4 RR-4 to Binary Conversion

An RR-4 number may contain both positive and negative digits. If there is no negative digit, then to convert it into binary each of the digits of RR-4 number is changed to binary Deleting its sign bit. But if the RR-4 number has negative digit then to convert it into binary we do as follows:

STEP 1: Two vectors are generated, one with the positive digits putting 0 in the place of negative digits and the other with negative digits putting 0 in the place of positive digit.

STEP 2: The second vector is subtracted (using 2's complement addition) from the first one to get the binary number equivalent of the given RR-4 number.

Based on the above principle, the following logic has been developed:-

Let $A_3A_2A_1A_0$ be a RR-4 number, where A_k is represented by sa_1a_0 ($\forall\, k=0,\ldots,3$). Let P and N be two binary vectors with 4 pairs of bits is generated from the RR-4 numbers. Let each pair be represented by p_1p_0, same as in the case for N.

$$p_{0i} = \bar{s}\, a_0$$
$$p_{1i} = \bar{s}\, a_1 \quad \forall \; i=0,1,\ldots,3$$

$$n_{0i} = \overline{a_0} + \bar{s}$$
$$n_{1i} = \overline{a1} + \bar{s} \quad \forall \; i=0,1,\ldots,3$$

STEP 3: In the third and final step, a full adder stage ($FA_k \;\forall\; k=0,1,\ldots,7$) is employed to obtain the final binary output bits (b_i) and carry bits (c_i).

$$b_k = (p_{ij} \oplus n_{ij}) \oplus C_{in} \quad \forall \; i = 0,1,..,3 \;\&\; j = 0,1$$

$$c_k = p_{ij} \bullet n_{ij} + (p_{ij} \oplus n_{ij}) \oplus C_{in} \quad \forall \; i = 0,1,..,3 \;\&\; j = 0,1$$

The point to be noted is that the carry bit for the first full adder operation (FA_0) is 0 and the carry bit from the last full adder operation is *Cout* which will be neglected.

2.5 An Overview of the Multiplication Algorithm

Once the given binary number is converted to its equivalent redundant radix-4 representation, the multiplication of two $m \times m$ RR-4 numbers is performed in $\lceil (1/2)\log_2 m \rceil + 1$ computational steps. The algorithm is carried out in two phases as follows:

STEP 1: The digit-by-digit products are represented by two digits of the RR-4 number system in order to generate two vectors corresponding to each partial product. The two vectors are generated in such a way that they can be added in constant time by parallel carry propagation – free adders to generate the required partial product. The redundancy in representation of numbers in RR-4 system is used to achieve this.

STEP 2: From step 1 we obtain m partial products, each consisting of (m+1) RR-4 digits. These m partial products are the added in parallel, four at a time, by a set of redundant quarternary adders in $\lceil log_4\ m \rceil$, i.e, in $\lceil (1/2) log_2\ m \rceil$ steps . Each step of this addition process involves two substeps:

i> generate the intermediate sum and carry vectors for each group of four partial products

ii> add these sum and carry vectors using a carry propagation – free addition process.

The first substep can be implemented by using ROMs. The ROM based design is modular in nature and most suitable for VLSI implementation. The number of computational elements required will be $O(m^2)$, thus giving an $O(m^2 log\ m)$ AT-value.

It is possible to eliminate the carry propagation-free addition in the partial product generation phase. For that, we would have to keep the digit by digit product as it is (i.e without using any redundant representation), generating two vectors for each partial product. Then these 2m vectors would be added in parallel by a set of approximately $2m^2/4 = m^2/2$ redundant quarternary adders in $\lceil log_4\ 2m \rceil$ steps. However the drawback is that the method would require $m^2/2$ more computational elements than the former technique.

2.6 Generation of Partial Products

In this section, the method for generating partial products to multiply two m-digit number by m-digit redundant quarernary numbers will be described. It is to be noted that an m-digit RR-4 number corresponds to an 2m+1 bit long signed binary number.

Let A and B represent the two RR-4 numbers to be multiplied. Then $P_i = b_i A$, is the i-th partial product. We represent the j-th digit of the I-th partial product by p_{ij}. Since we are multiplying in RR-4 number system, each p_{ij} can take on a maximum value of +9 and a minimum value of –9, which needs two RR-4 digits for representation.

Now the aim is to generate a digit pair $[c_{ij}(2)\ c_{ij}(1)]$, with weight of $c_{ij}(2)$ four times that of $c_{ij}(1)$ for each digit product p_{ij} in such a way that the sum of $c_{ij-1}(2)$ and $c_{ij}(1)$ becomes a carry propagation –free addition for all j, $1<=j<=m-1$.

Consider first a redundant radix – r number system in which the digits belong to the set S= {-(r-1),(-r-2),……..,-10,1,2,……(r-2),(r-1),}. Here the magnitude of the product of two

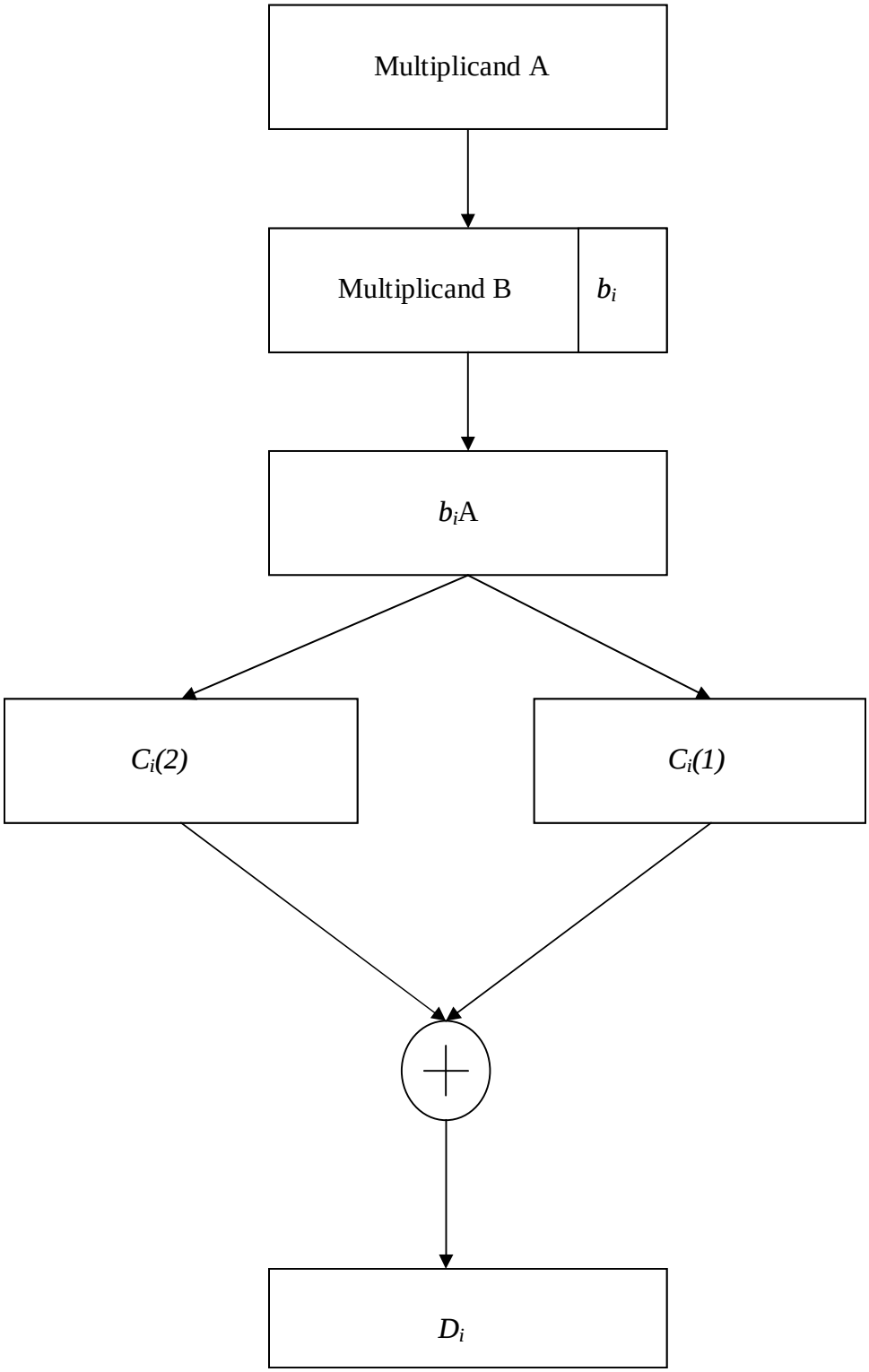

Figure 2.1 Generation of Partial Product of (n+1) digits

digits can have a maximum value of $(r-1)^2 = (r-1)r + (-(r-1))$. This can be represented by two digits as [r-1 -(r-1)]. An alternative representation of $(r-1)^2$ is [r-2 1]. We can generalize the above result in the for of the following lemma given in [1]:

LEMMA 1: In multiplying two m-digit numbers in redundant radix – r number system, the product of two digits can always be represented in two different forms with two different pairs of digits, except in the case when the product is a multiple of r.

Below, we outline the proof of this lemma as given as given in [1] for the sake of completeness.

Let us first consider a positive single digit by single digit product q, which we can represent as q=[g h], where $0 <= g <= (r-1)$ *and* $-(r-1) <= h <= (r-1)$, $h \neq 0$, (i.e q is not a multiple of r).

Note that, if g=r-1, h must be negative. We an write q as, $q=g \cdot r + h = (g + 1)r + (h - r)$, when h is positive and $q = g \cdot r + h = (g-1)r + (h + r)$ when h is negative.

For negative product value q=[g h] we similarly note that if g= -(r-1) then h must be positive. Hence, we can write $q= g \cdot r + h = (g+1)r + (h-r)$, when h is positive and $q=g \cdot r + h = (g-1)r + (h + r)$, when h is negative.

Thus we find that we an represent any digit q in redundant radix – r system as at least two different digit pairs, provided q is not a multiple of r, which completes the proof.

From the digit pair $[c_{ij}(2)\ c_{ij}(1)]$ generated for each digit-product p_{ij} , we construct two vectors $C_i(1)$ and $C_i(2)$ such that,

$$C_i(1) = c_{i,m-1}(1)\ c_{i,m-2}(1) \ldots\ldots\ c_{i,0}(1)$$
$$C_i(2) = c_{i,m-1}(2)\ c_{i,m-2}(2) \ldots\ldots\ c_{i,0}(2)$$

The i^{th} partial product is expressed as the sum of two vectors $C_i(1)$ and $C_i(2)$ (Fig.2.2). The resultant sum vector is denoted by D_i . A scheme for generating the digits of D_i is shown in the fig. In order to produce a carry propagation – free addition of two vectors, we use the *look-up table* shown in Table 2.1. The look-up table gives the conversion rule to generate the digit pair $[c_{ij}(2)\ c_{ij}(1)]$ for each possible value of the digit-product p_{ij}. For each possible value of $p_{ij,}$ we generate the digit pair $[c_{ij}(2)\ c_{ij}(1)]$ in such a way so that the sum of $c_{ij}(2)$ and $c_{ij}(1)$ becomes a carry propagation- free addition for all j. To implement this we observe the sign of previous digit-product $p_{i,j-1}$. If the sign of $p_{i,j-1}$ is positive we set $c_{ij}(1)$ to a negative value so that the magnitude of the sum of $c_{ij}(1)$ and $c_{ij}(2)$ never exceeds 3 and accordingly adjust $c_{ij}(2)$. Similarly for negative value of $p_{i,j-1}$, we choose $c_{ij}(1)$ to be positive. Table 2.1 shows the entries corresponding to only the positive

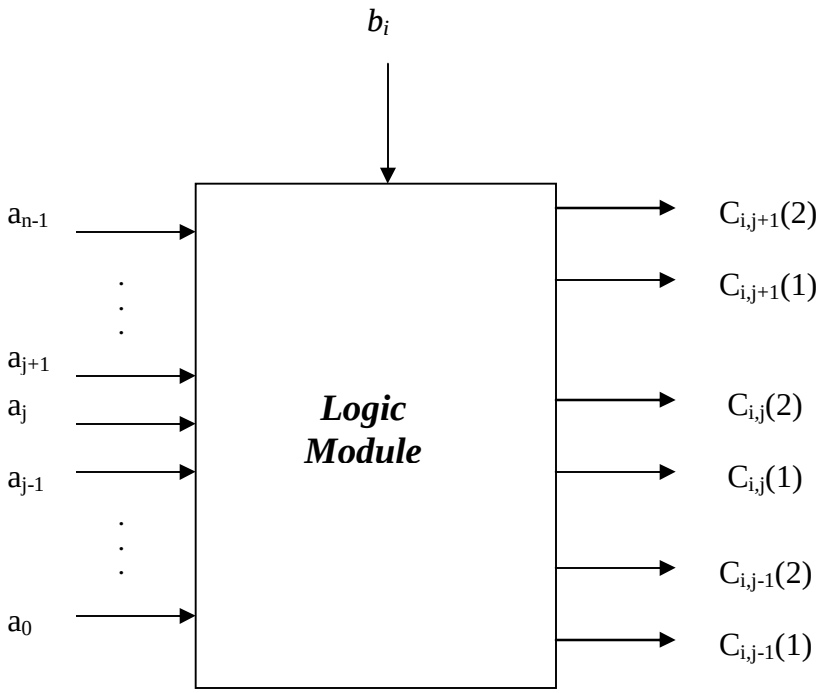

Figure 2.2 Initial computation of two vectors $C_i(1)$ and $C_i(2)$ from b_iA

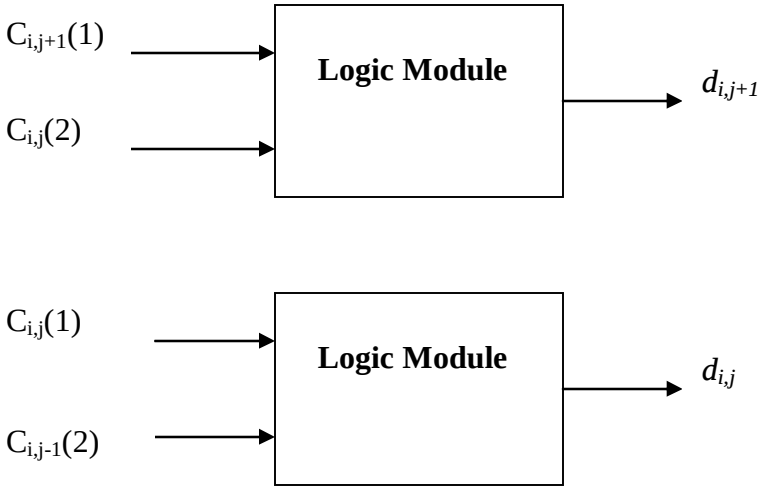

Figure 2.3 Reduction of two vectors to a single vector Di by carry propagation free addition

values of digit-product p_{ij}. For negative values of p_{ij} '*s* it can be shown that we just need to negate the conditions and entries of Table 2.1.

2.7 Carry propagation-free Addition of Partial Products

The addition of four redundant quarterary number X,Y,U and V is performed in two steps. In the first step we determine the intermediate sum s_i at each digit position i, $0 <= i <= m$, satisfying the relation $x_i + y_i + z_i + v_i = 4\, c_i + s_i$, where x_i, y_i, u_i, v_i are the digits to be added and c_i is the carry generated.

In the second step we get the final sum digits z_i's by adding s_i and c_{i-1} generated in the first step. No carry propagation is involved in this process. To determine s_i's and c_i's in the first step we follow the following conventions:

(1) If the digit sum at the i^{th} digit position (DS_i) is positive and there is a possibility of a positive carry from the previous lower order position (i.e the value of $x_i + y_i + z_i + v_i$ lies between 1 and 2), then we choose the s_i as negative or zero and adjust c_i accordingly. On the other hand, if there is a possibility of a negative carry from the previous digit position, we choose s_i to be positive and adjust the corresponding c_i. The rules for computing s_i and c_i are given in Table 2.2.

(2) For negative digit sums (DS_i), we need to negate the conditions in columns 2 and 3 of Table 2.2. and the intermediate sum and carry digits, s_i and c_i so as to generate carry propagation-free addition.

Thus it is possible to compute s_i and c_i by examining x_i , y_i , u_i ,v_I, x_{i-1}, y_{i-1} , u_{i-1} and v_{i-1}, and hence all of s_i's and c_i's can be computed in parallel. We assume that the digits x_{-1} , y_{-1}, $u_{-1}$and v_{-1} i.e to the right of the least significant digit position are all 0's. Thus it follows just by observing the twelve digits we can compute the final digit sum z_i. We hence have an addition procedure in constant time.

The steps involved in the generation of partial products and the method of carry propagation-free parallel addition of four redundant radix-4 numbers are illustrated in Figs. 2.4 and 2.5 respectively.

$P_{i,j}$	$p_{i,j-1}$	$C_{i,j}(2)$	$C_{i,j}(1)$
9	-	2	1
6	Positive value Negative value	2 1	-2 2
4	-	1	0
3	-	1	-1
2	Positive value Negative value	1 0	-2 2
1	-	0	1
0	-	0	0

Table 2.1. Generation of digit- pair corresponding to a digit product

DS_i	*Positive* DS_{i-1}		*Negative* DS_{i-1}	
	c_i	s_i	c_i	s_i
12	3	0	3	0
11	3	-1	2	3
10	3	-2	2	2
9	3	-3	2	1
8	2	0	2	0
7	2	-1	1	3
6	2	-2	1	2
5	2	-3	1	1
4	1	0	1	0
3	1	-1	0	3
2	1	-2	0	2
1	1	-3	0	1
0	0	0	0	0

Table 2.2 Sum and Carry generation during addition

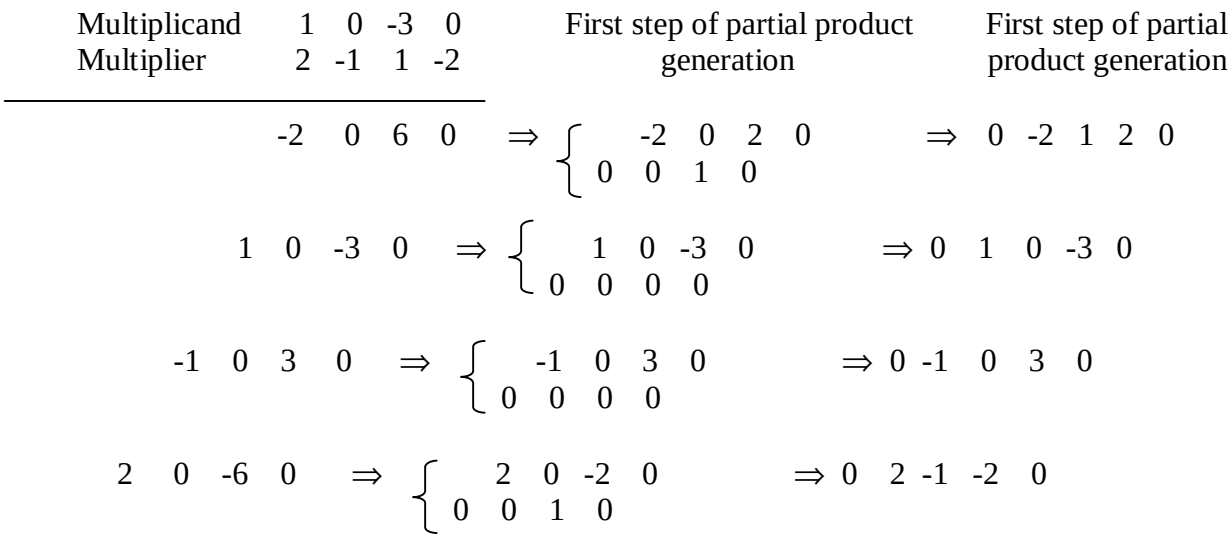

Figure 2.4 An example explaining the steps involved in the partial product generation

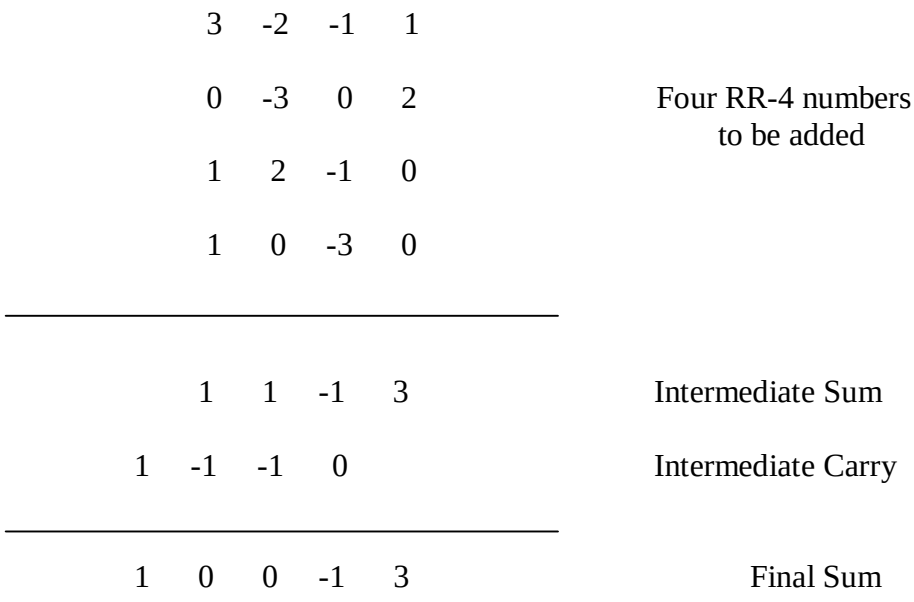

Figure 2.5 Example of carry propagation-free addition of four RR-4 numbers

Chapter 3

Implementation Details

3.1 Introduction

Our RR-4 arithmetic coprocessor is a 8-bit RISC processor and its design consists of two separate units: **1) Arithmetic Logic Unit (ALU)** and **2) Bidirectional Interface Unit** (which causes the interaction between a server program running on the PC and the RR-4 ALU platformed on the XS40 Board).

The RR-4 ALU consists of a i) binary ó RR-4 conversion unit *(binRR4.vhd, full_adder.vhd* and *RR4bin.*vhd), ii) an arithmetic unit *(arithmetic_unit.vhd, RR4_adder.vhd, RR4_multiplier.*vh*d* and *partial_product.vhd),* iii) a logic unit *(logic_unit.vhd),* iv) a control unit *(control.vhd),* and v) top level of the RR-4 ALU *(system.vhd).*

The Bidirectional Interface Unit on the other hand, includes i) a register unit (*regsinc.vhd* and *mux4.vhd*), ii) a 7 segment display unit (*7segmentoskk.vhd*), iii) an FSM protocol unit (*fsmrdwr.vhd*), iv) a decoder unit (*decod.vhd*), v) a glitch-filtering unit *(ffcleaner.vhd*), vi) an output unit *(muxsal.vhd)* and vii) top level of the IU *(sppinterf.vhd)*. A server program written in C++ (*spp.cpp* along with two accessory programs *xsboard.cpp* and *xsboarddlg.cpp*) runs in the PC environment to interact with the RR-4 ALU for the purpose of instruction and data transfer. This application was developed by Miguel Ángel Aguirre Echánove *et. al.* [19].

3.2 Architecture of the Coprocessor

3.2.1 The Arithmetic Logic Unit

The arithmetic logic unit (Fig. 3.1) is composed of the following separate logic blocks which are neatly interconnected by a detailed glue logic established in the top level VHDL file (system.vhd).

1. **Binary to RR-4 Conversion Unit:** The arithmetic operations in the RR-4 ALU are preceded by the conversion of 8 bit input operands into their 12 bit equivalent RR-4 representation (each RR-4 digit comprising of 3 binary bits). This operation is delineated in the VHDL file *binRR4.vhd.*

   After the completion of arithmetic operations, the RR-4 output digits are reconverted to binary bits following the logic described in the VHDL codes *RR4bin.vhd* and *full_adder.vhd*.

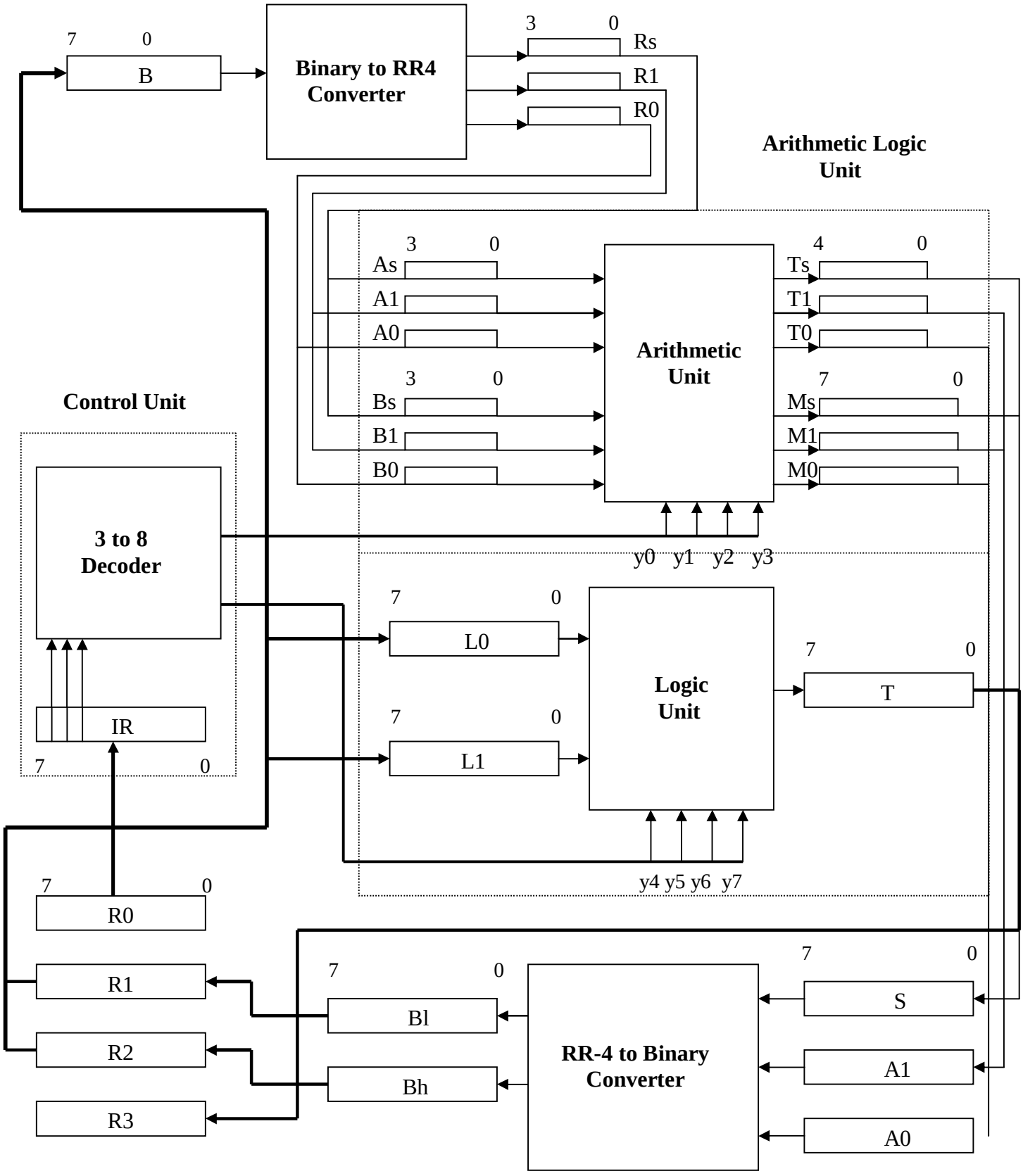

Figure 3.1 Block diagram representation of the RR-4 Coprocessor

Finally, the eight, ten or sixteen bit binary outputs for logical, addition or multiplication operations respectively are redestined to the operands registers of the Bidirectional Interface Unit (BIU).

2. **Arithmetic Unit:** The arithmetic unit of the RR-4 coprocessor performs four basic arithmetic operations using the RR-4 algorithm as discussed earlier:

    i) addition,
    ii) subtraction,
    iii) multiplication, and
    iv) complementation

    The addition circuitry is composed of two distinct modules RR4_adder.vhd and partial_product.vhd and gives the final sum in RR-4 representation.

    The subtraction circuitry is nothing but the same circuit as the addition only with the difference of changing the sign bits of the second operand (as the subtraction can be performed by 2's complement addition of the second operand to the first).

    The multiplication is the most complicated and time consuming operation of the coprocessor and it operates by repeated execution of the modules RR4_multiplier.vhd and partial_product.vhd. The final output digits are brought about by RR-4 addition of the partial products.

    The fourth arithmetic operation of the coprocessor is the complementation of a given operand. This simple operation is carried out by taking the advantage of the sign-magnitude form of the RR-4 number system. The change of sign of the operand sign bits keeping the two magnitude bits of each RR-4 digit unchanged engenders the complemented output of the operand.

3. **Logic Unit:** The logic unit also performs four basic logic operations:

    i) AND,
    ii) OR,
    iii) NOT, and
    iv) X-OR.

    The logic circuitries are nothing to do with the RR-4 representation of the numbers. Therefore, the logic unit is a conventional logic circuit as seen in a traditional processor.

    The logic unit takes two 8-bit operands (for first, second and fourth operations), performs bitwise logic operations and produces an 8-bit output.

4. **Control Unit:** The control unit of the coprocessor is rather simple and consists of an instruction register (IR) and a 3-to-8 decoder unit. The decoder generates 8

control signals by decoding the instructions corresponding to the operations performed by the coprocessor.

5. **Top level module:** The top level module of the coprocessor encapsulates all the separate units described in it and provides the interconnection between the separate units.

3.2.2 Bidirectional Interface Unit: This unit comes with two modules [19]:

1. A set of routines in C++ that communicates between the XS40 board and the PC in both directions.

2. VHDL design modules that controls the interface with the PC and the FPGA.

This unit consists of a 8 bit RAM with 4 address. This memory can be accessed from a host (PC) through the parallel port, in both write and read mode. The board display shows the address that is selected when we are writing or reading.

The parallel port is connected to the programming connector. No additional hardware is needed. The parallel port must be configured in SPP mode, because in this mode we can control every single control and status line from software routines. If any other mode is selected, the nSTROBE line will toggle when the port is written to, and the nPROGRAM signal will erase the current programming target.

3.2.2.1 C++ program: According to the authors [19], this program was developed in Borland C++ linked with the port access library DlPortIO, obtained from Scientific Software Tools, Inc. It runs fine in W95/NT. First of all the address and port lines assignment are documented in the source code. These issues were considered in the design phase:
1. The base address for the LPT1: port is fixed. It's not very difficult to obtain it automatically or make it variable.
2. Some lines must be toggled due to port and on board inversions.
3. Status register transfers through bit 1 to 4. We must shift right in low nibble and left in high nibble.
4. The program successfully communicates for our sample, in a dialog window. It's very easy to modify the program for other purposes.

3.2.2.2 VHDL Program: The VHDL design consists of the following modules:

i) register unit,
ii) FSM protocol unit,
iii) decoder unit.
iv) glitch filtering unit,
v) output unit,
vi) seven segment display unit, and
vii) top level BIU module (glue logic).

Fig. 3.2 shows a block diagram with the basic elements. The most interesting block is the Finite State Machine that controls the dialog with the parallel port (Fig. 3.3).

3.3 Instruction set of the Coprocessor

This RISC Coprocessor performs eight distinct operations whose command words are given below:

1. Addition:	**0**	**0**	**0**	**X**	**X**	**X**	**X**	**X**
2. Subtraction	**0**	**0**	**1**	**X**	**X**	**X**	**X**	**X**
3. Multiplication	**0**	**1**	**0**	**X**	**X**	**X**	**X**	**X**
4. Complement	**0**	**1**	**1**	**X**	**X**	**X**	**X**	**X**
5. Logical AND	**1**	**0**	**0**	**X**	**X**	**X**	**X**	**X**
6. Logical OR	**1**	**0**	**1**	**X**	**X**	**X**	**X**	**X**
7. Logical NOT	**1**	**1**	**0**	**X**	**X**	**X**	**X**	**X**
8. Logical XOR	**1**	**1**	**1**	**X**	**X**	**X**	**X**	**X**

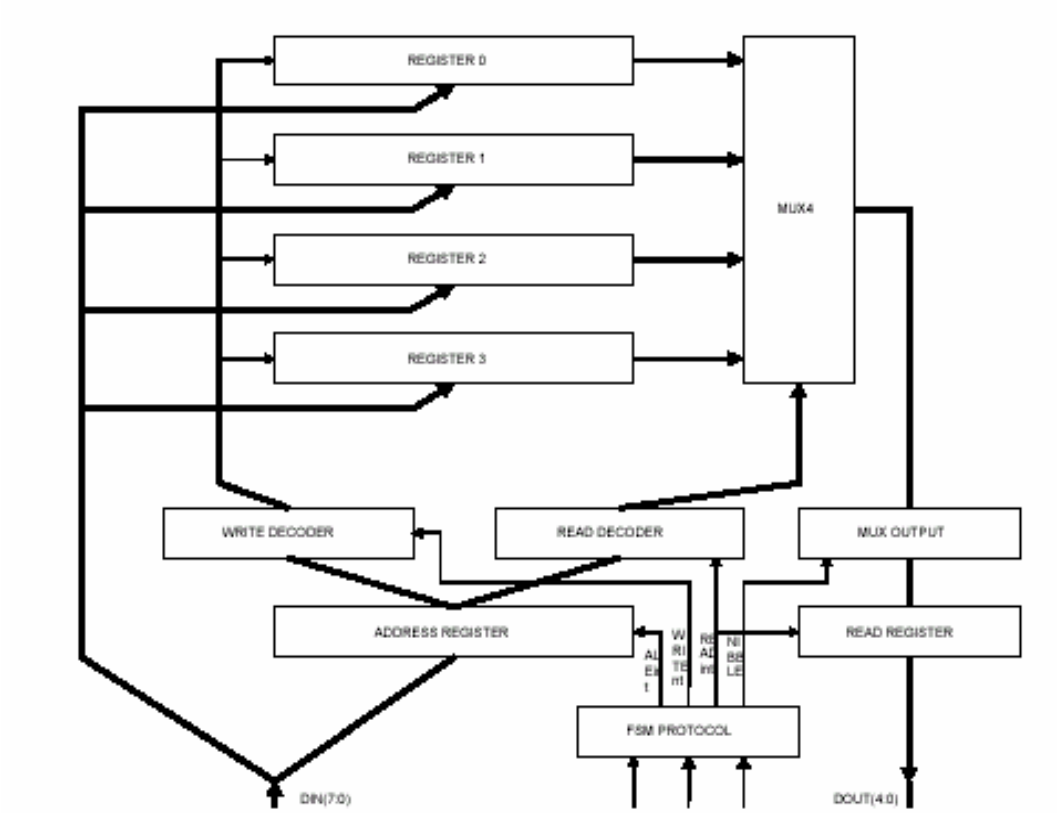

Figure. 3.2 Block diagram representation of the Bidirectional Interface Unit

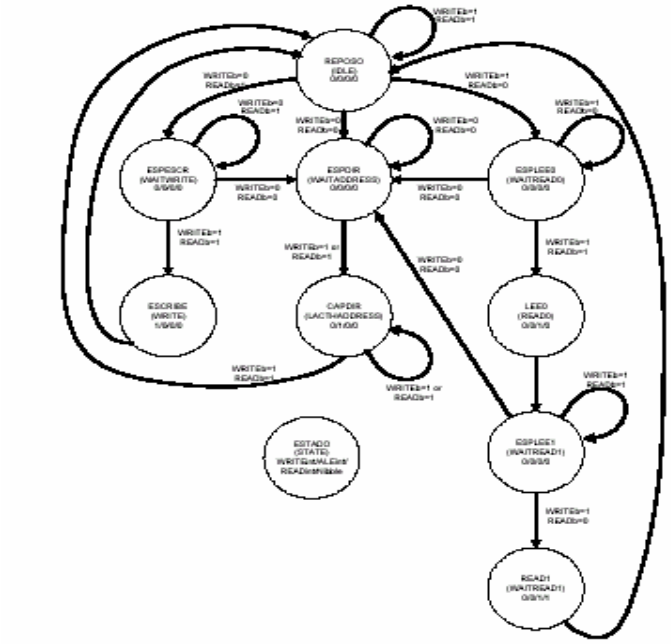

Figure. 3.3 State transition diagram for the FSM protocol unit

3.4 Process of accessing the RR-4 Coprocessor from PC

The process by which the functions of the reconfigurable coprocessor implemented on a Xilinx FPGA chip is controlled, can be demonstrated through the following steps. For convenience, we have chosen below an example of an arithmetic operation when we wish to execute a multiplication between to operands *e.g.* 21H and 89H.

STEP 1: The instruction code (in 8 bit hexadecimal form) is sent to the general purpose register *R0* of the coprocessor using the interface routine written in C++ (shown in Fig. 3.4).

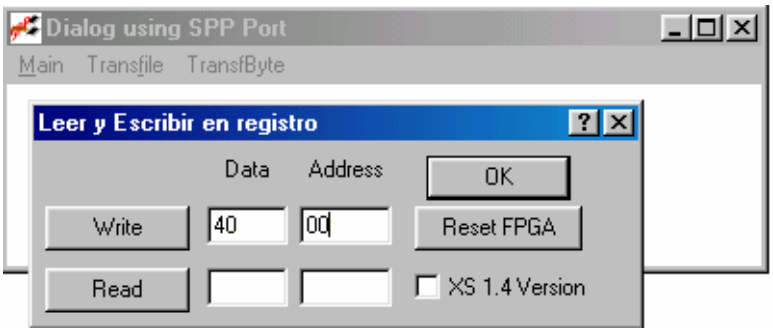

Figure 3.4 The instruction code for multiplication operation (40H) is sent to the register R0 (address: 00H)

STEP 2: The first operand (say, 21H) is sent to the general purpose register *R1* of the coprocessor.

STEP 3: The second operand (*i.e.* 89H) is sent to the register *R2* of the coprocessor.

STEP 4: As soon as the coprocessor receives the instruction and its required operands, it starts its internal operations (not observable from the user end *i.e.* the PC) and after performing all the relevant operations, it stores the final output in the registers *R1* and *R2* of the coprocessor.

STEP 5: The output is read from the two registers (*R1* and *R2*) in the coprocessor in succession one nibble per clock cycle and displayed in the dialog box shown above.

STEP 6: Finally, the instruction and its operands are cleared from the coprocessor by pressing the "Reset FPGA" button as shown in the dialog box in Fig 3.4.

Chapter 4

Conclusion

In this project, we have first designed a 8 bit arithmetic co-processor with the help of redundant radix-4 arithmetic number system. This co-processor is supposed to work on a parallel quarternary multiplication algorithm which is the fastest known in the domain of parallel computing [1]. The co-processor is capable of performing four arithmetic operations viz. addition, subtraction, multiplication and complementation, and four logical operations viz. logical AND, logical OR, logical NOT and logical XOR. That is to say, it is capable of performing eight operations having distinct instructions, the control words of which are explained in section 3.3.

The redundant radix-4 arithmetic coprocessor can be easily extended to perform other arithmetic operations. Since division can be performed by repeated multiplication, the division operation can thus be achieved in $O(log^2 m)$ time using the multiplication algorithm in [1]. New logical operations can also be suitably derived. For example, the shift Left, shift right operations can be done by shifting the digit columns accordingly.

We have implemented the co-processor on an FPGA. The target architecture is Xilinx Corporation's XC4000E device. The interfacing of the co-processor with a PC is executed by running a C++ program thereby controlling the input-output operations from the user domain.

Future work may include extension of this arithmetic co-processor to handle floating-point multiplication, implementation of other arithmetic and logical operations like division, shift left/right and evaluation of special functions like trigonometric functions (*sine, cosine, tangent*) and their inverses, Fourier transforms, FFT, DCT, matrix operations, etc. Because of the ease of VLSI implementation and fast, parallel operations, the RR-4 number system is highly suitable for such computation-intensive application areas like graphics, image processing, signal processing and weather forecasting.

Bibliography

[1] M. De and B. P. Sinha, "Fast Parallel multiplication using redundant quarternary number system", Parallel Processing Letters, Vol. 7, pp. 13-23 1997.

[2] S. Nakamura, "Algorithms for iterative array multiplication", IEEE Trans. Comput., Vol.35, pp.713-719, 1986.

[3] N. Takagi, H. Yassura, S Yajima, "High Speed VLSI multiplication algorithm with a redundnt binary addition tree" IEEE Trans. Comput., Vol. 34, pp. 789-796,1985.

[4] B. P. Sinha and P. K. Srimani, "Fast parallel algorithms for binary multiplication and their implementation on systolic architectures", IEEE Trans. Comput.,Vol. 38, pp. 424-431, 1989.

[5] M. De and B. P. Sinha, "Fast parallel algorithm for ternary multiplication using multivalued I^2L technology", IEEE Trans. Comput.,Vol. 43, pp. 603-607,1994.

[6] R. P. Brent and H.T. Kung, "A regular layout for parallel adders", IEEE Trans. Comput., Vol. 31, pp. 260-264,1982.

[7] K. Mehlhorn and F. P. Preparata, "Area-time optimal VLSI integer multiplier with minimum computation time", Information and Control, Vol. 58, pp. 137-156, 1983.

[8] A. Karatsuba and Y. Ofman, "Multiplication of multi-digit numbers on automata," Soviet Physics Doclady , Vol. 7, pp-595-596, 1963.

[9] N. Takagi and S. Yajima, "Modular multiplication hardware algorithms with a redundant representation and their application to RSA cryptosystem," IEEE Trans. Comput., Vol. 41, pp. 887-891,1992.

[10] A. Avizienis, "Signed-digit number representation for fast parallel arithmetic," IRE Trans. Electron. Comput., Vol. EC-10, pp. 389-400, 1961.

[11] P. J. Ashenden, The designer's guide to VHDL. San Francisco, California: Morgan Kaufman Publishers Inc. 1996.

[12] V. Vetz, J. Rose, A Marquardt, Architecture and CAD for deep-submicron FPGAs. USA: Kluwer Academic Publishers, 1999.

[13] Z. Salcic, VHDL and FPLDs in digital systems design, prototyping and customisation. USA : Kluwer Academic Publishers, 1998.

[14] Xilinx Data Book, 2000.

[15] J. P. Hayes, “Computer Architecture and Organisation”, McGraw Hill Publishing Company.

[16] William Stallings, “Computer Organization”, Prentice Hall of India Ltd.

[17] M. Morris Mano, “Digital Logic and Computer Design”, Prentice Hall of India Ltd.

[18] Pucknell and Eshraghian, “Basic VLSI Design”, Prentice Hall of India Ltd.

[19] **“**Design and Implementation of a RAM on the XS40 Board and the Bidirectional Interface with a PC” by Miguel Ángel Aguirre Echánove, Jon N. Tombs *et. al.* of University of Sevilla, Spain.

[20] K. Sinha, S. Roy and S. Sur-Kolay, "Design of a Redundant Radix 4 Arithmetic Coprocessor using Field-Programmable Gate Arrays", B. Tech. Thesis, Kalyani Government Engineering College, University of Kalyani, May 2001.