# FPGA based implementation of 8-bit ALU of a RISC processor using Booth algorithm written in VHDL language

Paresh Kumar Pasayat, Manoranjan Pradhan, Bhupesh Kumar Pasayat

**Abstract:**

*This paper explains the design and implementation of 8-bit ALU (arithmetic and logic unit) of a RISC processor that is implemented in Xilinx FPGA (Spartan-II) device. The ALU takes two 8-bits numbers and performs different arithmetic and logic operations like addition, subtraction, multiplication, division, logical AND, OR, NAND, NOR, NOT, XOR, XNOR, INCREMENT, DECREMENT, ROTATE LEFT AND ROTATE RIGHT. The main focus of concern in this ALU is the multiplication operation using booth algorithm and bit-pair recording technique which increases the speed of multiplication operation. There can be numbers of ALU operations based on the values of the select lines. The top level design consists of arithmetic unit and logic unit which is implemented by using structural model. The different design steps such as VHDL source code, synthesis, simulation and the implementation for 8-bit ALU of the processor are presented in this thesis. The synthesis, simulation and implementation of 8-bit ALU of the processor are performed by using Xilinx software, Xilinx ISE in-bulit simulator and SPARTAN-II FPGA device (Xc2s200-6pq208).*

*Index Terms- ALU (arithmetic and logic unit), FPGA (Field programmable gate arrays), VHDL (VHSIC HDL or Very High speed Integrated Circuit Hardware Description Language), RTL (Register transfer level).*

## 1. INTRODUCTION

The arithmetic and logic unit is very important block of the processor. The VHDL(very high speed integrated circuit hardware description language) is used to design ALU of the processor. The ALU is implemented in SPARTAN-II FPGA device.The processor requires simple instruction sets, less addressing modes, and supports eight numbers of arithmetic and logical operations. The algorithm state model approach is used to represent around fifteen numbers of different states of controller of processor.

The processor supports simple addressing modes like register addressing, register indirect addressing, immediate addressing etc. The processor is of register oriented architecture. It involves mainly register data transfer operations. There are less memory accesses to fetch opcodes and data from memory. So the speed and performance are comparatively higher.

The paper is organized as follows: In section-2, FPGA based design method is discussed. Section-3 describes ALU architecture. The synthesis and simulation results are presented in section-4. Section-5

and Section-6 are the conclusion and references respectively.

## 2. DESIGN METHOD

In classical method of design, a PCB(printed circuit board) contains many number of chips with other components. Development of these products starts with the definition of the overall structure. After that the required integrated circuit chips are selected followed by the PCBs that house and connect the chips together are designed. Since the complexity of circuits implemented on individual chips and on the circuit boards is usually very high, it is very much essential to make use of Xilinx software.

The software performs the following tasks: As the size and complexity of digital systems increases, more and more CAD(computer aided design) tools are being introduced in to the hardware design process. Design automation tools can help designer with design entry, hardware generation, and verification and design management. VHDL (very high speed integrated circuit hardware description language) is used to describe hardware for the purpose of simulation, modeling testing, design and documentation of processor. FPGA (field programmable gate arrays) provide the benefit of high integration levels and can be designed and verified quickly. FPGA devices are software configured and field programmable. Hence, there is a significant cost savings in design and productions.

Design entry of the ALU of the processor is carried out by using VHDL code. Initial synthesis generates an initial circuit, based on data entered during the design entry stage. Functional simulation is done to verify the functionality of the circuit, based on inputs provided by the designer. Logic synthesis and optimization uses optimization techniques to derive optimized circuits. Physical design determines how to implement the optimized circuit in a FPGA chip .Timing simulation determines the propagation delays that are expected in the implemented circuit. Chip configuration configures the actual chip to realize the designed circuit.

## 3. ARCHITECTURE

The proposed 16-bit ALU of the RISC processor consists of one arithmetic unit and logic unit.Using the concept of regularity, the arithmetic unit is divided into different blocks which perform the operations such as: addition, subtraction, multiplication, division. But, the main focus of concern in this arithmetic unit is the multiplier unit which has been implemented using Booth algorithm and bit pair recording techniques. The reason for choosing this technique is to increase the speed of 16-bit ALU. Similarly, the logic unit is also divided into various blocks which perform the operations such as: AND, OR, NAND, NOR, NOT, XOR, XNOR, INCREMENT, DECREMENT, ROTATE LEFT AND ROTATE RIGHT.

## Multiplier Unit:

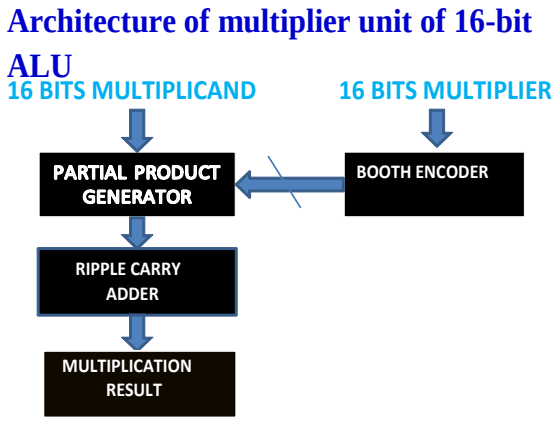


Figure 1: Architecture of multiplier unit of 16-bit ALU

The architecture for the multiplier unit is shown in Fig. 1.The 16-bit Multiplier unit consists of four components. They are the Booth Encoder, Partial Product Generator and the Adder units.

The booth encoder is the first block of the 8-bit multiplier unit. It takes only one input i.e. 8-bit multiplier. This block reduces the number of multiplier bits to half (i.e. 8-bit to 4-bit) and generates the booth code signals which is used to encode the 16-bit multiplicand into the partial products.The booth encoder takes three bits of 16-bit multiplier at a time and based on certain logic, it generates booth code signals.

The logic said above can best be explained with the help of a truth table which is given as follows:

| Y2 | Y1 | Y0 | Version of the multiplicand(M) selected at position i |
|---|---|---|---|
| 0 | 0 | 0 | 0 * M |
| 0 | 0 | 1 | +1 * M |
| 0 | 1 | 0 | +1 * M |
| 0 | 1 | 1 | +2 * M |
| 1 | 0 | 0 | -2 * M |
| 1 | 0 | 1 | -1* M |
| 1 | 1 | 0 | -1 * M |
| 1 | 1 | 1 | 0 * M |

**Table 1: Truth table for the Booth Encoder**

Now let us take 0 0 0 0 1 0 1 0 be the 8-bit multiplier.

It is assumes that there is a bit '0' at the right hand side of LSB of the multiplier.

0 0 0 0 1 0 1 0 0
0 +1 -1 -2

## Partial Product Generator:

The partial product generator is the second block of the 8-bit multiplier unit. The outputs of the booth encoder and the 8-bit multiplicand are fed to the inputs of the partial product generator. It consists of two stages: The booth encoder generates the booth code signals for encoding the 8-bit multiplicand into the partial products. Then, the partial product generator reads the booth code signals and encodes the multiplicand into the partial products.

The partial product generator takes booth code signals and the 8-bit multiplicand as the inputs to it and performs certain operations to produce the four partial products or summands.

The booth encoder generally takes the values which are 0, +1, +2, -1 and -2. Depending upon the values of booth code signals, the different operations are performed on the 8-bit multiplicand so as to produce the partial products.

LOGIC 1: If the value of the booth code signal is zero, then the value of the partial product is zero.

LOGIC 2: If the values of the booth code signal are +1 and +2, then the values of the partial products are same as the 8-bit multiplicand respectively.

LOGIC 3: If the values of the booth code signal are -1 and -2, then the values of the partial products are the 2's complement of the 8-bit multiplicand respectively.

**Explanation:**

Multiplier 0 0 0 0 1 0 1 0
0 +1 -1 -2
Multiplicand 0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0
* 0 0 0 0 1 0 1 0

0 0 0 0 0 0 1 0

0 +1 -1 -2

---

---------1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 pp1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 pp2
0 0 0 0 0 0 0 0 0 0 0 1 0 pp3
0 0 0 0 0 0 0 0 0 0 pp4

---

0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 = $(14)_{16}$

In the above example, the partial products are generated depending upon the value of the booth code signals. The symbols pp1, pp2, pp3 and pp4 represent the desired partial products or summands. Then the partial products are added to get the multiplication result.

If the addition of the partial product consists of any carry bits, then the carry bits are discarded. In the example given above, the carry bits are discarded. The result obtained after discarding the carry bits gives the desired multiplication result.

**Ripple Carry Adder:**

The ripple carry adder is the third block of the 8-bit multiplier unit. This unit takes the four partial products as the inputs and performs the addition operation on these partial products and gives the addition result which is the desired multiplication result.

In the ripple carry adder, the corresponding bits of the two operands are added and the sum and carry results are obtained. The carry of the current addition of individual bits are added to the addition of next most significant bits of the operands. The sum bits are grouped together to give final sum.

Suppose pp1, pp2, pp3 and pp4 are two partial products.Let us take,

pp1 = 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 and
pp2 = 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1

The two partial products are added in the following ways as shown below:

0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1

---

0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0

Similarly, the other two partial products pp3 and pp4 are added and the result is obtained. Finally, the two addition results are added using the ripple carry adder to give the final sum which is the desired multiplication result.

## Logic operations:

The logic operations are most useful and advantageous operations which describes the logical behavior of the logic circuit. The input given to the circuit is in the form of binary format (i.e. '0' or '1' or combination of '0' and '1'). The ALU can perform many types of logical operations such as logical and, or, nand, nor, not, xor, xnor, increment, decrement, rotate left and rotate right etc.

## Pin Assignments in FPGA device:

| Input Bit | FPGA Pin number | Output Bit | FPGA Pin number |
|---|---|---|---|
| num_A(0) | P5 | adder_result(0) | P46 |
| num_A(1) | P6 | adder_result(1) | P48 |
| num_A(2) | P7 | adder_result(2) | P57 |
| num_A(3) | P8 | adder_result(3) | P59 |
| num_A(4) | P9 | adder_result(4) | P61 |
| num_A(5) | P10 | adder_result(5) | P63 |
| num_A(6) | P22 | adder_result(6) | P68 |
| num_A(7) | P21 | adder_result(7) | P70 |
| num_B(0) | P20 | adder_result(8) | P47 |

**Table 2: Pin Numbers Assigned to input and output ports for Addition operation**

| Input Bit | FPGA Pin number | Output Bit | FPGA Pin number |
|---|---|---|---|
| num_A(0) | P5 | adder_result(0) | P46 |
| num_A(1) | P6 | adder_result(1) | P48 |
| num_A(2) | P7 | adder_result(2) | P57 |
| num_A(3) | P8 | adder_result(3) | P59 |
| num_A(4) | P9 | adder_result(4) | P61 |
| num_A(5) | P10 | adder_result(5) | P63 |
| num_A(6) | P22 | adder_result(6) | P68 |
| num_A(7) | P21 | adder_result(7) | P70 |
| num_B(0) | P20 | adder_result(8) | P47 |

**Table 3: Pin Numbers Assigned to input and output ports for Addition operation**

| Input Bit | FPGA Pin number | Output Bit | FPGA Pin number |
|---|---|---|---|
| num_A(0) | P5 | Sub_result(0) | P46 |
| num_A(1) | P6 | Sub_result (1) | P48 |
| num_A(2) | P7 | Sub_result (2) | P57 |
| num_A(3) | P8 | Sub_result (3) | P59 |
| num_A(4) | P9 | Sub_result (4) | P61 |
| num_A(5) | P10 | Sub_result (5) | P63 |

| | | | |
|---|---|---|---|
| num_A(6) | P22 | Sub_result (6) | P68 |
| num_A(7) | P21 | Sub_result (7) | P70 |

**Table 4: Pin Numbers Assigned to input and output ports for Subtraction operation**

| Input Bit | FPGA Pin number | Output Bit | FPGA Pin number |
|---|---|---|---|
| num_A(0) | P5 | product(0) | P46 |
| num_A(1) | P6 | product (1) | P48 |
| num_A(2) | P7 | product (2) | P57 |
| num_A(3) | P8 | product (3) | P59 |
| num_A(4) | P9 | product (4) | P61 |
| num_A(5) | P10 | product (5) | P63 |
| num_A(6) | P22 | product (6) | P68 |
| num_A(7) | P21 | product (7) | P70 |
| num_B(0) | P20 | product(8) | P47 |
| num_B(1) | P18 | product (9) | P49 |
| num_B(2) | P17 | product (10) | P58 |
| num_B(3) | P16 | product (11) | P60 |
| num_B(4) | P15 | product (12) | P62 |
| num_B(5) | P14 | product (13) | P67 |
| num_B(6) | P23 | product (14) | P69 |
| num_B(7) | P24 | product (15) | P71 |

**Table 5: Pin Numbers Assigned to input and output ports for Multiplication Operation**

| Input Bit | FPGA Pin number | Output Bit | FPGA Pin number |
|---|---|---|---|
| num_A(0) | P5 | quotient(0) | P46 |
| num_A(1) | P6 | quotient (1) | P48 |
| num_A(2) | P7 | quotient (2) | P57 |
| num_A(3) | P8 | quotient (3) | P59 |
| num_A(4) | P9 | quotient (4) | P61 |
| num_A(5) | P10 | quotient ( (5) | P63 |
| num_A(6) | P22 | quotient (6) | P68 |
| num_A(7) | P21 | quotient (7) | P70 |
| num_B(0) | P20 | remainder(0) | P47 |
| num_B(1) | P18 | remainder (1) | P49 |
| num_B(2) | P17 | remainder (2) | P58 |
| num_B(3) | P16 | remainder (3) | P60 |
| num_B(4) | P15 | remainder (4) | P62 |
| num_B(5) | P14 | remainder (5) | P67 |
| num_B(6) | P23 | remainder (6) | P69 |
| num_B(7) | P24 | remainder (7) | P71 |

**Table 6: Pin Numbers Assigned to input and output ports for Division operation**

| Input Bit | FPGA Pin number | Output Bit | FPGA Pin number |
|---|---|---|---|
| num_A(0) | P5 | log_result(0) | P46 |
| num_A(1) | P6 | log_result(1) | P48 |
| num_A(2) | P7 | log_result(2) | P57 |
| num_A(3) | P8 | log_result(3) | P59 |
| num_A(4) | P9 | log_result(4) | P61 |
| num_A(5) | P10 | log_result(5) | P63 |
| num_A(6) | P22 | log_result(6) | P68 |
| num_A(7) | P21 | log_result(7) | P70 |

**Table 7: Pin Numbers Assigned to input and output ports for Logic operation**

## 4. SIMULATION RESULTS AND COMPARISON:

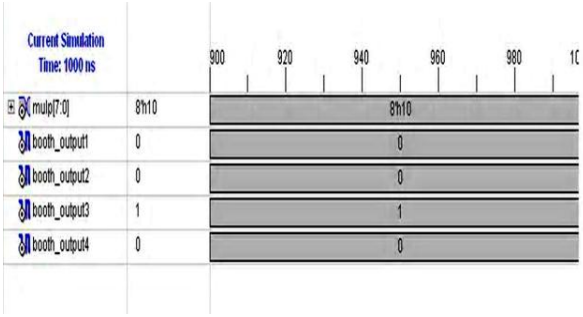


**Figure 2: Simulation Result for Booth Encoder**

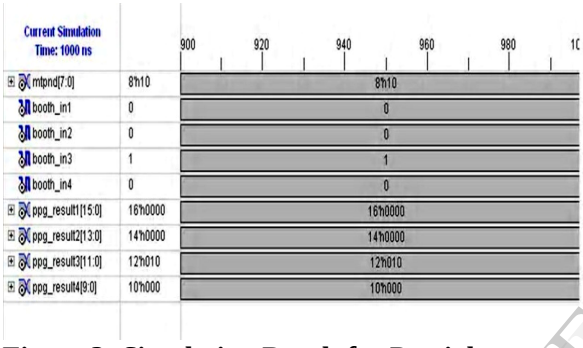


**Figure 3: Simulation Result for Partial Product Generator**

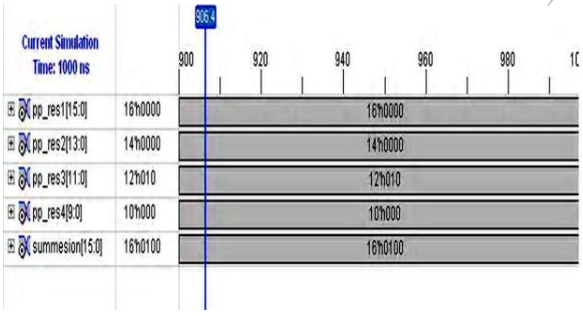


**Figure 4: Simulation Result for Partial Product Adder**

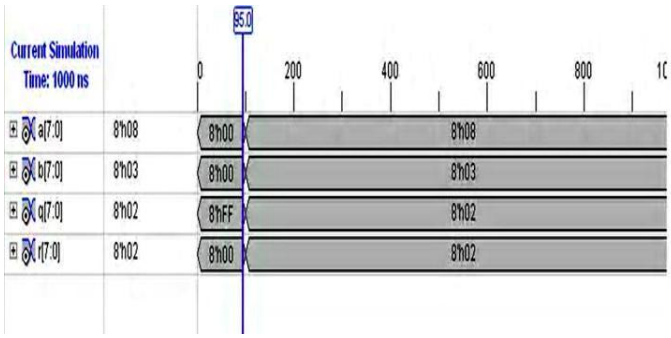


**Figure 5: Simulation Result for Restoring Division**

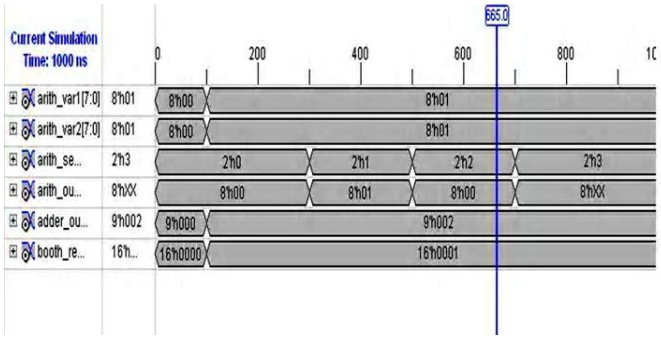


**Figure 6: Simulation Result for Arithmetic Unit**

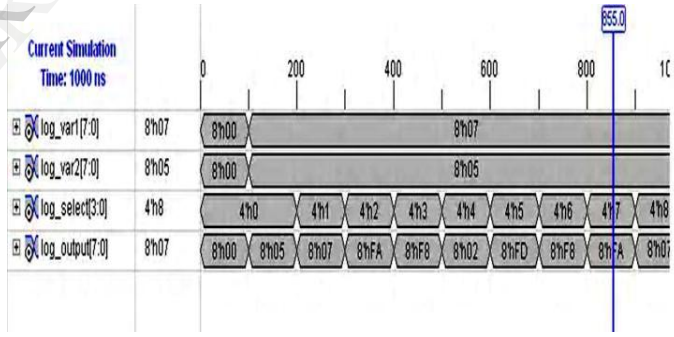


**Figure 7: Simulation Result for Logic Unit**

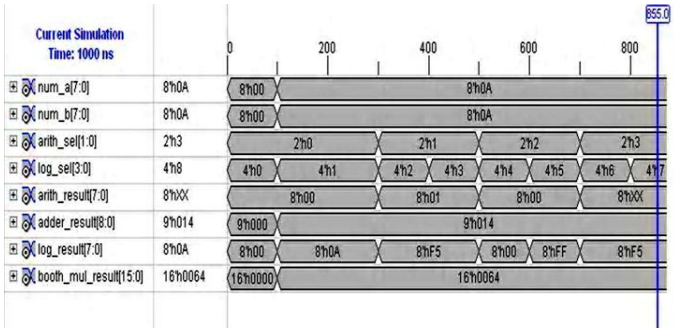


**Figure 8: Simulation Result for 8-Bit Arithmetic and Logic Unit**

The maximum combinational path delays of booth multiplier and the normal multiplier are tabulated as follows:

| Type of multiplier | Maximum combinational path delay (in ns) |
|---|---|
| Booth multiplier | 35.899 |
| Normal multiplier | 71.789 |

**Table 8: Comparison of maximum combinational path delays of booth multiplier and normal multiplier**

## 5. CONCLUSION:

This paper presented a new idea to design 8-bit ALU of a processor. This has been implemented in SPARTAN-II FPGA device. FPGA design offers greater design flexibility. The design is compact and is upgradable without any change in hardware. Here, the synthesis tool optimizes the design architecture of the FPGA for the ALU. Hence, additional features can be added to the existing design without any changes in hardware. The ALU is synthesized and simulated. The ALU of a processor is configured using each block designed.

The main advantage of the proposed 8-bit ALU design is the achievement of increase in the speed of ALU operation so that the time consumption will be reduced. This is because of the fact that the proposed ALU performs the multiplication operation using booth algorithm and bit-pair recording technique which reduces the number of multiplier bits to half (here 8-bit to 4-bit) so that the number of partial products generated will be reduced by a factor of 2 as compared to normal multiplication using general algorithm. Thus the time required to perform the addition of the partial products will be reduced to half so that the speed of multiplication will be enhanced. Therefore, the overall speed of the 8-bit ALU is increased.

## 6. REFERENCES:

[1] Mohamed Anane, Hamid Bessalah, Mohamed Issad, Nadjia Anane, and Hassen Salhi, "Higher Radix and Redundancy Factor for Floating Point SRT Division" IEEE Transactions on VLSI, Vol. 16, No. 6, June 2008.

[2] M. Mottaghi-Dastjerdi, A. Afzali-Kusha, and M. Pedram, "BZ-FAD: A Low-Power Low-Area Multiplier Based on IEEE Transactions on VLSI, 17, No. 2, February 2009.

[3] Roberr K. Monto ye, University of Illinois, Urbana-Champaign, IL, "THAM 11.1: Automatically Generated Area, Power and Delay Optimized ALUs", IEEE international conference on solid state circuits.

[4] Mauro Olivieri, "Design of Synchronous and Asynchronous Variable-Latency Pipelined Multipliers", IEEE Transaction on VLSI systems, pp-365-376, vol. 9, no.2,April, 2001.

[5] Douglas L. Perry. "VHDL Programming by Examples", TMH.

[6] Hamacher, Vranesic, and Zaky. Computer Organization, 5th edition, New York: McGraw-Hill Companies.