Implementation of 32-bit Multiplication Unit for a Out-of-Order RISC-V Processor utilizing Radix-4 Booth Algorithm and Wallace Tree

Enzo E. C. Ibiapina[1], **Benjamin S. Silva**[1], **Ivan S. Silva**[1]

[1]Departamento de Computação – Universidade Federal do Piauí (UFPI)
Teresina, PI – Brazil

{enzoeduardo_aluno,benjamin.santos,ivan}@ufpi.edu.br

Abstract. *Multiplication is a fundamental arithmetic operation critical to a broad spectrum of computational applications. Its implementation in hardware significantly influences overall system performance, given that multiplication inherently involves a series of successive addition operations, which can introduce potential bottlenecks in pipelined systems. This paper focuses on the design and implementation of an efficient 32-bit multiplier unit for educational purposes, within the context of a superscalar out-of-order RISC-V architecture, by utilizing and presenting standard techniques used on the industry in order to achieve fast multiplication results.*

1. Introduction

With the continuous advancements on fields such as computer graphics, artificial intelligence and neural networks, the multiplication operation is of utmost importance on the context of hardware implementation. The operation is considered to be a critical point on the pipeline of a processor, as it is composed by multiple addition operations, each one taking up to one clock cycle. In scenarios where multiple multiplication instructions are issued consecutively, instruction throughput and overall system performance can be significantly impacted. Therefore, accelerating the multiplication operation is essential to guarantee better performances during the runtime of the programs used in those areas of interest.

The RISC-V architecture represents an open and well-defined reduced instruction set that offers a standardized framework for developing hardware implementations [Foundation 2024]. Widely adopted in numerous deployed systems, the RISC-V instruction set architecture (ISA) includes the M extension module, which provides a contemporary and robust framework for implementing efficient multiplier units.

This paper aims to design a 32-bit integer multiplication unit for educational purposes, while discussing several techniques for speed optimization. It is organized as follows: on Section 2, the Booth algorithm for partial product generation is presented, as well as an chosen variation based on the number of bits. The Wallace Tree structure is explained on Section 3, a technique utilized to reduce the time spent on the successive additions of the partial products. The Section 4 shows the 4:2 Compressor, a component used on the previous explained structure, in order to perform the additions. The implementation of the techniques are commented on Section 5, while the details of the testing of the component are shown on Section 6. On Section 7, the conclusion of the paper is presented, as well as discussing future improvements to the component.

2. Booth Algorithm

The Booth Algorithm is a technique that generates partial products given two binary numbers of any sign combination [Booth 1951].

The algorithm operates as follows: a zero bit is appended to the right of the multiplier, and the multiplier is then divided into overlapping groups of two bits, starting from the right. In each group, the leftmost bit also serves as the rightmost bit of the preceding group. The summation of partial products is initialized to zero. Each unique combination of bits in a group corresponds to specific operations performed on the multiplicand, with the resulting value added to the cumulative sum. The final multiplication result is obtained through successive additions of these intermediate results. An example of this process is illustrated in Figure 1.

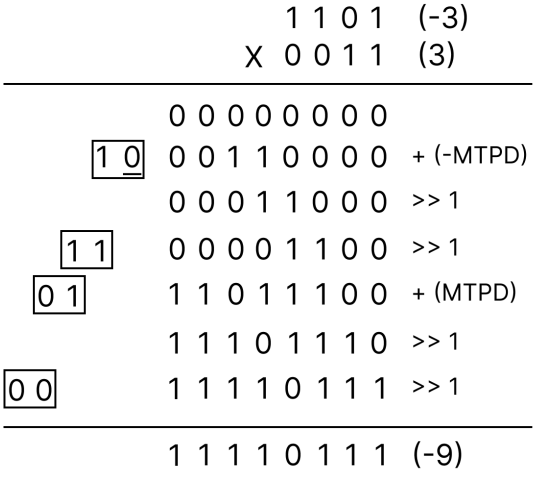

Figure 1. An example of the original Booth algorithm technique.

A modification of the algorithm can be used in order to encode more bits on the grouping step, effectively reducing the number of partial products generated. In general, a Radix-2^x modification groups x+1 bits, generating n/x partial products, where n is the number of bits of the multiplier. However, as x increases linearly, the cost and complexity of the logic increases exponentially. This trade-off must be considered when choosing the ideal version of the algorithm to a implementation project.

As this work utilizes 32-bit operands, the Radix-4 modification was chosen, where three bits of the multiplier are encoded per group, lowering the total number of products to $n/2$. The groups and its respective operations are presented on Table 1.

3. Wallace Tree

The Wallace Tree structure is a tree-based logic used to add partial products [Wallace 1964]. Each level of the tree divides its operands into blocks, which are then added together through subcomponents, such as pseudoadders or compressors, with the result of every block being stored on the next level. The logic continues up until the last level, where two operands are added in order to achieve the final result. This structure enables high-speed addition of partial products by leveraging parallel operations across

Table 1. Respective operations for every group on the Radix-4 Booth Algorithm.

Group	Operation
000	Add 0 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.
001	Add 1 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.
010	Add 1 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.
011	Add 2 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.
100	Add -2 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.
101	Add -1 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.
110	Add -1 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.
111	Add 0 * multiplicand to the current sum, and shift arithmetic the current sum one bit to the right.

multiple subcomponents, making it a scalable solution as the number of partial products increases.

An example of a Wallace Tree utilizing 4:2 compressors as subcomponents is illustrated on Figure 2. On the first level on the tree, the block of four products is added with the following logic: starting from the rightmost column of bits inside the block, if the column has less than three bits, the entire column is preserved onto the corresponding block on the next tree level. Otherwise, from the bottom, the last three bits of the column are fed into a 4:2 compressor. The resulting bit from the compressor, along with the unused fourth bit of the column, forms a column in the block on the next level. In this structure, every block with four products inside a tree level is reduced to two products on the next level. Since the next tree level has only two partial products, instead of dividing into blocks, an adder is used to perform a direct addition of the products, returning the final result of the multiplication.

4. 4:2 Compressor

The 4:2 Compressor is a combinational logic that produces two output bits with the addition of four input bits and a carry bit. It is utilized as the component that adds the partial products inside a group on a Wallace tree level. Different designs of such compressor have been developed [Yeh and Jen 2000, Zhang et al. 2023], with the standard composition of the component being two connected full adders [Edavoor et al. 2020]. The diagram for the exact 4:2 compressor is shown on Figure 3.

5. Implementation

The project utilized the Kanban methodology for the implementation. Standard techniques and components used in multipliers were studied, as well as the integration with the RISC-V's 20191213 specification of the M extension for the RV32I ISA [Foundation 2019]. The implementation uses VHDL-2002 for compilation.

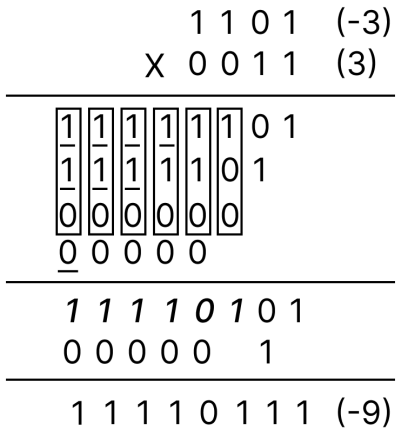

Figure 2. An example of a Wallace Tree structure on a usual multiplication, with 4:2 compressors as subcomponents.

The Booth Algorithm component receives the operands from the main component. The multiplicand is extended to 64 bits, while the multiplier receives an zero bit append before it gets analyzed, in order to produce the partial products. The component implements the Radix-4 modification, that generates 16 partial products for 32-bit operands.

The Wallace Tree component receives the 16 partial products, which composes the operands of the first tree level. The tree has four tree levels, with the number of operands reducing by half on every level. Furthermore, inside every tree level, the operands are divided into blocks of four, where they are added utilizing instances of 4:2 compressors. Each 4:2 compressor component receives four bits, A, B, C and D, as well as a $CarryIn$ bit. The logic follows the representation on Figure 3.

The compressor chain inside a block works as follows: starting from the right on every block, the first compressor on the pipeline has its D and $CarryIn$ bits set to zero, and each of the remaining input bits comes from one of the partial products inside that block. For every other compressor on the block, A, B and C continues to be each bit from one of the partial products, while D and $CarryIn$ are the output bits $CarryOut$ and $Carry$ from the previous component. The Sum bit from every compressor goes into the next tree level. The compressor chain extends through every column inside a block of a Wallace tree level, that has at least three bits. The interconnection between compressors on the same level is illustrated on Figure 4.

6. Analysis

For testing the multiplication unit, a testbench was designed, aiming to ensure the correctness of the results. A I/O file with 10000 different cases was created utilizing ChatGPT [OpenAI 2024], and it is read by the testbench in order to send the inputs to the component and compare the final output with the expected result.

An brief section of the testbench execution is shown on Figure 5. Utilizing the execution of the test A as an example, the inputs represented by A_i enter the internal pipeline of the multiplier. The implementation takes 6 clock cycles to return the expected result of the multiplication, as shown by the output represented by A_o. In comparison,

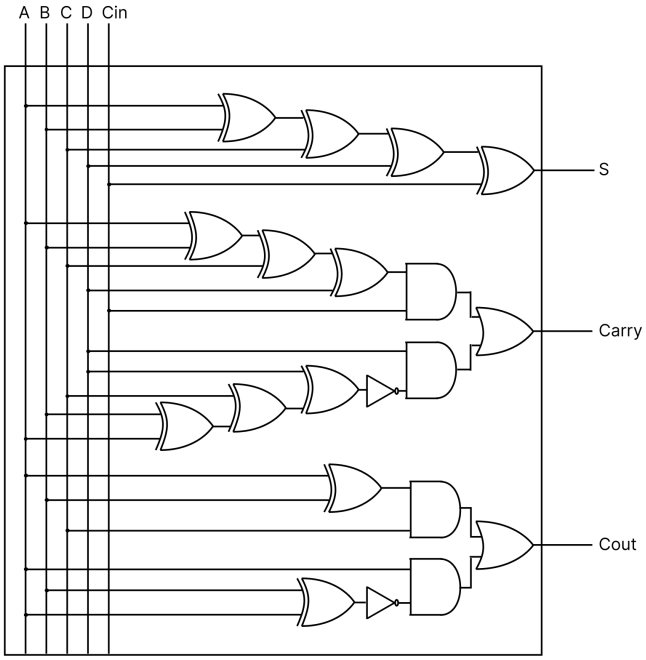

Figure 3. Exact 4:2 Compressor component.

the usual shift-and-add algorithm (also named sequential multiplication) [Koren 2002] can take at least 32 clock cycles to perform a operation, which shows the considerable increase in performance by implementing the algorithms described on this paper.

The implementation also presents a higher throughput: by utilizing registers between each stage of the process, the unit can store up to 5 operations in different stages of multiplication. The result is, to execute x upcoming instructions where the input is one instruction every cycle, the shift-and-add algorithm without pipeline can take at least $32x$ clock cycles to complete all the instructions, while the implementation takes $6 + x$ clock cycles, showing effective execution times on the context of superscalar processors.

7. Conclusion and Future works

In this paper, the design for an implementation of a integer multiplier was presented, while discussing several techniques for the different stages of the multiplication operation. The tests showed success for every input case, and an notable increase in speed performance of 80% on executing a instruction, in comparison with an shift-and-add multiplication solution, as well as an linear growth of clock cycles on executing multiple instructions in a row. However, the implementation realizes a trade-off to achieve these numbers, as it consumes a larger area of the processor.

In future works, the study aims to tackle floating-point multiplication, as well as implementing fixed-point multiplication by constants for further speed optimization.

References

Booth, A. D. (1951). A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, pages 236–240.

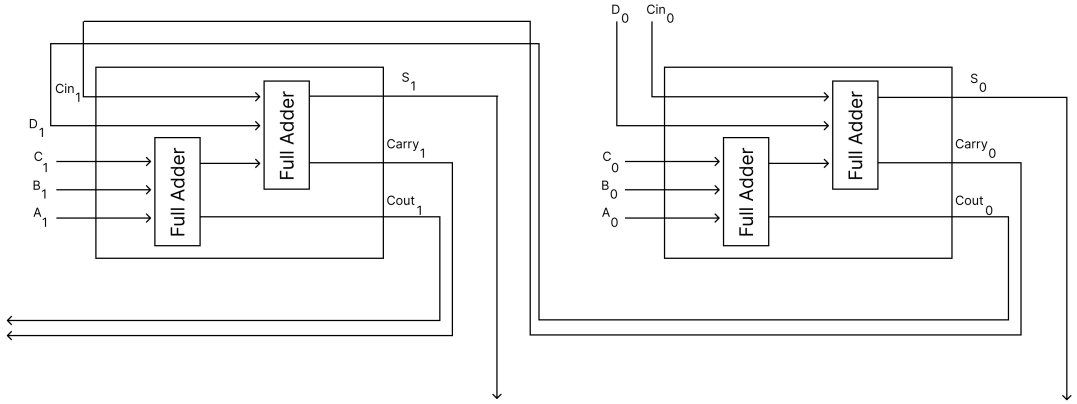

Figure 4. An example of the compressor chain, utilizing two 4:2 compressors.

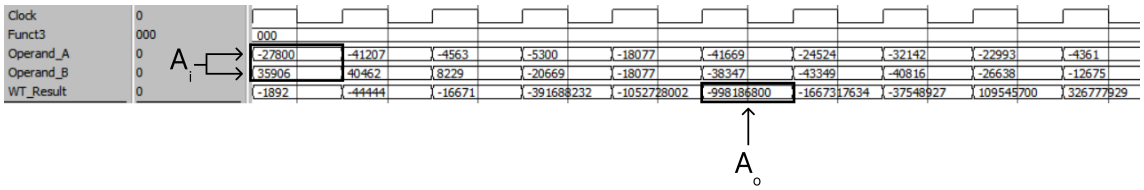

Figure 5. An excerpt of the execution of the testbench file.

Edavoor, P. J., Raveendran, S., and Rahulkar, A. D. (2020). Approximate multiplier design using novel dual-stage 4:2 compressors. *IEEE Access*, 8:48337–48351.

Foundation, R.-V. (2019). RISC-V Instruction Set Manual Volume I: User-Level ISA. `https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf`. Online; accessed in: 12-July-2024.

Foundation, R.-V. (2024). About RISC-V. `https://riscv.org/about/`. Online; accessed in: 12-July-2024.

Koren, I. (2002). *Computer Arithmetic Algorithms*. A K Peters/CRC Press, 2th edition.

OpenAI (2024). ChatGPT. `https://www.openai.com/research/chatgpt`. Online; accessed in: 12-July-2024.

Wallace, C. S. (1964). A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17.

Yeh, W.-C. and Jen, C.-W. (2000). High-speed booth encoded parallel multiplier design. *IEEE Transactions on Computers*, 49(7):692–701.

Zhang, M., Nishizawa, S., and Kimura, S. (2023). Area efficient approximate 4–2 compressor and probability-based error adjustment for approximate multiplier. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(5):1714–1718.