Design of ALU components

1 Purpose

The focus of this laboratory work is to show how the components (adders, multipliers, divisors) of the Arithmetic Logic Unit (ALU) can be designed and to highlight how different implementations can affect the performance and the usage of hardware resources.

2 Introduction

The Arithmetic and Logic Unit (ALU) is a digital circuit that performs arithmetic and logical components. This circuit is the basic block of the CPU and GPU and even for the simplest microprocessors.

An ALU must be able to calculate most operations. If there are more complex operations, the value of ALU is higher (i.e. more expensive), uses more space in the processor and dissipates more power. Thus, the design is a trade-off between computational power and area overhead/power consumption.

Almost all processor operations are done by one or more ALUs. An ALU loads the data from the input registers. The external control unit tells the ALU which operation (i.e. operation code) to perform on the data, and then the ALU stores the result into an output register. The control unit is responsible for moving the processed data between these registers, ALU and memory. Many of the ALU designs also contain a status register that indicates different cases, such as: carry-in or carry-out, overflow, divide-by-zero and so on.

3 Addition

The addition operation is the most frequently used arithmetic operation in any computer system. If there are more complex arithmetic functions, they are reduced to a series of additions. Note that by increasing the speed of addition, the speed of the ALU is also increased; but the speed and cost of adders are directly proportional to their complexity.

3.1 Full Adder

This circuit is the basic addition block, which adds three 1-bit inputs: 2 bits to be added (x_i and y_i) and a 1 bit carry-in (C_i). It generates 2 outputs: the sum bit (S_i) and the carry-out (C_{i+1}) bit. The figure below illustrates a full adder symbol.

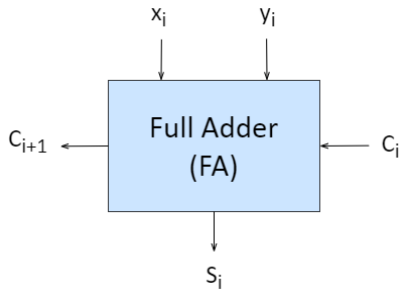

Figure 1: Full adder

x_i	y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 1: Full adder truth table

Given the truth table above, the Boolean expressions of the outputs (after reduction) are:

$$S_i = x_i \oplus y_i \oplus C_i$$
$$C_{i+1} = x_i \cdot y_i + (x_i + y_i) \cdot C_i$$

A **half adder** is basically a full adder without the carry input and generates a sum bit and a carry bit.

3.2 Ripple Carry Adder

This type of adders are implemented by several full adders in series, each carry output is connected to the input of the next one. This is a parallel adder and is used for adding n-bit number. It has the advantage of simplicity and low cost, but at the cost of reduced speed. The figure below illustrates the block diagram of a ripple carry adder for adding 4-bit numbers.

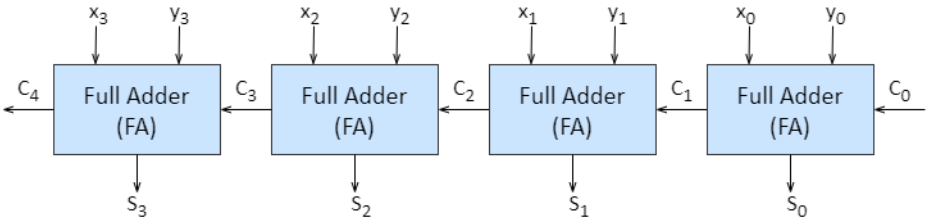

Figure 2: Ripple carry adder using 4 full adders for 4-bit numbers

3.3 Carry Lookahead Adder

In order to reduce the time required to form the carry signals, this type of adders use a separate block which calculates the carry output of each full adder. Thus, there is no "wait time" for carries to ripple from stage to stage (like in the ripple carry adder design). In the figure below, the block diagram of a carry lookahead adder that adds 4-bit integers is exposed.

Given the design above, two functions are used to *generate* (g) and *propagate* (p) the carry output.

$$g_i = x_i \cdot y_i$$
$$p_i = x_i + y_i$$

Using the 2 functions above, the output carry can be expressed as:

$$C_{i+1} = g_i + p_i \cdot C_i$$

Thus, each carry output can be formed using the g and p functions from the same stage and from the previous stages. The carry blocks from the picture above can be joined in a single carry lookahead generator and each carry output can be generated by a combinatorial circuit.

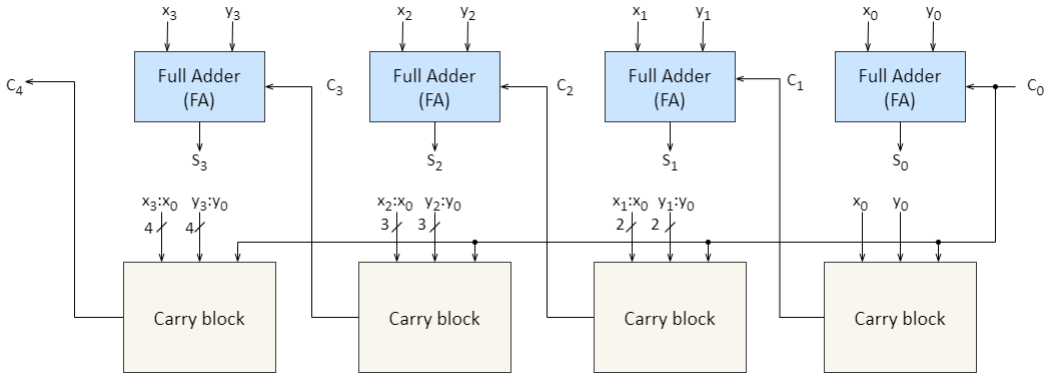

Figure 3: Carry lookahead adder design for adding 4-bit numbers

The main advantage of such a type of adder is that the output carries can be individually computed based on the input carry and the bits in the operand. Thus, the latency is significantly reduced, as the computation of the output carry at level $i+1$ does not require all the previously i carry outputs to be already computed, as it can be expressed only in terms of the inputs. An example for two 4-bit operands, $x(3..0)$ and $y(3..0)$, is given below:

$$C_1 = g_0 + p_0 \cdot C_0 = x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0$$

$$C_2 = g_1 + p_1 \cdot C_1 = x_1 \cdot y_1 + (x_1 + y_1) \cdot C_1 = x_1 \cdot y_1 + (x_1 + y_1) \cdot (x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0)$$

$$\begin{aligned} C_3 = g_2 + p_2 \cdot C_2 &= x_2 \cdot y_2 + (x_2 + y_2) \cdot C_2 = x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot C_1) \\ &= x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot (x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0)) \end{aligned}$$

$$\begin{aligned} C_4 = g_3 + p_3 \cdot C_3 &= x_3 \cdot y_3 + (x_3 + y_3) \cdot C_3 = x_3 \cdot y_3 + (x_3 + y_3) \cdot (x_2 \cdot y_2 + (x_2 + y_2) \cdot C_2) \\ &= x_3 \cdot y_3 + (x_3 + y_3) \cdot (x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot C_1)) \\ &= x_3 \cdot y_3 + (x_3 + y_3) \cdot (x_2 \cdot y_2 + (x_2 + y_2) \cdot (x_1 \cdot y_1 + (x_1 + y_1) \cdot (x_0 \cdot y_0 + (x_0 + y_0) \cdot C_0))) \end{aligned}$$

3.4 Other types of adders

Besides the adders described above, there are several types of adders that have advantages and disadvantages over regular designs and which are worth mentioning.

3.4.1 Carry Select Adder

This type of adder uses redundant hardware to speed up the addition process. It divides the adding operation into 2 parts: the high-order half and the low-order half. First, the high-order half of the sum is calculated for both possible input carries. When the carry from the low-order half of the sum is known, the proper high-order half can be selected. Note that there is the possibility to divide the adder into quarters, so even lower complexity. This is why the complexity of this type of adder overcomes the carry lookahead adder disadvantages.

3.4.2 Carry Save Adder

This type of adder is used when more than 2 numbers are to be added because it reduces the carry propagation time. The design consists of n independent full adders (for n-bit numbers), with the inputs being the n-bit numbers and the outputs are an n-bit sum word and an n-bit carry word. Basically, each full adder works independently from the others and the carry signals are not propagated between the individual full adders. To get the final result, both parts (sum and carry) must be added together using a normal type adder.

3.4.3 Serial Adder

This is the simplest type of adder because it uses a single full adder and a D-latch. Basically, it performs the addition step by step from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). The D-latch is used to propagate the carry of the previous sum to the next bits that are going to be added.

4 Multiplication

The multiplication of binary numbers is similar to that of decimal numbers. There are various methods of multiplying two n-bit numbers:

- Shift-and-Add Multiplication
- Booth's Technique
- Higher-Radix Multiplication
- Array Multiplier
- Wallace Tree

In the following subsections, only 2 out of the 5 methods will be explained in detail.

4.1 Shift-and-Add Multiplication

Shift-and-Add Multiplication is one of the basic and simplest method for adding 2 numbers.Basically, the whole idea is to add the multiplicand (let's say X) to itself for Y (multiplier) times. The algorithm is based on taking each digit of the multiplicand in turn and multiplying it by a single digit of the multiplier. Each intermediate product is placed in the appropriate positions, to the left of the earlier results. Finally, all the intermediate products are added together, to get the final result. The steps of the Shift-and-Add Multiplication algorithm are shown in Figure 4.

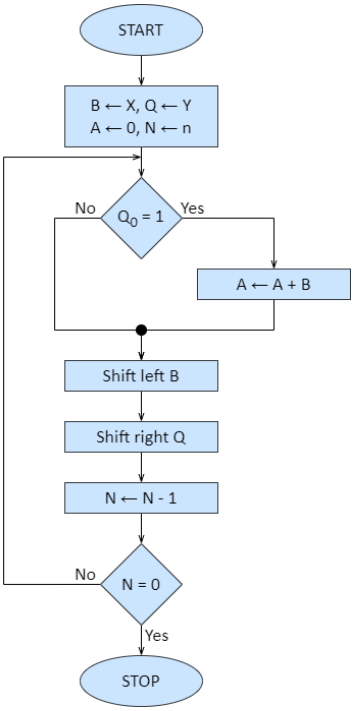

Figure 4: Shift-and-Add Multiplication Algorithm

The block design of the shift-and-add multiplication technique is illustrated in the Figure 5.

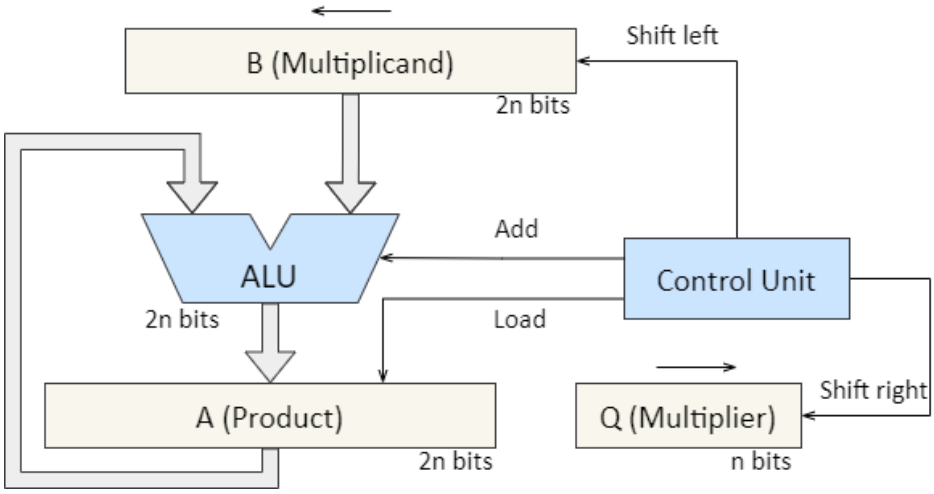

Figure 5: Block design of the shift-and-add multiplication technique

4.2 Wallace Tree Multiplication

The **Wallace Tree** technique is based on combining pairs of partial products with the help of multiple levels of Carry Save Adders:

- at each level of the tree, the numbers are grouped into three and are added together
- the levels continue until only 2 numbers are left to be added
- a Carry Propagate Adder is used to add the last 2 numbers and deliver the final result

This design reduces the number of terms to be added by a factor of 1.5, thus resulting in a total time of $O(\log_{1.5} n)$. Note that the preceding multipliers have a time of $O(n)$, where n is the number of bits for each number. Figure 6 shows the block design of a Wallace Tree for multiplying two 8-bit numbers.

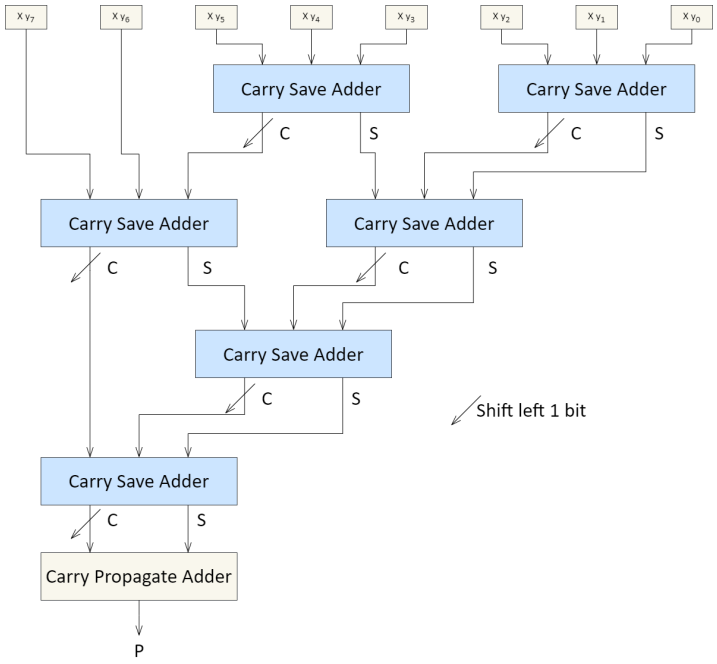

Figure 6: Block diagram of the Wallace Tree design for multiplying two 8-bit numbers

Another example for multiplying two 4-bit numbers using the Wallace Tree technique is shown in Figure 7.

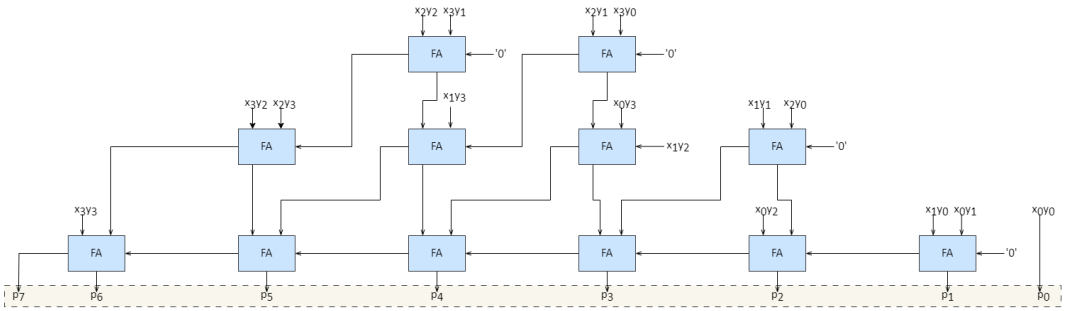

Figure 7: Wallace Tree Multiplier for multiplying two 4-bit numbers

The Wallace Tree method can be combined with other methods to further increase the speed. For example, the Booth technique can be used to produce the partial products, while a Wallace Tree adds the partial products.

5 Division

In any division operation, we have the first operand called *dividend* (X), the second operand called *divisor* (Y) and the results are the *quotient* (Q) and *remainder* (R). The mathematical expression is:

$$X = Q \cdot Y + R, R < Y$$

The algorithm for decimal division is explained below:

1: Choose a digit and subtract the product between this digit and the divisor from the partial remainder
2: If the result is smaller than the divisor, the digit was chosen correctly
3: Otherwise, choose another digit and repeat the subtraction

The binary division algorithm is based on repeated subtractions of the divisor Y from the partial remainder R, but are executed only if $Y \leq R$, which results in a quotient digit of 1 (otherwise it is 0).

All the steps of the binary division algorithm are shown in Figure 8.

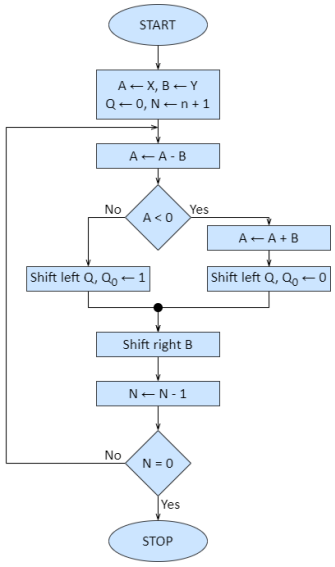

Figure 8: Restoring Division Algorithm

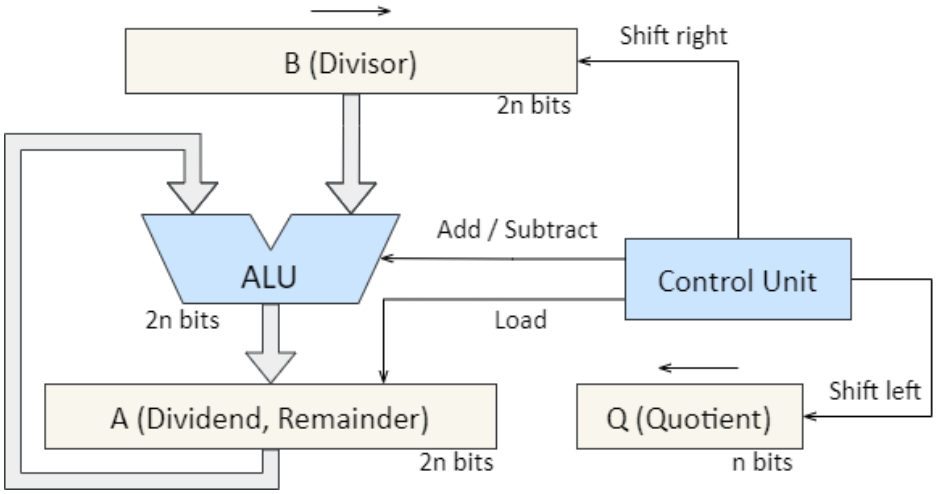

Figure 9: Basic block design of the division operation

The basic block design of the division operation is illustrated below in Figure 9.

Note that this block design can be improved in order to reduce the time and to simplify the hardware needed. For example, shifting the partial remainder to the left (instead of shifting the divisor to the right) produces the same alignment and simplifies the hardware of the ALU and the divisor register (n bits instead of $2n$). Another idea is based on the fact that the first step cannot generate a digit of 1 in the quotient, thus the order of the operations can be switched: first shift, then subtract (one iteration removed). Also, the size of the A register could be reduced to half and could be combined with the Q register. So, the bits of the dividend are shifted into the A register instead of shifting zeros, and both A and Q registers are shifted left together.

The improved version of the algorithm is shown in Figure 10.

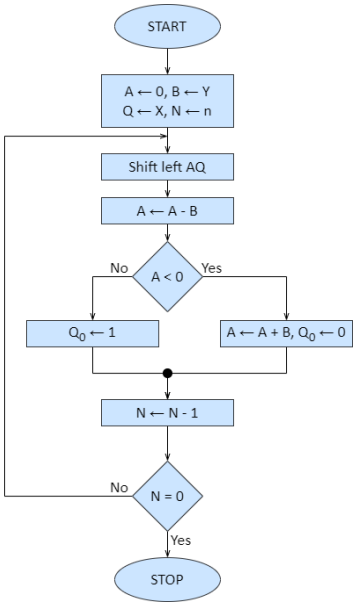

Figure 10: Improved Restoring Division Algorithm

Considering the above, the improved block design is in Figure 11.

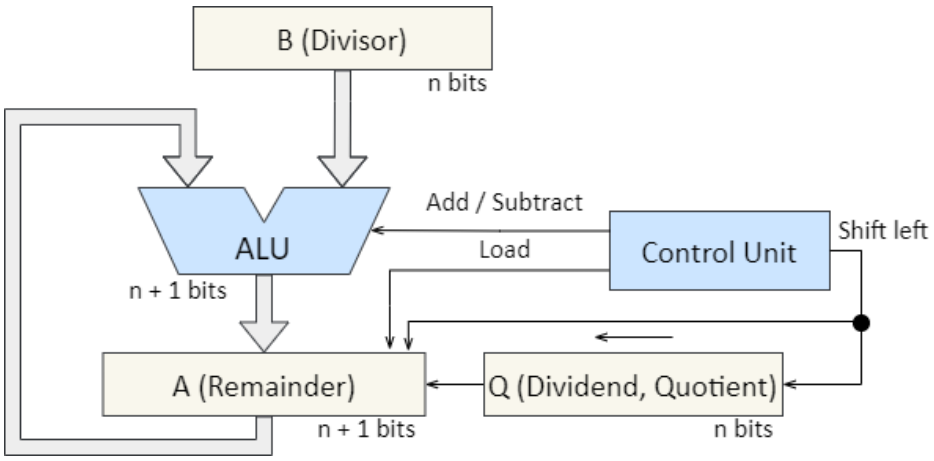

Figure 11: Improved block design of the division operation

6 Exercises

1. **Implement and simulate in VHDL a 4-bit counter (check the previous laboratory work).**
2. **Design and implement in VHDL the modified version of the carry lookahead adder for 4-bit numbers.**
   - Form the 4 carry outputs using the g and p functions (as previously shown)
   - VHDL implementation and simulation for two random 4-bit numbers
3. **Design and implement in VHDL Carry Save Adder for three 4-bit numbers.**
4. **Make a comparison chart that shows the advantages and disadvantages of each multiplying technique.**
5. **Implement the Wallace Tree for multiplying two 4-bit numbers depicted in Figure 7. Simulate the design for two random numbers.**

References

[1] Ripple-Carry Adder [accessed Oct. 2024], https://www.sciencedirect.com/topics/computer-science/ripple-carry-adder

[2] Carry Look-Ahead Adder [accessed Oct. 2024], https://www.geeksforgeeks.org/carry-look-ahead-adder/

[3] Arithmetic logic unit [accessed Oct. 2024], https://en.wikipedia.org/wiki/Arithmetic_logic_unit

[4] Wallace tree [accessed Oct. 2024], https://en.wikipedia.org/wiki/Wallace_tree

[5] Restoring Division Algorithm For Unsigned Integer [accessed Oct. 2024], https://www.geeksforgeeks.org/restoring-division-algorithm-unsigned-integer/