During all aspects of the design process, we have always aimed for higher throughput, rather than a smaller chip area. It was clear that multiplication had to be carried out by a dedicated hardware block in the ALU instead of software implementations. Most multiplication algorithms can be broken down into two main segments. First, partial products are generated, then adders sum up those to produce the result. The simplest way is to multiply each bit of the multiplicand with each bit of the multiplier. Digital data is in binary. Therefore, each partial product is either a copy of the multiplicand or it is 0. Using this method, multiplying an N bit number with another N bit number would result in N partial products. In our case, these 16 numbers must be shifted according to their weighting. Finally, they must be summed up to produce the product. Even with the use of varying sized carry-select adders, this would be a highly inefficient method. Instead, several algorithms are proposed which improve the speed of the computation. This can either be achieved by reducing the number of partial products generated (Booth’s algorithm) or by performing the addition of these products in parallel (Wallace trees). Radix 4 modified Booth’s algorithm cuts the number of partial products by half. Unfortunately, this method cannot be applied to unsigned numbers as it exploits unique features of the 2’s complement format. This is not optimal since the linear congruential generator uses strictly unsigned integers.

The final solution came with Vedic Mathematics. The generation of partial products and their summation is carried out by vertical and crosswise multiplication, then addition. Application of this concept to a 2-bit multiplier is illustrated in figure 1. The least significant bit of the product is obtained by multiplying together the least significant bits of each factor (). The next bit of the product is the summation of two crosswise multiplication (). Since the operation is adding two 1-bit numbers, overflow can occur. The carry bit must be propagated to the next step. The following most significant bit is calculated by multiplying the most significant bit of the factors and adding that carry from the previous step (. Finally, the most significant bit of the product is the carry from the previous step. The 2x2 multiplier block is implemented with 4 AND gates and 2 half-adders (figure 2). 4x4 blocks can be constructed with the use of four 2x2 blocks and adders (figure 3). The use of ripple adders is inefficient, to improve the speed of the calculation, carry-save adders can be used. They can add 3 numbers while producing 2 outputs which can later be summed by a ripple adder (in figure 3, the ripple adder is inside the carry-save adder block). This method is significantly faster than using 2 ripple adders sequentially. To save time and to ensure the best solution we have relied on Quartus’s synthesiser. In Verilog the “+” operator chooses the most appropriate adder module. Using the same method, we have constructed an 8x8 and later a 16x16 multiplication module.

Regardless of adder types used, the module is an extensive combinational logic. Ideally, primitive gates are constructed from transistors in a way, that in their stable states, their power consumption is negligible. However, when the gates are transitioning from one state to another, current flows through, dissipating heat. Even when the CPU does not perform a multiplication instruction the combinational logic will drain enormous chunks of power. To mitigate this, we have added an enable input to the 16x16 multiplication block that is AND-ed with the multiplicand and the multiplier. When executing a series of other instructions, the enable input will be low, forcing each factor to be 0, preventing the gates to enter transitional states. Ultimately reducing the power consumption of the CPU.

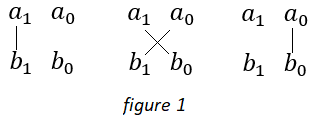
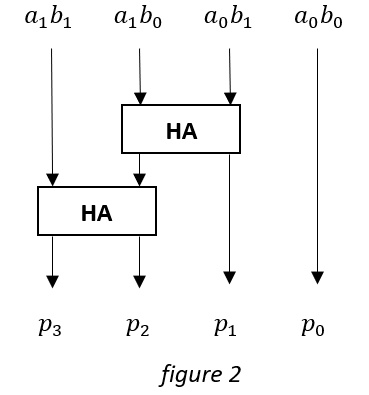
The multiplication module is expected to have a long latency. Results of a time analysis on the block showed, that a maximum operational clock frequency is 65 MHz. Although it is designed to perform the operation in 1 clock cycle, we have decided to make the module sequential. A set of registers was connected directly to the output of each 4x4 block. The decoder and the ALU both have combinational logic that will further increase the delay of a multiplication instruction. The purpose of placing the registers specifically here is to roughly half this latency in the CPU. The sequential version allowed for a clock frequency of 106 MHz in the 16x16 multiplication module. Another set of registers could be added to the output of each 8x8 block, further increasing the clock rate to 186 MHz. However, due to other paths in the CPU, which are much slower, this appeared to be unnecessary.

It is clear, that the product of N-bit numbers will always fit into 2N bits. In our case, the product will always be a 32-bit number. No need to concern about overflow. The least significant 16 bit of the result will be stored in the destination register; the most significant half will be stored in the consequent register. Further arithmetic on the product needs to be performed on both registers.

Sources:

<https://www.researchgate.net/figure/2x2-Reversible-Multiplier-Module_fig14_281267232>

<https://www.researchgate.net/publication/258029616_Hardware_Implementation_of_1616_bit_Multiplier_and_Square_using_Vedic_Mathematics>



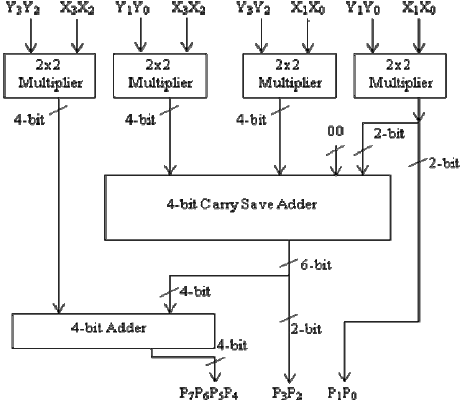


figure 3