CHAPTER-4

PROPOSED SYSTEM

The Karatsuba formula is used to speed-up the multiplication of large numbers by splitting the operands in two parts of equal length. The proposed configurable KMA is shown in Figure4.1.

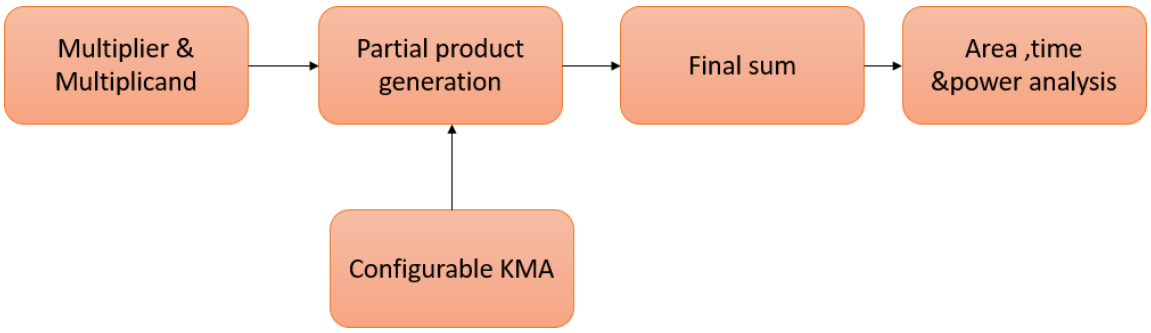

Figure4.1 : Proposed multiplier

The standard procedure for multiplication of two n-digit numbers requires a number of elementary operations proportional to n2, or o(n2) in big-O notation. Andrey Kolmogorov conjectured that the classical algorithm was asymptotically optimal, meaning that any algorithm for that task wouldrequire n2 elementary operations.

In 1960, Kolmogorov organized a seminar on mathematical problemsin cybernetics at the Moscow State University, where he stated the Ωn2conjecture and other problems in the complexity of computation. Within a week, Karatsuba, then a 23-year-old student, found an algorithm (later it was called "divide and conquer") that multiplies two n-digit numbers in o(log2) elementary steps, thus disproving the conjecture. Kolmogorov was very agitated about the discovery; he

communicated it at the next meeting of the seminar, which was then terminated. Kolmogorov did some lectures on the Karatsuba result at the conferences all over the world (see, for example, "Proceedings of the international congress of mathematicians 1962", pp. 351–356, and also "6 Lectures delivered at the International Congress of Mathematicians in Stockholm, 1962") and published the method in 1962, in the Proceedings of the USSR Academy of Sciences. The article had been written by Kolmogorov and contained two results on multiplication, Karatsuba's algorithm and a separate result by Yuri Ofman it listed "A. Karatsuba and Yu. Ofman" as the authors. Karatsuba only became aware of the paper when he received the reprints from the publisher.

Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can write the two given numbers as

$$x = x_1 B^m + x_0,$$
$$y = y_1 B^m + y_0,$$

where x_0 and y_0 are less than B^m. The product is then

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0),$$
$$xy = z_2 B^{2m} + z_1 B^m + z_0,$$

These formulae require four multiplications and were known to Charles Babbage.Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z0 and z2 as before one can calculate

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0,$$

which holds, since

$$z_1 = x_1 y_0 + x_0 y_1,$$
$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0.$$

An issue that occurs, however, when computing z1 is that the above computation of (x1+x0) and (y1+y0) may result in overflow which require a multiplier having one extra bit. This can be avoided by noting that

$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0,$$
$$z_1 = (x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0) - x_1 y_1 - x_0 y_0,$$
$$z_1 = (-x_1 y_1 + x_1 y_0 + x_0 y_1 - x_0 y_0) + x_1 y_1 + x_0 y_0,$$
$$z_1 = -(x_1 y_1 - x_1 y_0 - x_0 y_1 + x_0 y_0) + x_1 y_1 + x_0 y_0,$$
$$z_1 = -(x_1 - x_0)(y_1 - y_0) + x_1 y_1 + x_0 y_0,$$
$$z_1 = (x_0 - x_1)(y_1 - y_0) + x_1 y_1 + x_0 y_0.$$

This computation of (x0-x1) and (y1-y0) will produce a result in the range of -bm<result<bm. This method may produce negative numbers, which require one extra bit to encode signedness, and would still require one extra bit for the multiplier.

To compute the product of 12345 and 6789, where B = 10, choose m = 3. Then we decompose the input operands using the resulting base (Bm = 1000), as:

12345 = 12 · 1000 + 345

6789 = 6 · 1000 + 789

Only three multiplications, which operate on smaller integers, are used to compute three partial results:

$z2 = 12 \times 6 = 72$

$z0 = 345 \times 789 = 272205$

$z1 = (12 + 345) \times (6 + 789) - z2 - z0 = 357 \times 795 - 72 - 272205 = 283815 - 72 - 272205 = 11538$

We get the result by just adding these three partial results, shifted accordingly (and then taking carries into account by decomposing these three inputs in base 1000 like for the input operands):

$result = z2 \cdot (Bm)2 + z1 \cdot (Bm)1 + z0 \cdot (Bm)0,$

$result = 72 \cdot 10002 + 11538 \cdot 1000 + 272205 = 83810205.$

Note that the intermediate third multiplication operates on an input domain which is less than two times larger than for the two first multiplications, its output domain is less than four times larger, and base-1000carries computed from the first two multiplications must be taken into account when computing these two subtractions.

4.1 Recursive application

If n is four or more, the three multiplications in Karatsuba's basic step involve operands with fewer than n digits. Therefore, those products can be computed by recursive calls of the Karatsuba algorithm. The recursion can be applied until the numbers are so small that they can (or must) be computed directly.

In a computer with a full 32-bit by 32-bit multiplier, for example, one could choose B = 231 = 2147483648, and store each digit as a separate 32-bit binary word. Then the sums x1 + x0 and y1 + y0 will not need an extra binary word for storing the carry-over digit (as in carry-save adder), and the Karatsuba recursion can be applied until the numbers to multiply are only one-digit long.

4.2 Pseudocode

```
procedure karatsuba(num1, num2)
  if (num1 < 10) or (num2 < 10)
  return num1*num2
  /* calculates the size of the numbers */
  m = max(size_base10(num1), size_base10(num2))
  m2 = floor(m/2)
  /* split the digit sequences in the middle */
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)
  /* 3 calls made to numbers approximately half the size */
  z0 = karatsuba(low1, low2)
  z1 = karatsuba((low1 + high1), (low2 + high2))
  z2 = karatsuba(high1, high2)
  return (z2 * 10 ^ (m2 * 2)) + ((z1 - z2 - z0) * 10 ^ m2) + z0
```

4.3 Modified Multiplier

Common subexpression elimination (CSE) is a compiler optimization that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value

The possibility to perform CSE is based on available expression analysis (a data flow analysis). An expression b*c is available at a point p in a program if: every path from the initial node to p evaluates b*c before reaching p, and there are no assignments to b or c after the evaluation but before p.

The cost/benefit analysis performed by an optimizer will calculate whether the costof the store to tmp is less than the cost of the multiplication; in practice other factors such as which values are held in which registers are also significant.

Compiler writers distinguish two kinds of CSE:

local common subexpression elimination works within a single basic block

global common subexpression elimination works on an entire procedure,

Both kinds rely on data flow analysis of which expressions are available at which points in a program.

The benefits of performing CSE are great enough that it is a commonly used optimization.

In simple cases like in the example above, programmers may manually eliminate the duplicate expressions while writing the code. The greatest source of CSEs are intermediate code sequences generated by the compiler, such as for array indexing

calculations, where it is not possible for the developer to manually intervene. In some cases language features may create many duplicate expressions. For instance, Cmacros, where macro expansions may result in common subexpressions not apparent in the original source code.

Compilers need to be judicious about the number of temporaries created to hold values. An excessive number of temporary values creates register pressure possibly resulting in spilling registers to memory, which may take longer than simply recomputing an arithmetic result when it is needed.