Solomon I. Khmelnik

Computer Arithmetic of Geometrical Figures

Algorithms and Hardware Design

First Edition 2004
Second Edition 2013

Russia Israel Canada

ISBN 978-1-4116-3184-7

Technical Editor, Cover Designer – Inna S. Doubson

Published by "MiC" - Mathematics in Computer Comp.
BOX 15302, Bene-Ayish, Israel, 60860
Fax: ++972-3-9471301
Printed in the United States of America, Lulu Inc., ID **86485**

Summary

This book describes various versions of processors, designed for affine transformations of many-dimensional figures – planar and spatial. This processors is oriented to affine transformation of unstructured geometrical figures with arbitrary points distribution. The type of data presentation used in this book is non-conventional, based on a not well-known theory of vectors and geometrical figures coding. The problems of affine transformation are used widely in science and engineering. The examples of their application are computer tomography and data compression for telecommunication systems.

The book covers the figures coding theory – the codes structure, algorithms of coding and decoding for planar and spatial figures, arithmetical operations with planar and spatial figures. The theory is supplemented by numerous examples. The arrangement of several versions of geometrical processor is considered – data representation, operating blocks, hardwares realization of coding, decoding and arithmetic operations algorithms. The processor's internal performance is appraised.

The book is designed for students, engineers and developers, who intend to use the computer arithmetic of geometrical figures in their own research and development in the field of specialized processors. With that in view the book includes

- Theory of coding,
- Operations algorithms,
- Examples of coding, decoding and computations,
- Description of several versions of processors,
- A system of commands for them,
- Schemes of operational units,
- Comparative analysis.

Algorithms and units described in this book are developed into models in VHDL and FPGA. We shall welcome any kind of cooperation proposals sent to the address:

solik@netvision.net.il

Contents

1. Introduction \ 8

2. Prototypes \ 13

2.1. Data Representation \ 13

2.2. The Simplest Arithmetic Unit \ 14

2.3. Arithmetic Unit with Rectangular Codes \ 17

3. Foundations of Computer Arithmetic for Complex Numbers and Vectors \ 19

3.1. Coding Method for Complex Numbers \ 19

3.2. Special Algebra in Vector Space \ 21

3.2.1. Algebra in 3-dimensional vector space \ 21

3.2.2. Component-wise multiplication \ 22

3.2.3. Vector product \ 22

3.2.4. Scalar product \ 22

3.2.5. The turning of a vector \ 22

3.2.6. Centroaffine transformation \ 23

3.2.7. Many-dimensional space \ 24

3.3. Two Methods of Multidimensional Vectors Coding \ 25

3.3.1. Method 1 \ 29

3.3.2. Method 2 \ 26

3.4. Algebraic addition of M-codes \ 28

3.4.1. Multidigit circuits for M-codes \ 28

3.4.2. M-code Inverter \ 29

3.4.3. M-codes Inverse Adder \ 29

3.4.4. M-code Adder \ 30

3.4.5. M-code Subtractor \ 31

3.4.6. Sign Determinant M-code \ 31

3.5. Multiplication of Many-dimensional Vectors \ 34

3.5.1. Multiplication Method of Many-dimensional Vectors \ 34

3.5.2. Multiplication by Base Function to the Radix (3.3.10) \ 34

3.5.3. Multiplication by Base Function to the Radix (3.3.7) \ 34

3.5.4. Multiplication of the Whole Codes of Vectors to the Radix (3.3.10) \ 35
3.5.5. Multiplication of the Whole Codes of Vectors to the Radix (3.3.7) \ 36
3.5.6. Componentwise Multiplication of Many-dimensional Vectors \ 37
3.6. Scalar and Vector Multiplication \ 38
3.6.1. Scalar Product \ 38
3.6.2. Vector Product \ 39
3.6.3. Carries in Scalar Multiplication \ 40
3.6.4. Carries in Vector Multiplication \ 42
3.7. Algoritms and Devices for Coding and Decoding of Complex Numbers and Vectors \ 44
3.7.1. Coding of Complex Number in System 1 \ 44
3.7.2. Decoding of Complex Number in System 1 \ 45
3.7.3. Coding of Complex Number in System 2 \ 45
3.7.4. Decoding of Complex Number in System 2 \ 45
3.7.5. Coder of Positive M-code into P-code \ 46
3.7.6. Decoder of M-code into P-code \ 47
3.7.7. Full Decoder of M-code into P-code \ 50
3.7.8. Precoder of P-code into M-code \ 50
3.7.9. Partitioning Unit for Parts of the Code \ 51
4. Vector Processor \ 53
4.1. Data Representation and Vector Arithmetic Unit \ 53
4.2. Comparisons \ 57
5. Figure Coding theory \ 60
5.1. Primary Geometrical Codes \ 60
5.1.1. Data Representation \ 60
5.1.2. Arithmetic operations with geometrical codes in a real radix \ 63
5.1.2.1. Introduction \ 63
5.1.2.2. Writing of Base Code \ 64
5.1.2.3. Transpositions \ 64
5.1.2.4. Addition of Geometrical and Basic Codes when ρ=2\ 65
5.1.2.5. Algebraic Addition of Geometrical and Basic Codes when ρ=2 \ 68
5.1.2.6. Algebraic Addition of Geometrical and Basic Codes when ρ=-2 \ 68
5.1.2.7. Multiplication of Geometrical and Basic Codes \ 71

5.1.2.8. Division of Geometrical Code by Basic Code \ 76
5.1.2.9. Rounding-off of Geometrical Code \ 76
5.1.3. Geometrical codes in a complex radix \ 77
5.1.3.1. Algebraic Addition of Geometrical and Basic Codes \ 77
5.1.3.2. Multiplication of Geometrical and Basic Codes \ 79
5.1.4. Coding and transformation of planar figures \ 82
5.1.4.1. Method of coding \ 82
5.1.4.2. Carry \ 86
5.1.4.3. Centroaffine transformation \ 86
5.1.4.4. Affine transformation \ 87
5.1.5. Coding and Transformation of Spatial Figure \ 89
5.2. Attribute Geometrical Codes \ 91
5.2.1. Data Representation \ 91
5.2.2. AGC in a real radix\ 94
5.2.2.1. Writing of a given Number \ 94
5.2.2.2. Writing of a given Value \ 94
5.2.2.3. Reading the value of the path with the given number \ 95
5.2.2.4. Addition of AGC to the basic code when ρ=2 \ 95
5.2.2.5. Inverse addition of AGC to the basic code when ρ= -2 \ 97
5.2.2.6. Inversion of AGC when ρ=-2 \ 99
5.2.2.7. Algebraic addition of AGC \ 99
5.2.2.8. Search for the Next Open Path, its Number and it's Value \ 100
5.2.2.9. Multiplication of AGC by the basic code \ 100
5.2.3. Attribute geometrical codes in a complex radix \ 101
5.2.3.1. Inverse addition with the basic code \ 101
5.2.3.2. Invertion \ 102
5.2.3.3. Deformation \ 102
5.2.4. Attribute Geometrical Codes of Spatial Figures \ 106
5.2.5. Contracted attribute geometrical codes \ 108
6. Geometrical Processor \ 110
6.0. Data Presentation \ 110
6.1. Full Specific Random-access Memory \ 113
6.2. Fragmentary Specific Random-access Memory \ 114
6.3. Maximal Arithmetic Unit of Geometrical Figures \ 118
6.4. Fragmentary Arithmetic Unit of Geometrical Figures \ 119
6.5. Processor with a Maximal Arithmetical unit \ 121
6.6. Processor with Fragmentary Arithmetic Unit \ 124
6.7. The Main Procedures \ 127

6.7.1. Affine Transformation \ 127
6.7.2. Rounding \ 127
6.7.3. Rough rounding \ 128
6.7.4. Attributes Correction \ 129
6.7.5. Attributes Calculation \ 129
6.7.6. Coding a Figure \ 130
6.7.7. Decoding a Figure \ 130
6.8. Operational units \ 131
6.8.1. Writing unit for the number with the given code \ 131
6.8.2. Writing unit for the value with the given code \ 132
6.8.3. Reading unit for path value with the given number \ 132
6.8.4. Inverse adder \ 133
6.8.5. Search unit for the first open path, its numbers and its values \ 134
6.8.6. Reading unit for the number and value of the path with the given terminal vertex \ 135
6.8.7. Next terminal vertex search unit \ 136
7. Comparative Analysis \ 137
References \ 142
Designation \ 143
List of Examples \ 145
List of Tables \ 146
List of Figures \ 149

1. Introduction

This book is concerned with a full theory, not well known, and with patented engineering solutions for computer arithmetic of geometrical figures – planar and spatial. This theory is directed to the **affine** transformation of unstructured geometrical figures with arbitrary way of pints distribution. The transformation is aimed at this structure's identification. That's why the observed object may be defined only as a space in which a point has certain characteristics. The problems concerned with affine transformation of space are widely used in science and engineering – in medicine, in data processing and visualization, in astronomy, in seismology etc. Most striking and well-known examples of affine transformation applications are computer пraphics [1, 2], computer tomography [3] and information compression for telecommunication systems [4].

This book describes affine transformations (displacements, turns, scaling, shifts) of *n*-dimensional figures, where *n*=2,3,4. Usually the above-mentioned transformations are performed by calculating the coordinates of the points of the transformed figure according to the known coordinates of the points of the initial figure. However this method takes up a great deal of computer time since the calculation of coordinates is performed sequentially for every point and requires several operations per point (for instance, in order to calculate the new coordinates during an affine transformation of a planar figure, 4 operations each of addition and multiplication are required).

The above problems include operations with complex numbers since a point on a plane can be represented by a complex number. In this case operations of the same name can be performed simultaneously with a set of complex numbers. Processors with SIMD (Single Instruction, Multiple Data) architecture are used to solve problems of this type. However these processors operate with real numbers, and each operation with complex numbers requires several operations with real numbers – the real and imaginary parts of these complex numbers. Similarly, geometrical transformations in a three-dimensional space operate with three-dimensional vectors – sets of three real numbers. And each operation with vectors demands even more operations with real numbers. All this increases the calculation time substantially. In addition, set of complex numbers and vectors describing a figure takes up a great deal of memory. Thus there is a need for a method and system for effective SIMD calculations with a set of complex numbers and vectors that describe a figure. These calculations must be efficient as to calculation time and memory requirements.

The solution of the above problems can be greatly accelerated with special coding of *sets of complex numbers*. Due to that, reviewed further will be a method of *representing a set of complex numbers and vectors by a so-called geometrical code*, and then various operations with them will be described, as well as the hardware support of these operations. The geometric codes were first put forward in [5, 6] and were considered also in [7-14]. In the construction of a geometrical code a method of complex numbers and vectors representation by a single binary code [11-16] is being used.

By this method the set of binary codes of complex numbers and of the vectors is represented by a single binary code. Its volume is considerably smaller that the total volume of the initial binary codes array. The comparative volume reduction depends on the amount of numbers being coded and increases as this amount grows. The coded set of complex numbers is NOT structured. We can say that the set is a random one. The coded complex numbers and vectors are a coordinate set (which is significant) that the calculations are to be performed with.
Any additional information about the points (for instance, their color), if it does not take part in the calculations, is not subject to coding, and should be saved in a separate array –an attribute array. Geometrical code saves (in addition to the coordinates) also the information about every point's connection with its attributes.

All arithmetic operations may be performed with geometrical code (complex numbers and vectors algebraic addition, multiplication, affine transformation). These operations are equivalent to group operations with the coordinates of all points simultaneously.

It is significant that the performance time of an operation with geometrical code is equal to the performance time of the same operation with a pair of numbers, if *only* the whole geometrical code may be placed in the operative register of arithmetic unit.

It is assumed that the initial codes were codes with fixed point (for example, the coordinates of the point on the screen).

A method of geometrical code fragmentation is also proposed, allowing to operate with separate fragments of the geometrical code, if the arithmetic units register's dimension is not sufficient to hold the whole code.

It is important that geometrical code permits to operate with a geometrical figure as a whole, single object. So the data volume (coordinate codes) is reduced. However (and it should be emphasized to avoid mistakes), geometrical code does not compress geometrical figures themselves. It is assumed that the coded geometrical figure is described by a random set of points and does not have any special structure, which is typical for raster images.

In general, application of GC reduces data volume *n* times, where *n* – digit capacity of linear codes. The group operations performance speed is many times higher than the same operations speed for group operation with complex codes array. General computations time is also reduced due to data access time reduction.

Next we shall consider three types of arithmetic units -

- Traditional, operating with the proposed vector codes and containing several calculators, working simultaneously.
- Vectorial, operating with the proposed vector codes and also containing several calculators, working simultaneously, and
- Geometrical, operating with geometrical codes of figures.

We shall also consider a specialized random-access memory unit based on the geometrical figures coding method.

A comparison between the performances of these units is given. It appears reasonable to describe the device's *performance* by a ratio between the unit's volume and the number of certain procedures performed by the unit in a time unit. Let us call this ratio *relative volume* of a unit. A standard procedure for arithmetic units is affine transformation. For random-access memory units such standard procedures are either search for a point with given coordinates in an unordered array, or plain access, or a mixture of these procedures.

Fig. 1.1 gives a bar graph of the relative volume of the named arithmetic units in relation to the dimension *p* of the coded space. The unit of measurement in this figure is *14*M*, where *M* – number of points in the coded space. For example, for p=3 the ratio of relative volumes values is **(84:14:1).**

To compare the variants of random-access memory realization let us assume that in a given problem the reading/writing operations are *H* time more frequent than operations of search by given coordinates. It is shown that the relative volume of specialized RAM is *(~M/10H)* times smaller than the relative volume of traditional RAM unit.

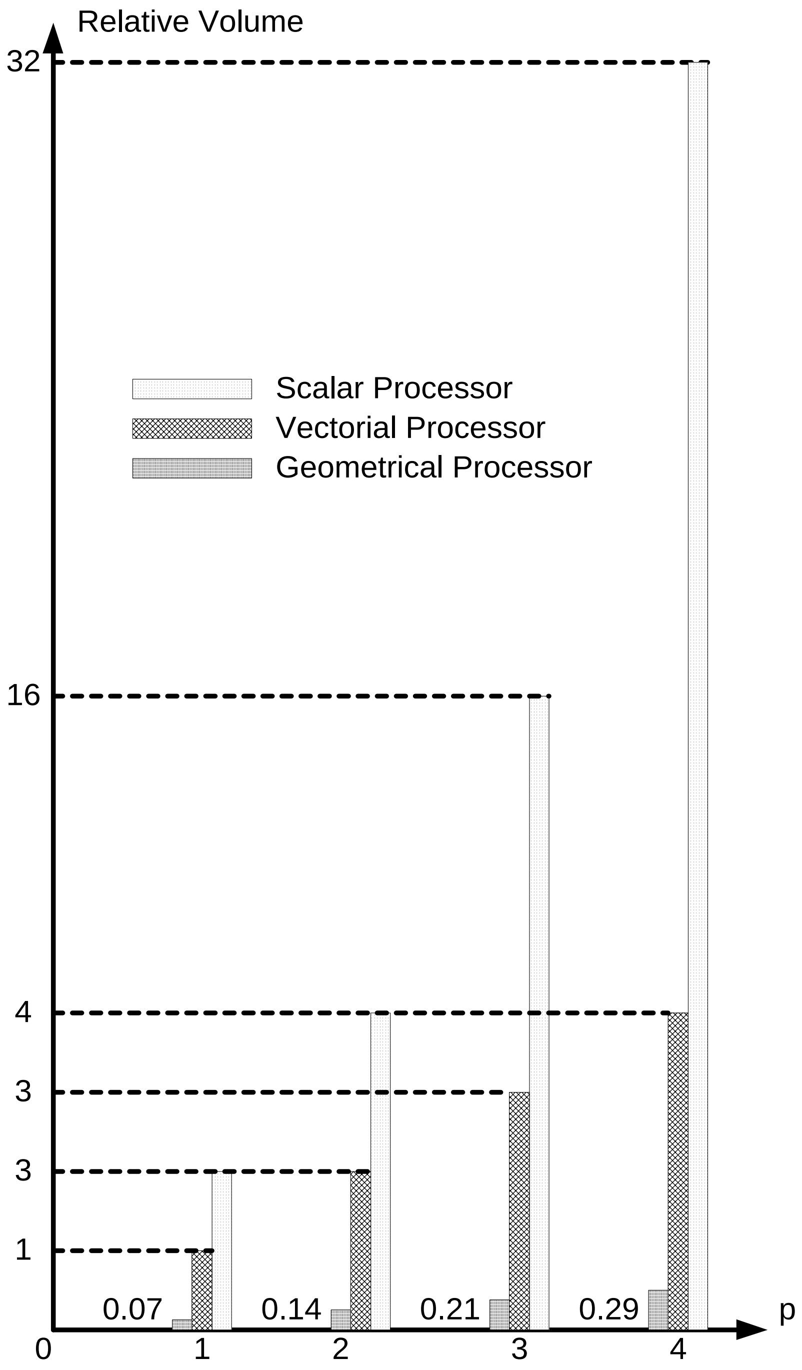


Fig. 1.1. Bar graph of the relative volume of the named arithmetic.

The book consists of 7 chapters, including the present introduction.

The second chapter is concerned with known devices for figure transformation – with one or several calculators.

The **third chapter** presents foundations of computer arithmetic for complex numbers and vectors –theory and hardware solutions. This chapter is essential, since in coding and decoding the geometrical figure code, we have to make use of the codes of complex numbers and vectors.

In the **fourth chapter** a vector arithmetic unit is discussed, which is based on vector computer arithmetic, stated in the previous chapter.

In the **fifth chapter** the theory of figures coding is described – the structure of codes, coding and decoding algorithms for planar and spatial figures, arithmetic operations with planar and spatial figures. The theory is supplemented by numerous examples.

The **sixth chapter** deals with the arrangement of raster geometric processor – data representation, operational units, technical realization of coding, decoding and arithmetic operations algorithms; the operating speed of this processor is also appraised.

The **seventh chapter** is dedicated to the characteristics comparison between arithmetic units and random-access memory units designed for operations with figures. The traditional units, described in the previous chapters, are being compared with the vector arithmetic units and geometrical figures arithmetic units.

2. Prototypes

2.1. Data Representation

The processors presented below are intended for solving the following problem. A set of points in a p-dimensional space is given. The points comprise a domain of definition, which is a p-dimensional cube, and they are distributed in this domain at the nodes of a uniform network. Each coordinate is represented by an n-digit code with fixed point. The network step is equal to the lowest digit of this code. Each point is characterized by its coordinates and attributes (certain values, associated with each point). We shall say that in such way a figure in p-dimensional space, or, simply, a p-dimensional figure F, is defined. We must find another figure F_a, obtained from the figure F by means of an affine transformation. The affine transformation is described by a p-dimensional transformation matrix and by an r-digit carry vector (for carry of this figure in some direction). Each element of the transformation matrix is represented by an n-dimensional code with a fixed point. The total digit capacity of the affine transformation parameters is equal to

$$a=p\,n+p^2\,r. \qquad (2.1.1)$$

For each point a pair "*coordinates-attribute*" is stored in the processor's memory. Evidently, the maximal number of coded points is

$$M=2^{pn}. \qquad (2.1.2)$$

Addresses of these pairs do not change during the problem solution in order to be able to find the point's attribute by the coordinates modified in the process of the coordinates transformation. Moreover, during some transformations the points' coordinates may coincide. Hence every point is determined by a triad "*address-coordinates-attribute*".

In future a processor for solving the described problem will be called a raster geometrical processor – **RGP**. Further we shall consider different variants of arithmetic units for such processor.

2.2. The Simplest Arithmetic Unit

As a preliminary we shall view a simplest construction of a scalar arithmetic unit **SAU** (see Fig. 2.2.1); it will be used further for analogies and comparison with the more complex constructions. In this unit a simplest multiplier of sequential type is used, containing only a shifter and an adder.

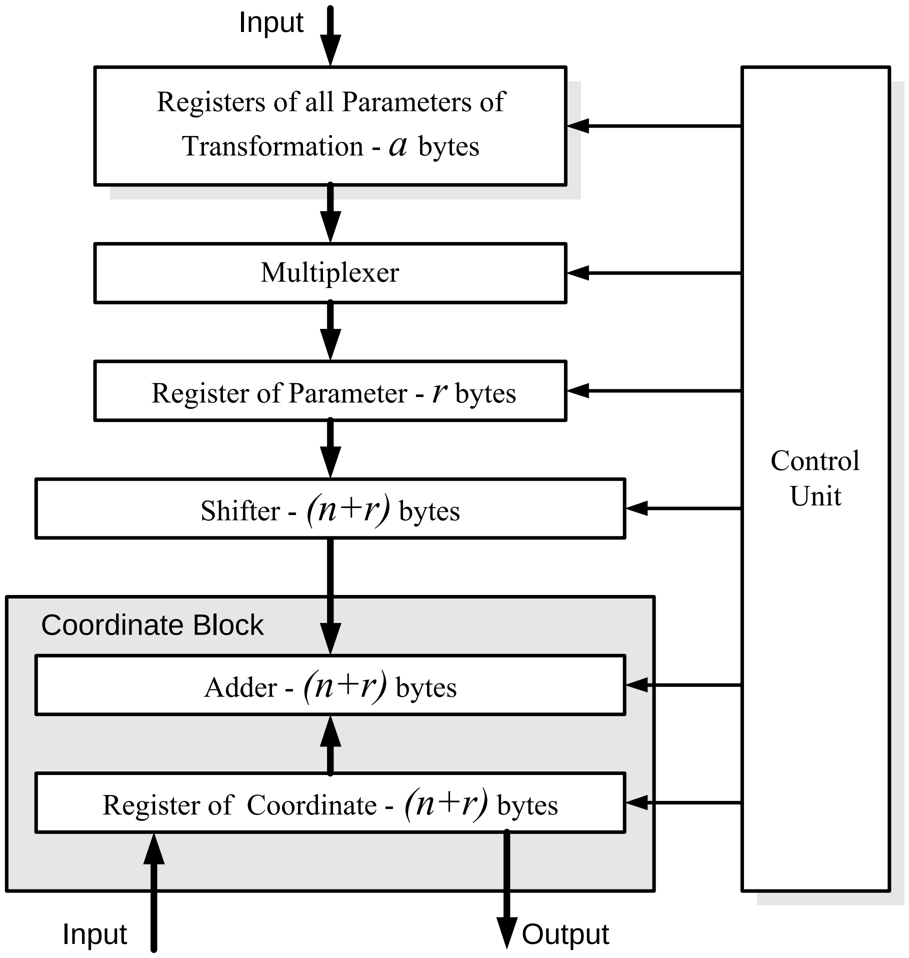

Input
Registers of all Parameters of Transformation - a bytes
Multiplexer
Register of Parameter - r bytes
Shifter - $(n+r)$ bytes
Coordinate Block
Adder - $(n+r)$ bytes
Register of Coordinate - $(n+r)$ bytes
Control Unit
Input
Output

Fig. 2.2.1. The simplest arithmetic unit.

In this SAU we need only an *(n+r)*–digit adder, *(n+r)*–digit multiplier, *(n+r)*–digit coordinate register, *(r)*–digit register of the chosen parameter and *a*–digit register of all the transformation parameters – the components of transformation matrix and carry vector – see (2.1.1). Besides, this SAU contains a multiplexer for parameter choice and a control unit. Affine transformation of any point contains p^2 multiplications and

$$D = p(p-1) = 2,\ 6,\ 12,\dots\ \mathit{if}\ \ p = 2,\ 3,\ 4,\dots \qquad (2.2.1)$$

additions.

In this arithmetic unit the adder serves for adding the coordinate to one of the components of the carry vector, and for adding the partial product to the multiplicand when performing multiplication. The sum register in this AU contain triggers with complementing input, and therefore it may be combined with the register of one of the addends – the register of the initial coordinate value.

The multiplier in this AU realizes the following algorithm

0. Given: r-digit factor A, representing one component of the transformation matrix, and an n-digit multiplicand *B,* representing one coordinate of the point.
1. At the beginning the partial product is equal to 0, and the multiplicand *B* is located so, that its lowest *0*-digit is combined with the highest *(n-1)*-digit α_{n-1} of the multiplicand *A*. The number of the current multiplicand's digit is *t=n-1*, i. e. $\alpha_t = \alpha_{n-1}$.
2. Adding the partial product to the multiplicand *B*, if $\alpha_t = 1$.
3. Shifting the multiplicand *B* to the right by1 digit and decreasing the current number $t := t - 1$.
4. Stopping the calculation if $t < 0$, or going to 2. As a result an *(n+r)*–digit product C is formed.

Thus, the multiplier consists only of the shifter to the left by 1 digit, and of the adder.

The digit capacity of the affine transformation results may reach *(n+r)*. This may lead to some points exceeding the bounds of the initial p-cube. At that we may:

1. or exclude these points from the given set (by putting an appropriate sign in the point's attribute),
2. or round off all the coordinates codes (by discarding the lower digits) and change the network step (that is a parameter of the whole figure).

In either case it may occur that

1. There are several points present in a certain node of the network. The attribute of the node is defined as a function of the attributes of all points present in this node. If for instance the attribute is intensity of monochrome color, then this function is a simple average of the intensities of joined vectors.
2. A point is lacking in a certain node. The node's attribute is defined as a function of the attributes of all adjacent nodes.

Let us review now the list of processor's commands realized in the SAU:

- Receiving the transformation parameters
- Receiving coordinate
- Adding to the carry vector component (*D* modifications – see (2.2.1))
- Multiplying by a transformation matrix component (p^2 modifications)
- Yielding a coordinate
- Rounding off

2.3. Arithmetic Unit with Rectangular Codes

If we are not keen on hardware economy, then we may construct a **MSAU**, containing a set (*M*) of simultaneously working elementary arithmetic units SAU. In such unit the codes of one coordinates for all the figure's points comprise an array called a *rectangular number code* **RCS**. RCS contains *M* registers of digit capacity *(n+r)*. MSAU contains total *M* adders of digit capacity *(n+r), M* multipliers of digit capacity *(n+r)*, RCS and one *a*–digit register of parameters. This MSAU performs group operations - adding RCS to carry code and multiplying RCS by the code of centroaffine transformation matrix element. Affine transformation of a figure contains p^2 group multiplications and *D* group additions.

Such a version of MSAU must have a very large size and its realization is beyond the bounds of today technology. So let us consider another version, intermediate between arithmetic units with individual and group operations. For that let us divide RCS in several fragments RCS_q, each containing *Q* registers of digit capacity *(n+r)*. Arithmetic unit **FSAU** totally contains *Q* adders of digit capacity *(n+r), Q* multipliers of digit capacity *(n+r)*, RCS_q and one *a*–digit register of parameters. The diagram of FSAU is presented in the Fig. 2.2.2. This diagram is similar to the diagram of Fig. 2.2.1, apart from the fact that *Q* coordinate units are used in it, as is shown in the Fig. 2.2.1.

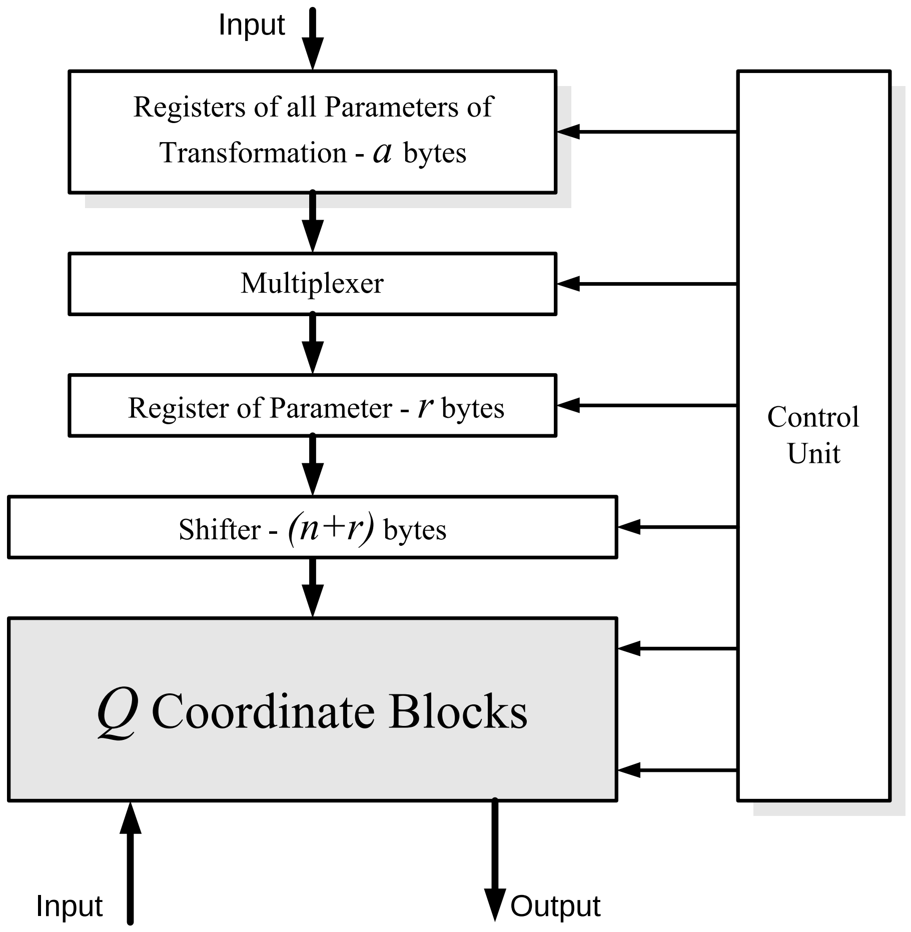

Input
Registers of all Parameters of Transformation - a bytes
Multiplexer
Register of Parameter - r bytes
Shifter - $(n+r)$ bytes
Q Coordinate Blocks
Control Unit
Input
Output

Fig. 2.2.2. Arithmetic unit with fragmentary rectangular codes.

FSAU performs group operations with the point's coordinates of a figure's fragment. There the affine transformation of a figure contains Qp^2 group multiplications and QD group additions.

3. Foundations of Computer Arithmetic for Complex Numbers and Vectors

3.1. Coding method for complex numbers

Complex numbers can be used as the radix for coding linear codes. Such code corresponds to the complex number Z being represented by decomposition of this type:

$$Z = \sum_{m} \alpha_m f(\rho,\ m),$$

where m – is the number of the decomposition digit,

$\alpha_m = \{0,\ 1\}$ – is the value of the decomposition digit,

ρ – is the radix of decomposition,

$f(\rho,\ m)$ – is the basic function of the number and radix.

The binary positional code of the complex number Z that corresponds to this decomposition looks as follows:

$$K(Z) = \ldots\alpha_m\ldots$$

In Table 3.1.1. all the possible basic functions are enumerated [14]. For illustration we shall now show the binary codes of numbers in all the enumerated coding systems, including coding system with a real (positive and negative) and complex radixes - see. Table 3.1.2.

Table 3.1.1. Systems of complex numbers coding

№	Basic Function	Radix
1	$f(\rho,\ m)=\begin{cases}\rho^{m/2} \text{ if } m-\text{even}\\ j\cdot\rho^{m-1/2} \text{ if } m-\text{odd}\end{cases}$	$\rho=-2$
2	$f(\rho,\ m)=\rho^m$	$\rho = j\sqrt{2}$
3	$f(\rho,\ m)=\rho^m$	$\rho = -j\sqrt{2}$
4	$f(\rho,\ m)=\rho^m$	$\rho=(-1+j)$
5	$f(\rho,\ m)=\rho^m$	$\rho=(-1-j)$
6	$f(\rho,\ m)=\rho^m$	$\rho = \frac{1}{2}(-1+j\sqrt{7})$

Table 3.1.2. Binary coding systems.

	ρ	2	ρ	$\overline{\rho}$	- 2	- 1
1	$\begin{cases}(-2)^{m/2} \text{ if } m-\text{even}\\ j(-2)^{m-1/2} \text{ if } m-\text{odd}\end{cases}$	10100	10	1010	100	101
2	$j\sqrt{2}$	10100	10	1010	100	101
3	$-j\sqrt{2}$	10100	10	1010	100	101
4	$-1+j$	1100	10	110	11100	11101
5	$-1-j$	1100	10	110	11100	11101
6	$\frac{1}{2}(-1+j\sqrt{7})$	1010	10	101	110	111
7	-2	110	10		10	11
8	2	10	10			

3.2. Special Algebra in Vector Space

3.2.1. Algebra in 3-dimensional vector space

Let us consider a certain algebra in a three-dimensional vector space. Let **i, j, k** be the base of a 3-dimensional vector space. Let us determine the ort mutiplication Table 3.2.1 for this base. According to this table, multiplication of vectors is described as follows: if $U = U_1 * U_2$, where

$$U_m = x_m i + y_m j + z_m k$$

for any *m* indexes, then

$$x = x_1x_2 - y_1z_2 - z_1y_2,$$
$$y = x_1y_2 + y_1x_2 - z_1z_2,$$
$$z = x_1z_2 + y_1y_2 + z_1x_2.$$

Table 3.2.1. 3-dimensional vectors multiplication

*	**i**	**j**	**k**
i	*i*	*j*	*k*
j	*j*	*k*	*-i*
k	*k*	*-i*	*-j*

This multiplication does not have anything in common with vector or scalar multiplication and, unlike those, is designated further by the symbol ‘*’. It is not hard to ascertain that multiplication determined by Table 3.2.1 for orts **i, j, k** will be associative and commutative. Consequently, multiplication determined for any vectors or the vector space under consideration, will also be commutative and distributive with respect to addition. Aside from that, the following condition is fulfilled:

$$(bU_1)*U_2 = U_1*(bU_2) = b(U_1*U_2),$$

where *b* is a real number. Consequently, Table 3.2.1 determines, within a 3-dimensional vectorial space, an algebra without division over a real numbers field.

Addition within the algebra under consideration corresponds to common addition of vectors, and multiplication – to turning the multiplicand vector while simultaneously changing its length. In a general case, the turn parameters, i.e. the position of the turn axis, the angle of the turn, and the scale multiplier, depend upon the coordinates of both comultipliers. Therefore, the geometrical interpretation of multiplication in this algebra is fairly complex, however a few operations of multiplication and addition will be enough to decribe those vector transformations that have simple geometrical meaning.

3.2.2. Component-wise multiplication

Let us consider certain operations in this algebra, having first defined an operation referred to as *component-wise multiplication*. This term will be used to name the operation of multiplication of vector U_1 by an ordered threesome of vectors U_2, U_3, U_4. Component-wise multiplication consists in calculating the vector according to the formula

$$U = x_1 i * U_2 + y_1 j * U_3 + z_1 k * U_4$$

or

$$x = x_1 x_2 - y_1 z_3 - z_1 y_4,$$
$$y = x_1 y_2 - y_1 x_3 - z_1 z_4,$$
$$z = x_1 z_2 - y_1 y_3 - z_1 x_4.$$

In order to designate this operation, we shall also use the following notation:

$$U = U_1 * [U_2, U_3, U_4].$$

In particular, $U_1 * [U_2, U_2, U_2] = U_1 * U_2$.

3.2.3. Vector product

$$U_1 \times U_2 = U_1 * [(-jz_2 + ky_2), (-jx_2 - kz_2), (jy_2 - kx_2)].$$

3.2.4. Scalar product

$$i(U_1 \bullet U_2) = U_1 * [(ix_2), (-ky_2), (-jz_2)]$$

- here, for calculation convenience, the real number $U_1 \bullet U_2$ is considered identical with the vector $i(U_1 \bullet U_2)$.

3.2.5. The turning of a vector

The turning of a vector, while it moves along the surface of a certain cone, can also be described by component-wise multiplication according to the threesome of vectors obtained from the turn parameters. Here, it is worthwhile to note the analogy with the algebra of complex numbers, where multiplication is equivalent to turning a planar vector.

Let us consider a straight line with an ort

$$r_o = iCos\alpha + jCos\beta + kCos\gamma,$$

passing through the point of origin. Let a point rotate around this line in a circle of a certain radius. Its radius vector turns from position U_1 to position U. If furthermore the point rotates counterclockwise (as observed from the edge of the vector r_o) and the angle of rotation is $0 \le \varphi \le \pi$, then it may be shown that

$$U = U_1 Cos\varphi + \left(r_o \times U_1\right) Sin\varphi - r_o\left(r_o \bullet U_1\right)\left(1 - Cos\varphi\right)$$

or

$$U = U_1 * [U_2, U_3, U_4].$$

where

$$U_2 = \begin{Bmatrix} i\left(Cos\varphi \cdot Sin^2\alpha + Cos^2\alpha\right) + \\ j\left(Cos\gamma \cdot Sin\varphi + Cos\alpha \cdot Cos\beta \cdot (1 - Cos\varphi)\right) + \\ k\left(-Cos\beta \cdot Sin\varphi + Cos\alpha \cdot Cos\gamma \cdot (1 - Cos\varphi)\right) \end{Bmatrix}$$

$$U_3 = \begin{Bmatrix} i\left(Cos\varphi \cdot Sin^2\beta + Cos^2\beta\right) + \\ j\left(Cos\alpha \cdot Sin\varphi + Cos\beta \cdot Cos\lambda \cdot (1 - Cos\varphi)\right) + \\ k\left(Cos\lambda \cdot Sin\varphi - Cos\alpha \cdot Cos\beta \cdot (1 - Cos\varphi)\right) \end{Bmatrix}$$

$$U_4 = \begin{Bmatrix} i\left(Cos\varphi \cdot Sin^2\gamma + Cos^2\gamma\right) + \\ j\left(-Cos\beta \cdot Sin\varphi - Cos\alpha \cdot Cos\lambda \cdot (1 - Cos\varphi)\right) + \\ k\left(Cos\alpha \cdot Cos\varphi - Cos\beta \cdot Cos\lambda \cdot (1 - Cos\varphi)\right) \end{Bmatrix}$$

3.2.6. Centroaffine transformation

Centroaffine transformation is equivalent to component-wise multiplication by three vectors, built from the elements of centroaffine matrix elements.

$$U_2 = U_1^T \cdot \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} == U_1 * \begin{bmatrix} (ia_{11} - ja_{12} - ka_{13}), \\ (ia_{21} - ja_{22} - ka_{23}), \\ (ia_{311} - ja_{32} - ka_{33}) \end{bmatrix}. \quad (3.2.1)$$

3.2.7. Many-dimensional space

Let us now generalize the results obtained so they can apply to n-dimensional space. Let us select a base within it, $E_1, E_2, ..., E_n$, the elements of which satisfy the following condition:

$$E_a E_b = - E_c \text{ with } d>n+1,$$

$$E_a E_b = E_c \text{ with } d<n+2,$$

where $d=(a+b-1)$, $c=d \bmod_n$. (3.2.2)

It can be demonstrated that multiplication, as determined for the elements of this base, is associative, commutative and distributive with respect to addition, and it also satisfies the following condition:

$$(bU_1)*U_2 = U_1*(bU_2) = b(U_1*U_2),$$

where b is a real number. Consequently, the set of n-dimesional vectors is an algebra. In particular, if n=2 and the base of this space is $\{1, j\}$, we obtain the algebra of complex numbers – see also Table 3.2.2 of complex numbers multiplication; if n=3 we come up with the algebra in Table 3.2.1 described above; if n=4, the last formula correspond to the Table 3.2.3 of multiplication of four orts, and so on.

Table 3.2.2. Compex numbers multiplication.

*	1	j
1	*1*	*j*
j	*j*	*-1*

Table 3.2.3. 4-dimensional vectors multiplication

*	i	j	k	m
i	*i*	*j*	*k*	*m*
j	*j*	*k*	*m*	*-i*
k	*k*	*m*	*-i*	*-j*
m	*m*	*-i*	*-j*	*-k*

3.3. Two Methods of Multidimensional Vector Codes Synthesis

As indicated earlier, complex number may be used as radixes of linear codes coding - see. Table 3.1.1. For the systems 1 and 2 this method is based on constructing a certain composition of codes of real numbers with a negative radix. Such method of constructing codes of complex numbers can be generalized and used for coding multidimensional vectors. Let us consider n real numbers $\{X\}_i$, each of which has been determined by binary decomposition with a radix $\rho = -2$, i.e.

$$X_i = \sum_{(m)} \alpha_m^i \rho^m , \qquad (3.3.1)$$

where $(i=1, 2,..., n)$. Each such decomposition has its corresponding code

$$K(X_i) = ...\alpha_m^i ... \qquad (3.3.2)$$

3.3.1. Method 1.

Let us now consider an n-dimensional vector:

$$Z = E_1 X_1 + E_2 X_2 + ... + E_n X_n , \qquad (3.3.3)$$

where $\{E_i\}$ is the base of n-dimensional vectorial space. In this case, the set of codes $\{K(X_i)\}$ can be interpreted as a unique code of the vector Z with a radix '-2'. Every m-digit of this code is represented by a set of binary digits, $\left\{\alpha_m^i\right\}$. Having assigned the figures σ_m to these sets, we obtain the vector code

$$K(Z) = ...\sigma_m ..., \qquad (3.3.4)$$

which corresponds to the decomposition

$$Z = \sum_{(m)} r_m \rho^m ,$$

where the vector

$$r_m = E_1 \alpha_m^1 + E_2 \alpha_m^2 + ... + E_i \alpha_m^i + ... + E_n \alpha_m^n \qquad (3.3.5)$$

is represented by the figure σ_m. In particular, if n=2, codes of complex numbers with a radix '-2' are formed, which were reviewed above. If n=3, codes of 3-dimensional vectors are formed, wherein the digits assume one of the following eight values:

$$r_m \in \{ \mathbf{0, i, j, k, i+j, i+k, j+k, i+j+k} \}, \qquad (3.3.6)$$

where **i, j, k** are orts of rectangular coordinate axes.

Similarly to the coding of complex numbers, for coding 3-dimensional vectors we can introduce a vector function of a real integer argument

$$\vartheta_2(m) = \begin{Bmatrix} i(-2)^m \text{ if } m = 3k \\ j(-2)^{m-1} \text{ if } m = 3k+1 \\ k(-2)^{m-2} \text{ if } m = 3k+2 \end{Bmatrix},$$

that will be designated hereinafter as ϑ_2^m. Note that the code of the 3-dimensional vector with a radix (-2), which we are reviewing, can be regarded as the code of a 3-dimensional vector with a radix ϑ_2 with binary digits. Vector decomposition in the form of $Z = \sum_m \left(\alpha_m \vartheta_2^m \right)$ corresponds to this code.

Similarly, for coding n-dimensional vectors we can introduce a vector function of a real integer argument:

$$\vartheta_n(m) = \begin{Bmatrix} i(-2)^m \text{ if } m = nk \\ j(-2)^{m-1} \text{ if } m = nk+1 \\ \dots \\ k(-2)^{m-n+1} \text{ if } m = nk+n-1 \end{Bmatrix}, \qquad (3.3.7)$$

further denoted as ϑ_n^m.

3.3.2. Method 2.

Let us now build, same as for complex numbers, a series of alternating binary digits α_m^i:

$$\cdots \alpha_{m+1}^2 \alpha_{m+1}^1 \alpha_m^n \alpha_m^{n-1} \cdots \alpha_m^2 \alpha_m^1 \alpha_{m-1}^n \alpha_{m-1}^{n-1} \cdots \qquad (3.3.8)$$

In other designations, this series is the binary code

$$K(Z) = \dots \alpha_k \cdots, \qquad (3.3.9)$$

of a certain vector Z. In this case the coding radix is also a vector

$$\rho = E_2 \sqrt[n]{2}\,, \tag{3.3.10}$$

where E_2 is the second ort of the base $\{E_i\}$ of n-dimensional vectorial space. The coded vector Z is determined in this situation according to the formula

$$Z = X_1 + \rho X_2 + \ldots + \rho^{i-1} X_i + \ldots + \rho^{n-1} X_n\,. \tag{3.3.11}$$

Positional codes of vectors lend themselves to operations of algebraic addition, vectorial and scalar multiplication, and component-wise multiplication. Algorithms of these operations contain cycles of algebraic addition of number codes and vector shifts, i.e. they are easy to implement technically. This may be utilized when building processors that operate with vectors as a whole. Such a processor requires a simpler algorithm to solve problems with vectors, and when using the given algorithm it works according to a shorter program and has increased performance speed. In order to assess these values, it can be stated, for instance, that a program of vector multiplication for vectors determined by three numbers contains 6 operations of multiplication and 3 operations of subtraction.

The best coding systems for creating geometrical codes formation are coding systems of complex numbers 1, 2, 3. The last two systems are similar in many respects. Therefore in further narrative we shall consider algorithms and units for operations only in the systems 1 and 2. In these systems the complex code is presented as a composition of real number codes to the radix (-2). Such codes are formed from traditional codes to the radix (2). Therefore as a preliminary we shall consider the algorithms and units for arithmetic operations, coding and decoding of real numbers to various radixes. Thus, let us consider now several types of binary codes:

- Traditional code in the radix “2” – the so-called **P-code**,
- Real numbers code in the radix “-2” – the so-called **M-code**,
- Complex code in complex radix – the so-called **C-code**.

The range of variation of a positive real integer number, represented by an *n*-digit **P-**code, are as follows:

$$(0) \div \left(2^{n+1} - 1\right).$$

The range of variation of a positive real integer number, represented by an *n*-digit **M-**code, are as follows:

$$\left(\frac{-2^n + 2}{3}\right) \div \left(\frac{2^{n+1} - 1}{3}\right) \text{ if } n - \text{odd;} \quad \left(\frac{-2^{n+1} + 2}{3}\right) \div \left(\frac{2^n - 1}{3}\right) \text{ if } n - \text{even}\,.$$

3.4. Algebraic addition of M-codes

The best systems for building geometrical codes are coding systems 1, 2. In these systems the complex code is presented as a composition of real number codes to the radix (-2). Therefore we shall first consider below the algorithms and units for arithmetic operations with M-codes.

3.4.1. Multidigit Circuits for M-codes

We shall consider next the devices for coding and decoding. The main units of these devices are linear multidigit circuits for algebraic addition. They consist of series-connected one-digit circuits – see Fig. 3.4.1, where

N – digit capacity of the codes,

k={0,1,2,…,n-2,n-1} – numbers of digits and one-digit circuits,

cop – operation code, common for all one-digit circuits,

V1, V2 – the input carry digits, representing the number *V*,

W1, W2 – the output carry digits, representing the number *W*,

A, B – the operands,

C – the result.

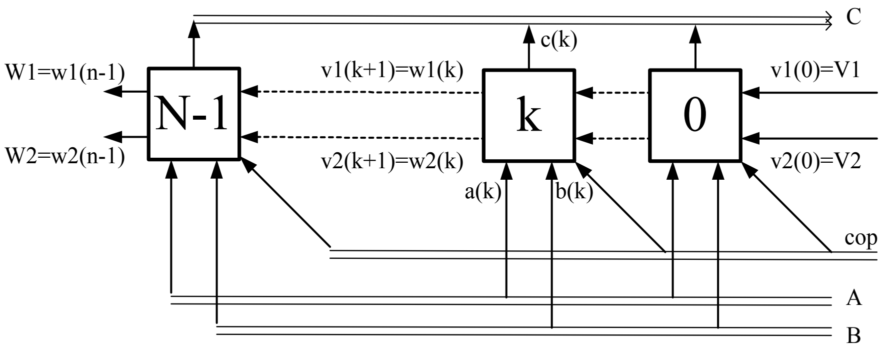


Fig. 3.4.1. Multidigit algebraic addition circuit

In special linear circuits cases the operation code and/or the second operand may be lacking. The input carry *V*, as a rule, is equal to zero. The input carry *V*, as a rule, is equal to zero. Non-zero output carry *W* is indicative of an overflow.

Further when considering specific schemes of algebraic addition, we shall describe (as a rule) only one-digit schemes.

3.4.2. M-code Inverter

Fig. 3.4.2 presents a one-digit inverting circuit Inv. Its functioning is described by a verity table Table 3.4.2. This table computes the value

$c-2*w=-a+v$.

Table 3.4.2. One-digit inverting circuit

a	v	w	c
0	0	0	0
1	0	1	1
0	1	0	1
1	1	0	0

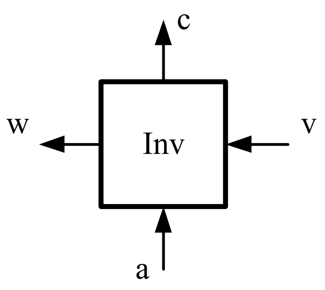


Fig. 3.4.2. One-digit inverting circuit.

3.4.3. M-codes Inverse Adder

Fig. 3.4.3 presents an one-digit inverse adder's circuit InvAdd. Its functioning is described by a verity table Table 3.4.3. This table computes the value $c-2*w=(-a-b+v)$.

Table 3.4.3. One-digit inverse adder circuit

a	b	v	w	c
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	0	1	0
0	0	1	0	1
0	1	1	0	0
1	0	1	0	0
1	1	1	1	1

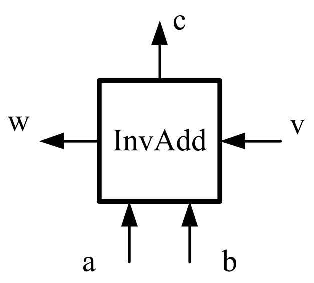


Fig. 3.4.3. One-digit inverse adder circuit.

3.4.4. M-code Adder

Figure 3.4.4 presents a one-digit adder scheme Add. Its functioning is described by truth table 3.4.4. This table calculates the sum

c-2 w = (a + b + v)*

Table 3.4.4. One-digit adder circuit

a	b	v1	v2	w1	w2	c
0	0	0	0	0	0	0
0	1	0	0	0	0	1
1	0	0	0	0	0	1
1	1	0	0	1	1	0
0	0	1	1	0	1	1
0	1	1	1	0	0	0
1	0	1	1	0	0	0
1	1	1	1	0	0	1
0	0	0	1	0	0	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	0	1	1	1	1

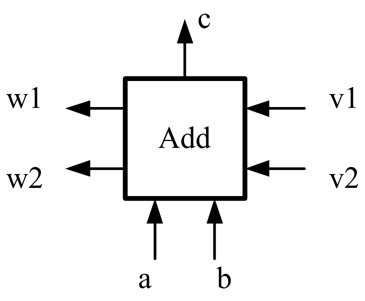


Fig. 3.4.4. One-digit adder circuit

3.4.5. M-code Subtractor

Figure 3.4.5 presents a one-digit subtractor scheme **Sub.** Its functioning is described by truth table 3.4.5. This table calculates the sum $c-2*w = (a - b + v)$.

Table 3.4.5. One-digit subtractor circuit

a	b	v1	v2	w1	w2	c
0	0	0	0	0	0	0
0	1	0	0	0	1	1
1	0	0	0	0	0	1
1	1	0	0	0	0	0
0	0	1	1	0	1	1
0	1	1	1	0	1	0
1	0	1	1	0	0	0
1	1	1	1	0	1	1
0	0	0	1	0	0	1
0	1	0	1	0	0	0
1	0	0	1	1	1	0
1	1	0	1	0	0	1

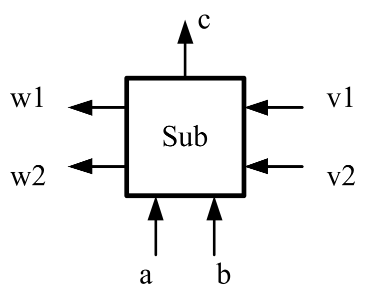


Fig. 3.4.5. One-digit subtractor circuit

3.4.6. Sign Determinant M-code

The sign determinant determines the sign of the number represented by M-code. It exploits one-digit sign determinants shown in the Fig. 3.4.6.1 of two modifications:

Seven – one-digit circuit of sign determinant for even-numbered digit,

Sodd - one-digit circuit of sign determinant for odd-numbered digit,

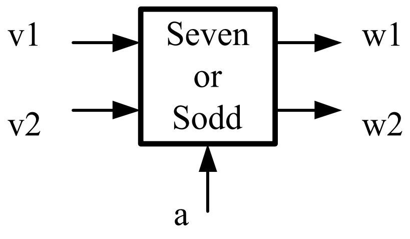


Fig. 3.4.6.1. One-digit sign determinant circuit.

The carries codes in these circuits are interpreted as follows:

00 – the code has a zero value,
01 – the code has a positive value,
10 – the code has a negative value.

The functioning of one-digit sign determinants *Seven* and *Sodd* is described in the tables 3.4.6.1 and 3.4.6.2 accordingly.

Table 3.4.6.1. One-digit sign determinant circuit for even-numbered digits

a	v2	v1	w2	w1
0	0	0	0	0
0	0	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	1	0	1

Table 3.4.6.1 realizes the rule

'w2, w1' = 'v2 v1', if a = '0',
'w2, w1' = '01', if a = '1'.

Table 3.4.6.2 realizes the rule

'w2, w1' = 'v2 v1', if a = '0',
'w2, w1' = '11', if a = '1'.

Table 3.4.6.2. One-digit sign determinant circuit for odd-numbered digits

a	v2	v1	w2	w1
0	0	0	0	0
0	0	1	0	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	1	1	1

Sign determinant as a whole - **nSign** is presented by Fig. 3.4.6.2, showing circuit diagram between the circuits *Seven* and *Sodd* and between them and the M-code circuit. The following notations are used

N - digit capacity of sign determinant,
A – input code,
W1, W2 – output carries.

The output carries code (W2, W1) is interpreted as follows:

00 – the code has a zero value,
01 – the code has a positive value,
10 – the code has a negative value.

Hence, if W2=1 then A<0, and if W2=0 then A>=0.

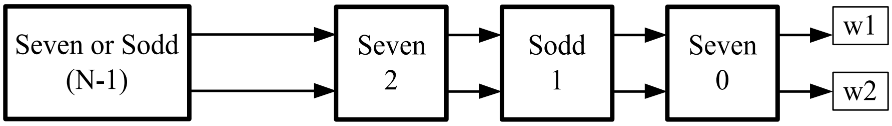


Fig. 3.4.6.2. Sign determinant

3.5. Multiplication of Many-dimensional Vectors

3.5.1. Multiplication Method of Many-dimensional Vectors

We have to find the product of vectors *C=A*B*, where the expansions of multiplier and multiplicand are accordingly:

$$A = \sum_h \alpha_h f(\rho\ ,h), \qquad B = \sum_k \beta_k f(\rho\ ,k).$$

Code of the product is defined as $C = \sum_h \lfloor B\alpha_h f(\rho\ ,h) \rfloor$.

Since $\alpha_h = \{0,1\}$, the multiplication consists only of multiplication by the base function $f(\rho,\ h)$ and summation. Let us review the multiplication by the base function for two cases important for our application.

3.5.2. Multiplication by Base Function to the Radix (3.3.10).

In this case multiplication of the multiplicand by the base function is equivalent to a shift by *h* digits. Consequently, multiplication of codes in this system amounts to successive operations of shift and addition.

3.5.3. Multiplication by Base Function to the Radix (3.3.7).

Dimension of multiplier is $N = n \cdot m$, where *n* – dimension of the vector. Code of the multiplier may be divided into *m* groups, and in each *t*-group there is the first (lowest) digit, second digit,… *i*-digit, … *n*-digit. We shall number the groups in the same way as the code's digits, from right to left from 0 to (*m* -1). The multiplication of multiplicand by the base function (if the corresponding digit of the multiplier is equal to 1) consists of two operations (which are simultaneous):

1. Shifting the multiplicand by $h = n \cdot t$ digits, if we are in the *t*-group of multiplier's digits
2. Multiplying the multiplicand *B* by the ort according to number *i* of the digit in the group.

$$B_i = E_i B,\ \ i = \overline{1,n} \qquad (3.5.1)$$

- see also (3.3.3). For example, if *n=2,* then:

$$B_1 = B,\ \ B_2 = jB;$$

if *n=3*, then:

$$B_1 = iB,\ \ B_2 = jB,\ \ B_3 = kB\,;$$

if $n=4$, then:

$$B_1 = iB,\ \ B_2 = jB,\ \ B_3 = kB,\ \ B_4 = mB$$

and so on. Notice, that the reformed multipliers B_i may be prepared before the multiplication.

Calculations by the formula (3.5.1) are performed according to the multiplication table of vector, or by formula (3.2.2). Let us in accordance with (3.3.3) assume that

$$B = E_1 b_1 + E_2 b_2 + \ldots + E_n b_n, \qquad (3.5.2)$$

Specifically, for complex numbers Table 3.2.2 is used. We have:

$$B_1 = B,\ \ B_2 = j(b_1 + jb_2) = -b_2 + jb_1.$$

For three-dimensional vectors Table 3.2.1 is used. We have:

$$B_1 = B,$$

$$B_2 = j(b_1 + jb_2 + kb_3) = -b_3 + jb_1 + kb_2,$$

$$B_{32} = k(b_1 + jb_2 + kb_3) = -b_2 - jb_3 + kb_1.$$

For four-dimensional vectors Table 3.2.3 is used. We have:

$$B_1 = B,$$

$$B_2 = j(b_1 + jb_2 + kb_3 + mb_4) = -b_4 + jb_1 + kb_2 + mb_3,$$

$$B_3 = k(b_1 + jb_2 + kb_3 + mb_4) = -b_3 - jb_4 + kb_1 + mb_2,$$

$$B_4 = m(b_1 + jb_2 + kb_3 + mb_4) = -b_2 - jb_3 - kb_4 + mb_1.$$

Hence it follows that multiplication of a code by a vector consists of inverting some of the components and permuting the vector code's components.

3.5.4. Multiplication of the Whole Codes of Vectors to the Radix (3.3.10).

Let us consider a system of coding n-dimensional vectors to the radix (3.3.10). For our application the digits of multiplier should be analyzed beginning from the highest, and the multiplicand should be shifted to the right. According to this, the multiplication algorithm is as follows:

1. At first the partial product is equal to 0, and the multiplicand B is located so, that its lowest *0*-digit coincide with the highest *(N-1)-*

digit α_{N-1} of the multiplier *A*. The number of the multiplier's current digit is *t=N-1*, i.e. $\alpha_t = \alpha_{N-1}$.

2. Add the partial product to multiplicand *B*, if $\alpha_t = 1$.
3. Shift multiplicand *B* by 1 digit to the right and decrease the current number $t := t - 1$.
4. Stop calculation if $t < 0$ or go to p. 2.

3.5.5. Multiplication of the Whole Codes of Vectors to the Radix (3.3.7).

Let us consider a coding system for *n*-dimensional vectors to the radix (3.3.7). The digit capacity of the multiplier's code is $N = n \cdot m$. In this case multiplication algorithm is as follows:

1. Prepare *n* variants of multiplicand *B* by formula (3.5.1).
2. Take groups of *n* digits of multiplier. Begin with the partial product equal to 0, and multiplicands B_i located in such way, that their lowest *0*-digits coincide with the digit of multiplier *A* of the number $N - n = n \cdot (m-1)$. The number of the current digit group of the multiplier is *t=m*.
3. Take the *t*-group of digits of the multiplier *A*. In it:
   3.1. Take the first (lowest) digit $\alpha_{n(t-1)}$. Perform addition of partial product to multiplicand B_1 if $\alpha_{n(t-1)} = 1$.
   3.2. Take the second digit $\alpha_{n(t-1)+1}$. Perform addition of partial product to multiplicand B_2, if $\alpha_{n(t-1)-1} = 1$.
   …
   3.i. Take the *i*-digit $\alpha_{n(t-1)+i}$. Perform addition of partial product to multiplicand B_i, if $\alpha_{n(t-1)+i} = 1$.
   ...
   3.n. Take the *n*-digit α_{nt}. Perform addition of partial product to multiplicand B_n, if $\alpha_{nt} = 1$.
4. Shift the multiplicand by *n* digits to the right (recall that here *n* – is dimension of the vector) and decrease the current number $t := t - 1$.
5. Stop calculation if $t < 0$, or go to p. 3.

3.5.6. Componentwise Multiplication of Many-dimensional Vectors.

Unlike simple multiplication, in componentwise multiplication for each cycle the value of multiplicand which is being added, depends on the number m of multiplier's digit (that is, to which component of multiplier vector this digit belongs). Let us assume that

m – number of digit of multiplier A,

k – integer number.

If componentwise multiplication of *a complex number* is being performed,

$$C = A * (X, Y),$$

then multiplicand B is defined as follows:

$B = X$, if $m = 3k$,

$B = Y$, if $m = 3k+1$.

If componentwise multiplication of *a three-dimensional vector* is being performed,

$$C = A * (X, Y, V),$$

then multiplicand B is defined as follows:

$B = X$, if $m = 3k$,

$B = Y$, if $m = 3k+1$,

$B = V$, if $m = 3k+2$.

If componentwise multiplication of *a four-dimensional vector* is being performed,

$$C = A * (X, Y, V, W),$$

then multiplicand B is defined as follows:

$B = X$, if $m = 3k$,

$B = Y$, if $m = 3k+1$,

$B = V$, if $m = 3k+2$,

$B = W$, if $m = 3k+3$.

As indicated earlier, componentwise multiplication by threesomes of vectors prepared beforehand, is equivalent to scalar and vector multiplication, multiplication by a number, centroaffine transformation, etc.

3.6. Scalar and Vector Multiplication

When performing the operations of *scalar and vector* multiplication **not** subject to any rules requisite for a ring, the multiplication becomes more complicated. It has been shown above that these operations may be substituted by componentwise multiplication. But to do this, the co-multipliers should be prepared beforehand. Hence we shall further consider some other methods of scalar and vector multiplication in a coding system of three-dimensional vectors to the radix (3.3.7), suggested in [16].

In this system the vector code takes the form (3.3.4), and its digits assume the values (3.3.6). We shall use the following 8 "numbers" for their designation:

$$\sigma_m = a, b, c, d, e, f, g, h\,.$$

The digits of vectors-co-multipliers V and W are further presented as vectors v_m and w_m with three components – real numbers taking value 0 or 1:

$$v_m = \{\alpha', \beta', \gamma'\},\ w_m = \{\alpha'', \beta'', \gamma''\}.$$

3.6.1. Scalar Product

By the formula of scalar product

$$v \bullet w = \alpha'\alpha'' + \beta'\beta'' + \gamma'\gamma'' \qquad (3.6.1)$$

Table 3.6.1 is built, where for products – number the codes to the radix ρ = -2 are indicated.

Table 3.6.1. One-digit scalar multiplication.

●	a	b	c	d	e	f	g	h
a	0							
b	0	1						
c	0	0	1					
d	0	1	1	110				
e	0	0	0	0	1			
f	0	1	0	1	1	110		
g	0	0	1	1	1	1	110	
h	0	1	1	110	1	110	110	111

Scalar product $Z = \boldsymbol{V} \bullet \boldsymbol{W}$ may be calculated by the formula:

$$Z = \sum_{k} (V \bullet w_k)\rho^{k}.$$

Consequently, scalar multiplication of multidigit vectors codes consists of cycles 'shift – scalar multiplication by *k*-digit of multiplier – addition' . This process results in forming the code of the number *Z* to the radix $\rho = -2$.

3.6.2. Vector Product

Formula of vector product is as follows: $Z = V \times W$, where $z_m = \{\alpha, \beta, \gamma\}$, and

$$\alpha = \beta'\gamma'' - \gamma'\beta'',$$
$$\beta = \gamma'\alpha'' - \alpha'\gamma'', \qquad (3.6.2)$$
$$\gamma = \alpha'\beta'' - \beta'\alpha''.$$

Coordinates of vector ***Z*** computed according to the formula (3.6.2) may assume values –1, 0, 1. Therefore, vector ***Z*** may be always presented as a difference of two vectors, each of them with a code to the radix (3.3.7). Using further the rules of algebraic addition of these vectors, we shall be able to build Table 3.6.2, describing vector multiplication. In contrast to the previous table, this table must be fully filled, as it is asymmetrical about the diagonal (vector product is non-commutative).

Table 3.6.2. One-digit vector multiplication

×	a	b	c	d	e	f	g	h
a	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
b	**0**	**0**	**e**	**e**	**cc**	**cc**	**cg**	**cg**
c	**0**	**ee**	**0**	**ee**	**b**	**ef**	**b**	**ef**
d	**0**	**ee**	**e**	**0**	**cd**	**gh**	**ch**	**cd**
e	**0**	**cc**	**bb**	**bd**	**0**	**c**	**bb**	**bd**
f	**0**	**cc**	**bf**	**bh**	**cc**	**0**	**dh**	**bf**
g	**0**	**eg**	**bb**	**fh**	**b**	**eh**	**0**	**eg**
h	**0**	**eg**	**bf**	**bd**	**cd**	**ef**	**cg**	**0**

Vector product ***Z=V***× ***W*** may be computed from the formula $Z = \sum_{k} (V \times w_k)\rho^{k}$. Consequently, the vector multiplication of multidigit vectors codes consists of cycles 'shift – vector multiplication by k-digit of the multiplier - addition'. The result is the forming the code of vector Z to the radix (3.3.7).

3.6.3. Carries in Scalar Multiplication.

When performing operations of shift and algebraic addition the vector's code may be conveniently considered consisting of three independent parts – codes of the numbers – projections of the vector, and the named operations are to be performed independently for each part. However during vector and scalar multiplication of one vector code by a digit of another vector code, there appears an added complication - a cross influence of the digits of dissimilar parts one onto another. Let us look into this question first for scalar multiplication.

Scalar multiplication is described by the formula (3.6.1), but in the case of multidigit code we must also take into account the carry p from the lower digit. Then the result for each digit should be calculated by the formula

$$S = \alpha'\alpha'' + \beta'\beta'' + \gamma'\gamma'' + p\,. \qquad (3.6.3)$$

The value S must be presented in the form

$$S = \sigma + P\rho, \qquad (3.6.4)$$

where σ=(0, 1) - the value of the considered digit of the result,
P - the value of carry into the higher digit.

It is easy to show that the carry P from the considered digit (and, therefore, also the carry p into this digit) may assume one of the values: P=(-1, 0, 1, 2). So S=(-1, 0, 1, 2, 3, 4, 5), and the scalar multiplication is described by the Table 3.6.3.

Table 3.6.3. Carries in scalar multiplication

S	-1	0	1	2	3	4	5
σ	1	0	1	0	1	0	1
P	1	0	0	-1	-1	2	2

The carries propagation may be organized another way, namely, so that the carry into a given digit would arrive from two previous digits (p and q) pass from the given digit to the two consequent digits (P and Q). In this case the digit-to digit result should be computed by the formula

$$S = \alpha'\alpha'' + \beta'\beta'' + \gamma'\gamma'' + p + q\,. \qquad (3.6.5)$$

and be presented in the form

$$S = \sigma + P\rho + Q\rho^2 \qquad (3.6.6)$$

In this case the carries will be able to take on only two values (0,1) and *S*=(0, 1, 2, 3, 4, 5). The scalar multiplication is described in Table 3.6.4.

Table 3.6.4 Carries in scalar multiplication.

S	**0**	**1**	**2**	**3**	**4**	**5**
σ	**0**	**1**	**0**	**1**	**0**	**1**
P	**0**	**0**	**1**	**1**	**0**	**0**
Q	**0**	**0**	**1**	**1**	**1**	**1**

Fig. 3.6.1. gives the scheme of adder in the unit of scalar multiplication. In this scheme the following notations are used:

a, b, c – digits of multiplicand's code,
d, e, f – digits of multiplier's code,
p – input carry,
P – output carry,
Sum – one-digit adder.

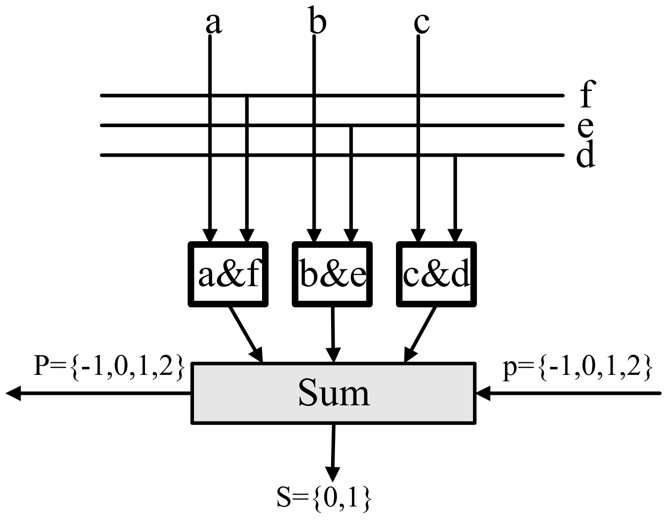


Fig. 3.6.1. Adder in the scalar multiplication unit.

3.6.4. Carries in Vector Multiplication

Vector multiplication is described by the formula (3.6.2). If we take into consideration the carries, the formula becomes as follows:

$$\begin{aligned} \rho P_\alpha + \alpha &= p_\alpha + \beta'\gamma'' - \gamma'\beta'', \\ \rho P_\beta + \beta &= p_\beta + \gamma'\alpha'' - \alpha'\gamma'', \\ \rho P_\gamma + \gamma &= p_\gamma + \alpha'\beta'' - \beta'\alpha''. \end{aligned} \tag{3.6.7}$$

where *p, P* – the values of carries into the given and the highest digits. In the formula (3.6.7) the carries can assume only two values (0,1). Consider now the first of these formulas. For it the operation is described by the Table 3.6.5.

Table 3.6.5. Carries in vector multiplication.

$\beta'\gamma''$	$\gamma'\beta''$	p_α	α	P_α
0	0	0	0	0
0	1	0	1	1
1	0	0	1	0
1	1	0	0	0
0	0	1	1	0
0	1	1	0	0
1	0	1	0	-1
1	1	1	1	0
0	0	-1	1	1
0	1	-1	0	1
1	0	-1	0	0
1	1	-1	1	1

Fig. 3.6.2. gives the scheme of adder in the vector multiplication unit. The notations in this scheme are as following:

a, b, c – digits of multiplicand's code,
d, e, f – digits of multiplier's code,
pG, pH, pM – input carries,
PG, PH, PM –output carries,
SumG, SumH, SumM – one-digit adders.

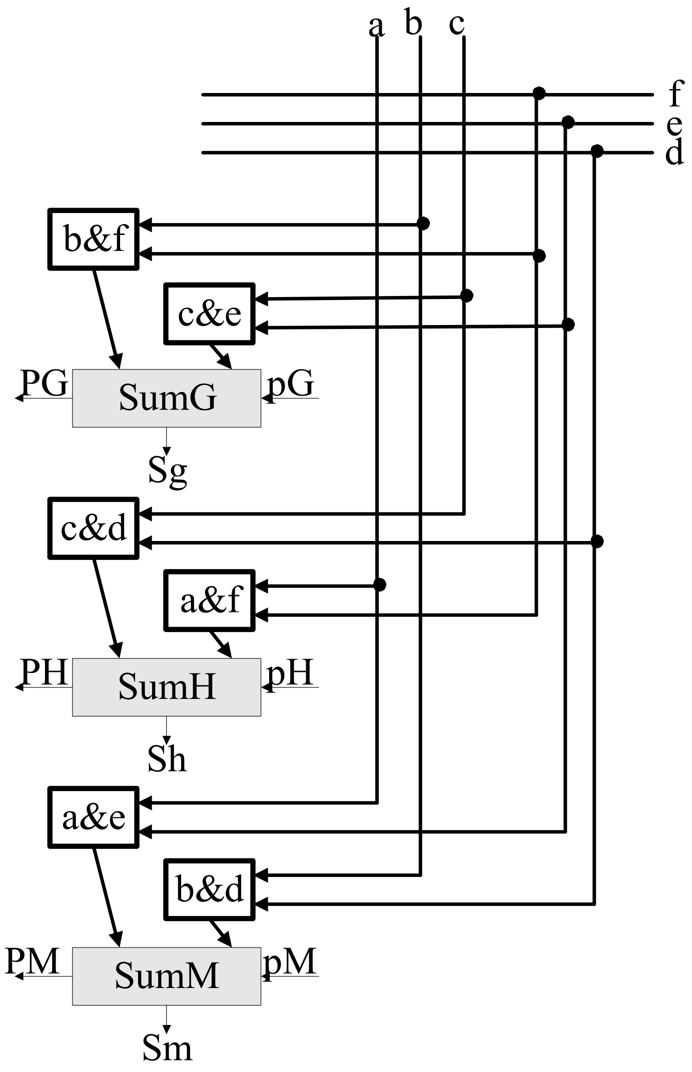


Fig. 3.6.2. Adder in vector multiplication unit.

3.7. Algorithms and Devices for Coding and Decoding of Complex Numbers and Vectors

The best coding systems for geometrical codes construction are the systems 1, 2, 3. The last two systems are similar in most ways. Therefore the further discussion will be confined to algorithms and coding devices only in system 1 and system 2. In these systems the complex code is represented by a composition of real number's codes in the radix "-2". Such codes are formed from the traditional codes in the radix "2". Hence we shall first discuss coding/decoding algorithms and devices for real numbers in different codes.

No consideration will be given to algorithms and devices for coding and decoding of vectors, as they are completely similar to the algorithms and devices for coding and decoding of complex numbers.

The complex number to be coded is presented as $Z = X_\alpha + jX_\beta$, where X_α, X_β - the real and imaginary part of the complex number, and are real (positive or negative) numbers

3.7.1. Coding of Complex Number in System 1

1. The coding of real (positive or negative) numbers X_α, X_β from P-code to M-code. It is advisable at first to perform the coding of only real positive numbers and to keep the signs $sign(X_\alpha)$, $sign(X_\beta)$ and M-codes of the numbers $|X_\alpha|$, $|X_\beta|$. For real positive numbers coding *a coder of positive P-code to M-code* is applied. Then with the aid of the *M-code inverter* we must compute the numbers
   $|X_\alpha| \cdot sign(X_\alpha)$, $|X_\beta| \cdot sign(X_\beta)$.
2. The generation of C-code
   $K(Z) = \ldots\beta_m\alpha_m\ldots\beta_1\alpha_1\beta_0\alpha_0, \beta_{-1}\alpha_{-1}\beta_{-2}\alpha_{-2}\ldots$ of the complex number $Z = X_\alpha + jX_\beta$, which in future will be represented by

$K(Z) = \ldots\gamma_m\ldots$, where $\left\{\begin{array}{l} \gamma_{2m} = \alpha_m \text{ if } m - \text{even} \\ \gamma_{2m+1} = \beta_m \text{ if } m - \text{odd} \end{array}\right\}$. To do this a *dispenser* should be used.

3.7.2. Decoding of Complex Number in System 1.

1. The extraction from complex number's C-code $Z = X_\alpha + jX_\beta$ of even and odd-numbered digits according to the rule $\left\{\begin{array}{l} \alpha_{m/2} = \gamma_m \text{ if } m - \text{even} \\ \beta_{(m-1)/2} = \gamma_m \text{ if } m - \text{odd} \end{array}\right\}$ and forming from them the digits α_m and β_m of the M-codes of real numbers X_α, X_β accordingly. To do this *a precoder* should be used.
2. The decoding of real numbers (positive or negative) X_α, X_β from M-code to P-code. To do this *a full decoder of M-code into P-code* should be used.

3.7.3. Coding of Complex Number in System 2.

1. Computation of $\overline{X}_\beta = \mu \cdot X_\beta$ with $\mu = 1/\sqrt{2}$. This computation is performed in traditional binary coding system.
2. Coding of real (positive or negative) numbers X_α, $\overline{X}_\beta$ from P-code into M-code similarly to p.1 of the algorithm 3.6.1.
3. The C-code generation for a complex number $Z = X_\alpha + j\overline{X}_\beta$ similarly to p.2 of the algorithm 3.6.1.

3.7.4. Decoding of Complex Number in System 2.

1. The extraction from complex number's C-code $Z = X_\alpha + jX_\beta$ of even and odd-numbered digits according to the rule: $\left\{\begin{array}{l} \alpha_{m/2} = \gamma_m \text{ if } m - \text{even} \\ \beta_{(m-1)/2} = \gamma_m \text{ if } m - \text{odd} \end{array}\right\}$ and forming from them the digits α_m and β_m of the M-codes of real numbers X_α, X_β accordingly,

where. $\overline{X}_\beta = \mu \cdot X_\beta$ with $\mu = {}^1\!/\!\sqrt{2}$. To do this <u>*a precoder*</u> should be used.

2. Decoding real (positive or negative) numbers X_α, $\overline{X}_\beta$ from M-code into P-code similarly to p. 2 of the algorithm 3.6.2.
3. Computation of $X_\beta = \overline{X}_\beta \sqrt{2}$. This computation is performed in traditional binary coding system.

3.7.5. Coder of positive P-code into M-code - CoderPM.

This code converts the P-code of a **positive** number into M-code of this number. Its circuit is shown in the Fig. 3.7.5.1, where

N – digit capacity of the coder,
Meven – one-digit coding circuit for an even-numbered digit,
Modd - one-digit coding circuit for an odd-numbered digit,
A – input P-code,
C – output M-code.

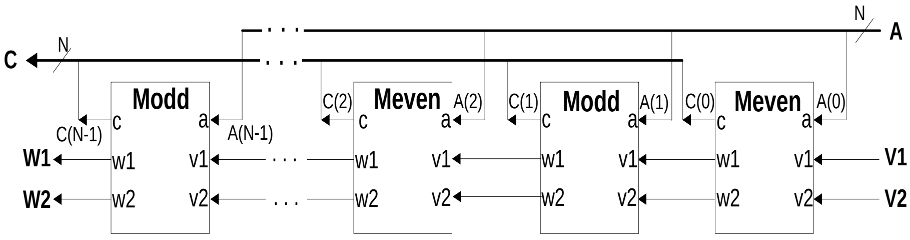


Fig. 3.7.5.1. Coder of positive P-code into M-code

Essentially the transformation consist in the following: the code including only odd-numbered P-code digits is subtracted from the code including only even-numbered P-code digits, and the subtraction is performed by the rules of M-codes subtraction. One-digit circuits *Meven* and *Modd* are shown in the Fig. 3.7.5.2. Their functioning is described by the verity tables 3.7.5.1 and 3.7.5.2 accordingly.

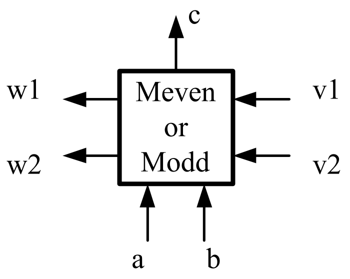


Fig. 3.7.5.2. One-digit coder circuit.

Table 3.7.5.1. One-digit coder circuit for even-numbered digit

a	v1	v2	w1	w2	c
0	0	0	0	0	0
1	0	0	0	0	1
0	0	1	0	0	1
1	0	1	1	1	0
0	1	1	0	1	1
1	1	1	0	0	0

Table 3.7.5.2. One-digit coder circuit for odd-numbered digit

a	v1	v2	w1	w2	c
0	0	0	0	0	0
1	0	0	0	1	1
0	0	1	0	0	1
1	0	1	0	0	0
0	1	1	0	1	1
1	1	1	0	1	0

3.7.6. Decoder of M-code into P-code – DecoderMP.

This decoder transforms the M-code of a number into P-code of this number. Its circuit is shown in Fig. 3.7.6.1, where

N - digit capacity of the decoder,

Deven – one-digit decoding circuit for an even-numbered digit,

Dodd – one-digit decoding circuit for an odd-numbered digit,

A – input M-code,

C – output P-code,

cop – operation code,
W – output carry.

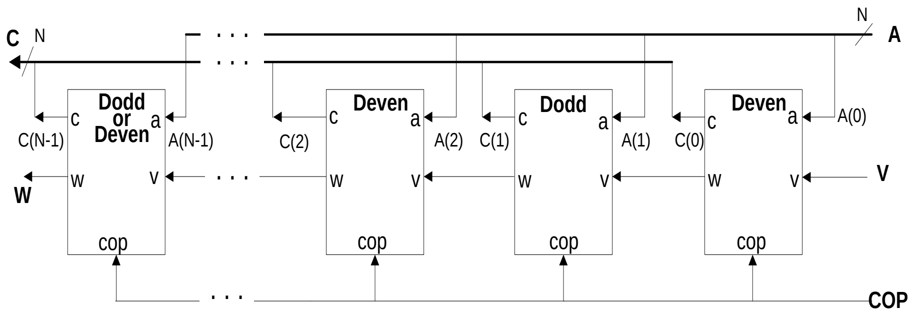


Fig. 3.7.6.1. Decoder of M-code into P-code

Essentially the transformation consist in the following:

- if the M-code is a *positive* number, then the code including only *odd-numbered* digits of M-code is subtracted from a code including only *even-numbered* digits of M-code (in this case cop=0),
- if the M-code is a negative number, then the code including only *even-numbered* digits of M-code is subtracted from a code including only *odd-numbered* digits of M-code (in this case cop=1), and the subtraction is performed according to the rule of P-codes subtraction.

One-digit circuits *Deven* and *Dodd* are shown in the Fig. 3.7.6.2. Their functioning is described by the verity tables 3.7.6.1 and 3.7.6.2 accordingly.

Table 3.7.6.1 computes $(\mathbf{-2w+c}) = \begin{cases} \mathbf{(a - v)}, \text{ if } \mathbf{cop} = \mathbf{0} \\ \mathbf{(-a - v)}, \text{ if } \mathbf{cop} = \mathbf{1} \end{cases}$

Table 3.7.6.2 computes $(\mathbf{-2w+c}) = \begin{cases} \mathbf{(-a - v)}, \text{ if } \mathbf{cop} = \mathbf{0} \\ \mathbf{(a - v)}, \text{ if } \mathbf{cop} = \mathbf{1} \end{cases}$

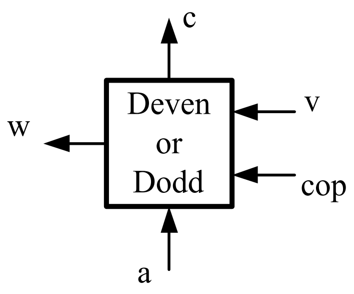


Fig. 3.7.6.2. One-digit decoder circuit.

Table 3.7.6.1. One-digit decoder circuit for even-numbered digit

	cop	a	v	w	c
Even - Odd	0	0	0	0	0
	0	1	0	0	1
	0	0	1	1	1
	0	1	1	0	0
Odd - Even	1	0	0	0	0
	1	1	0	1	1
	1	0	1	1	1
	1	1	1	1	0

Table 3.7.6.1. One-digit decoder circuit for odd-numbered digit

	cop	a	v	w	c
Even - Odd	0	0	0	0	0
	0	1	0	1	1
	0	0	1	1	1
	0	1	1	1	0
Odd - Even	1	0	0	0	0
	1	1	0	0	1
	1	0	1	1	1
	1	1	1	0	0

3.7.7. Full Decoder of M-code into P-code – mDecoderMP

The decoder DecoderMP is controlled by operation code *cop*. As a result full Decoder **mDecoder** must (beside DecoderMP) contain also the *sign determinant* of M-*code* – nSign. The unit's circuit will take the form presented in the Fig. 3.7.7.

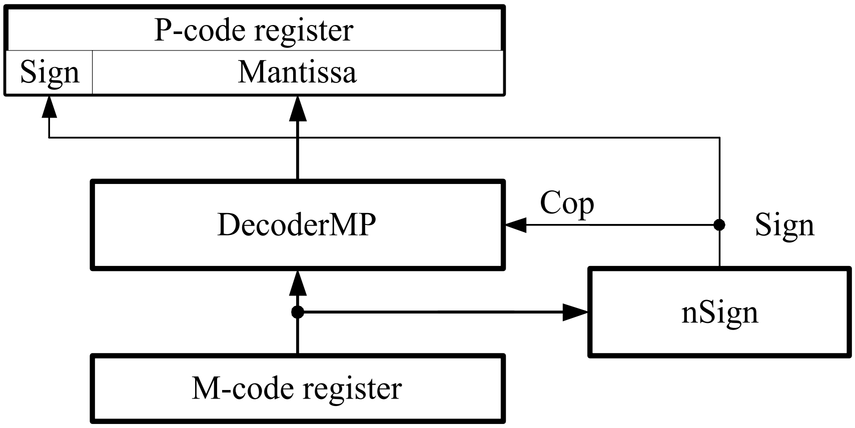


Fig. 3.7.7. Full decoder.

3.7.8. Precoder of P-code into M-code – PreCoder.

Precoder of P-code into M-code arranges the digits of P-code A in two groups – group A_{even} - of even-numbered digits (0, 2, 4, …) and group A_{odd} - of odd-numbered digits (1, 3, 5, …). In that way the coder кодер created from a code $\boldsymbol{A}$ two codes - A_{even} and A_{odd}. Notice that these codes later on will arrive on two inputs of M-codes algebraic adder. This adder computes or $\left(A_{even} - A_{odd}\right)$, or $\left(A_{odd} - A_{even}\right)$ depending on the value of control signal $s = \{0,1\}$ accordingly. The precoder is presented in Table 3.7.8, where for the code A *the* numbers of digits are indicated, and for codes A_{even} and A_{odd} the digits whose values are the same as of the corresponding digit of code A are indicated by "=". «**0**» is indicating a certain value of this code's digit's value.

Table 3.7.8. Precoder of P-code into M-code

A	…	7	6	5	4	3	2	1	0
A_{even}	…	0	=	0	=	0	=	0	=
A_{odd}	…	=	0	=	0	=	0	=	0

3.7.9. Partitioning Unit for Code's Parts – Partitioning.

The partitioning unit for code's parts transforms the real and imaginary parts of the code A into two M-codes of real numbers ReA and ImA. The imaginary part is transmitted with 1-digit shift to the left. The code partitioning unit is presented in Table 3.7.9, where for the code A the number of digits are indicated, and for the codes ReA and ImA the indicated numbers are the A code digit's numbers, which were moved to this digit in the course of partitioning

Table 3.7.9. Partitioning Unit for Code's Parts

A	…	7	6	5	4	3	2	1	0
Re A	…	14	12	10	8	6	4	2	0
Im A	…	15	13	11	9	7	5	3	1

4. Vector Processor

4.1. Data Representation and Vector Arithmetic Unit

Contrary to the ordinary data representation described in section 2.1 the coordinates of a point are presented as a code of point-vector. A simple vector arithmetic unit **VAU** operates with *p*–dimensional vectors. Such unit has to contain *p(n+r)*–digit multiplier, *p(n+r)*–digit adder, *p(n+r)*–digit coordinate register and *a*–digit parameters register. In this unit an affine transformation includes only one operation. VAU is shown in the Fig. 4.1.1. It is similar to the SAU unit, but, unlike the latter, contains a vector adder. All the transformation parameters are delivered simultaneously into the coordinate unit, in the transformation vector's code – see the description of operations with vector codes. In addition, coding and decoding units are provided in it.

Let us view the list of processor's commands realized in VAU:

- Receiving and coding the transformation parameters
- Receiving and coding all point's coordinates
- Adding to the carry vector
- Multiplying by the transformation matrix.
- Decoding and generating all coordinates
- Rounding off

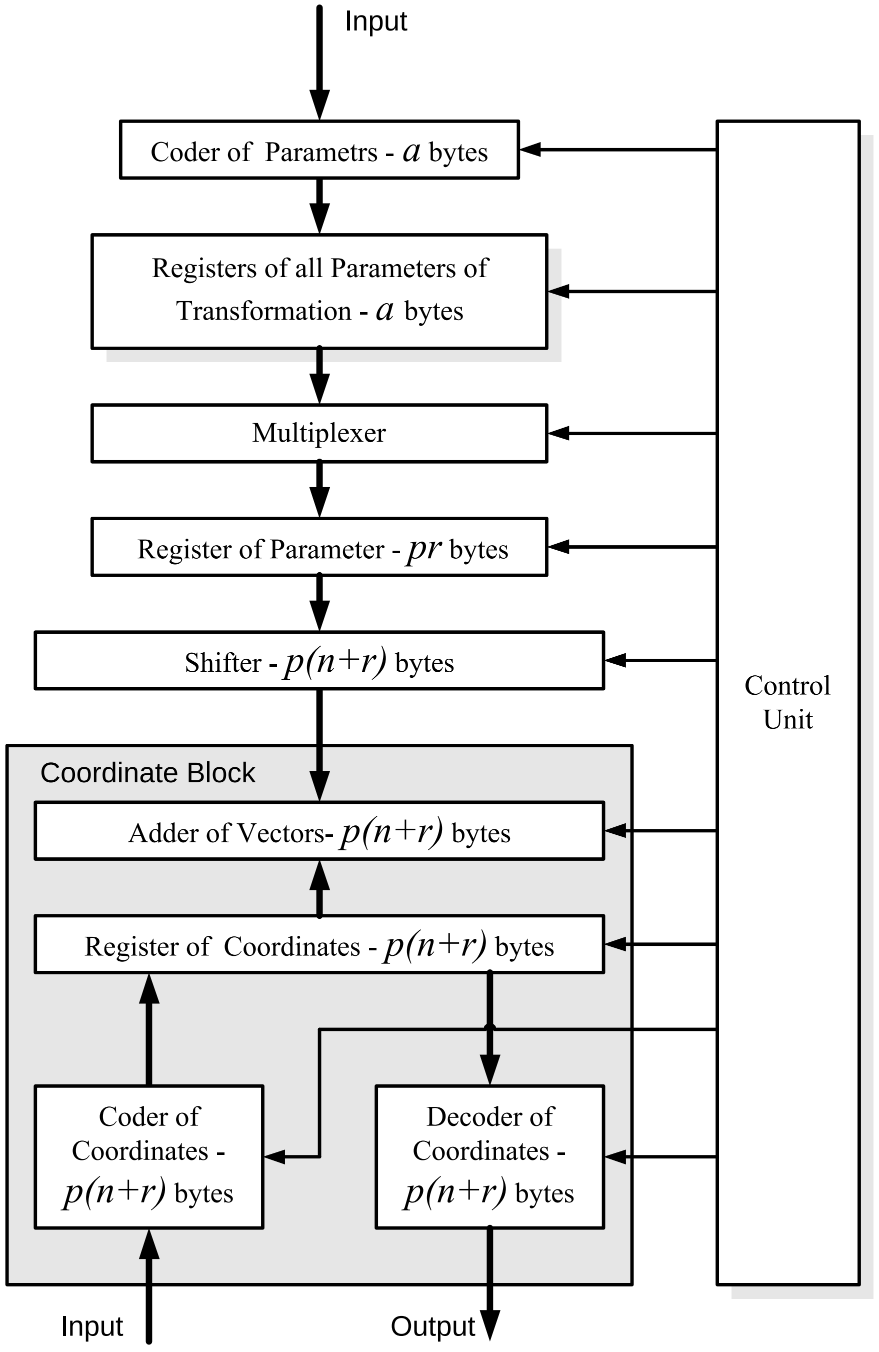

Input
Coder of Parametrs - a bytes
Registers of all Parameters of Transformation - a bytes
Multiplexer
Register of Parameter - pr bytes
Shifter - $p(n+r)$ bytes
Coordinate Block
Adder of Vectors- $p(n+r)$ bytes
Register of Coordinates - $p(n+r)$ bytes
Coder of Coordinates - $p(n+r)$ bytes
Decoder of Coordinates - $p(n+r)$ bytes
Control Unit
Input
Output

Fig. 4.1.1. Vector arithmetic unit

There is a way of constructing (by analogy with MSAU) an arithmetic unit **MVAU**, by using a set (*M*) of elementary units VAU operating simultaneously. In this unit the codes of all points-vectors of the figure comprise an array that will be called rectangular code of the vector - **RCV**. RCV contains *M* registers of digit capacity *p(n+r)*. MVAU as a whole contains *M* adders of digit capacity *p(n+r)*, *M* multipliers of digit capacity *p(n+r)*, RCV and one *a*–digit parameters register. This MVAU performs group operations - adding RCV to the carry vector and multiplying RCV by the transformation matrix.

Further, by analogy with FSAU, we may construct another version of AU that is intermediate between the AU with individual and group operations. For this purpose we must split RCV into several segments RCS_q, each containing *(Q)* registers of digit capacity *p(n+r)*. The arithmetic unit **FVAU** contains a total of *Q* adders of digit capacity *p(n+r), Q* multipliers of digit capacity *p(n+r)*, RCS_q and *a*–digit parameters register. The diagram of FVAU is shown in the Fig. 4.1.2. This diagram is identical with the diagram in the Fig. 4.1.1, except that its operational unit has *Q* coordinate units, detailed in the Fig. 4.1.1.

FVAU performs group operations with the point's coordinates of a figure fragment. In it the affine transformation of a figure has *Q* group multiplications and *Q* group additions.

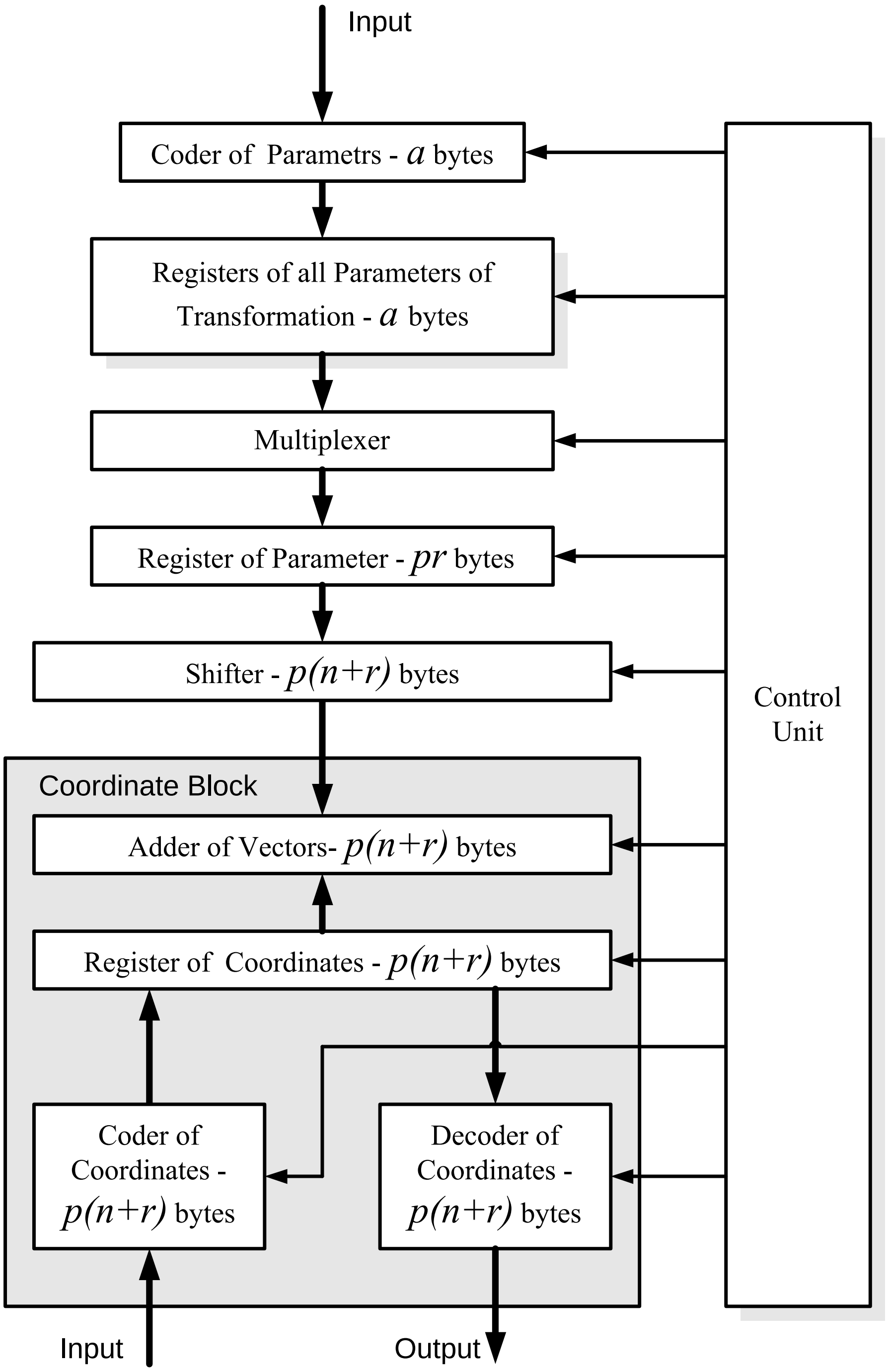

Input
Coder of Parametrs - a bytes
Registers of all Parameters of Transformation - a bytes
Multiplexer
Register of Parameter - pr bytes
Shifter - $p(n+r)$ bytes
Coordinate Block
Adder of Vectors- $p(n+r)$ bytes
Register of Coordinates - $p(n+r)$ bytes
Coder of Coordinates - $p(n+r)$ bytes
Decoder of Coordinates - $p(n+r)$ bytes
Control Unit
Input
Output

Fig. 4.1.2 Vector arithmetic unit with rectangular codes.

4.2. Comparisons

Table 4.2.1.a and 4.2.1.b review the comparative features of the units described above. In these tables:

- R – digit capacity of all registers;
- U – number of multipliers;
- A – adder's volume, measured in register's digits; it is assumed that the adder's volume is *three times* larger than that of the register.
- M – shifter's volume in multiplier, measured in register's digits; it is assumed that the shifter's volume is *twice* as large as the register's volume;
- $W=(R+A+M)$ – the volume of arithmetic unit, measured in register's digits;
- S – number of elementary operations of the given AU for an affine transformations of a whole figure.

Table 4.2.1a. Comparative features of AU.

#	AU	U	R	A	M
1	**SAU**	1	$2(n+r)+a$	$2(n+r)$	$3(n+r)$
2	**VAU**	1	$2p(n+r)+a$	$2p(n+r)$	$3p(n+r)$
3	**MSAU**	M	$2M(n+r)+a$	$2M(n+r)$	$3(n+r)M$
4	**MVAU**	Mp^2	$2Mp(n+r)+a$	$2Mp(n+r)$	$3p(n+r)M$
5	**FSAU**	Q	$2Q(n+r)+a$	$2Q(n+r)$	$3(n+r)Q$
6	**FVAU**	Qp^2	$2Qp(n+r)+a$	$2Qp(n+r)$	$3p(n+r)Q$

Table 4.2.1b. Comparative features of AU.

#	AU	U	W	S
1	**SAU**	1	$7(n+r)+a$	$M(D+p^2)$
2	**VAU**	1	$7p(n+r)+a$	M
3	**MSAU**	M	$7M(n+r)+a$	$D+p^2$
4	**MVAU**	Mp^2	$7Mp(n+r)+a$	1
5	**FSAU**	Q	$7Q(n+r)+a$	$(D+p^2)M/Q$
6	**FVAU**	Qp^2	$7Qp(n+r)+a$	M/Q

Tables 4.2.2, 4.2.3, 4.2.4 show the numerical comparative characteristics of the above named units with various values of n, r, p,

M, Q. This table is built on the base of Table 4.2.1. Besides, this table shows the quality of AU, measured as $H=W*S/M$, and also the value

$$h_k = {}^{H_k}\!/_{H_{k+1}},$$

which determines a comparative quality of AU operating with numbers as compared with AU operating with vectors.

Table 4.2.2. Numerical characteristics of AU with p=2, r=6, M= 10^6, n=12, Q=256, a=72.

АУ	**R**	**U**	**W**	**S**	**H**	**h**
1	126	1	198	$6*10^6$	1188	3.7
2	180	1	324	10^6	324	
3	$54*10^6$	10^6	$126*10^6$	6	756	3
4	$108*10^6$	10^6	$254*10^6$	1	254	
5	14000	256	32000	24000	768	3
6	28000	256	64000	4000	256	

Table 4.2.3. Numerical characteristics of AU with p=3, r=6, M=10^6, n= 12, Q=256, a=90.

АУ	**R**	**U**	**W**	**S**	**H**	**h**
1	144	1	216	$15*10^6$	3240	6.9
2	252	1	468	10^6	468	
3	$54*10^6$	10^6	$126*10^6$	15	1890	5
4	$162*10^6$	10^6	$378*10^6$	1	378	
5	14000	256	32000	60000	1920	5
6	42000	256	96000	4000	384	

Table 4.2.4. Numerical characteristics of AU with p=4, r=6, M=10^6, n= 12, Q=256, a=240.

АУ	**R**	**U**	**W**	**S**	**H**	**h**
1	292	1	366	$28*10^6$	10248	13.8
2	384	1	744	10^6	744	
3	$54*10^6$	10^6	$126*10^6$	28	3528	7
4	$216*10^6$	10^6	$504*10^6$	1	504	
5	14000	256	32000	112000	3584	7
6	56000	256	128000	4000	512	

From the above tables it follows that the quality of AU operating with vectors exceeds the quality of AU operating with numbers. The comparative quality increases h =3, 5, 7 times for p=2, 3, 4 and with $Q >> 1$. Comparative quality h increases with $Q \to 1$. It means that for a given volume of AU the performance of VAU, MVAU, FVAU increases h times as compared with SAU, MSAU, FSAU. It means also that for a given capacity of AU the volume of VAU, MVAU, FVAU decreases h times as compared with SAU, MSAU, FSAU. Therefore it is profitable to use vector arithmetic for the construction of geometrical processors.

To choose the optimal value of Q, one can minimize the criterion $\lambda = kW + S$, where k is a certain weight coefficient. The optimal value of FVAU is equal to $Q = \sqrt{\dfrac{M}{7ap(n+r)}}$

Specifically, if $a=0.05$, $M=10^6$, $r=6$, $n= 12$, $p =(2, 3, 4)$ then the optimal value is Q = (282, 230, 199).

In the further discussion we shall concentrate mainly on geometrical processors based on the arithmetic of geometrical codes. The described above unit FVAU, based on vector arithmetic, will be used for comparison.

5. Figure Coding Theory

5.1. Primary Geometrical Codes

5.1.1. Data Representation

Let us consider the binary tree shown in Fig. 5.1.1, and accord each of its vertexes a binary number *(i, k),* where *k* will be the tier number and *i* – the number of the vertex. Let us further assume that the tier numbering proceeds from left to right, and the numbering of vertexes – from top to bottom.

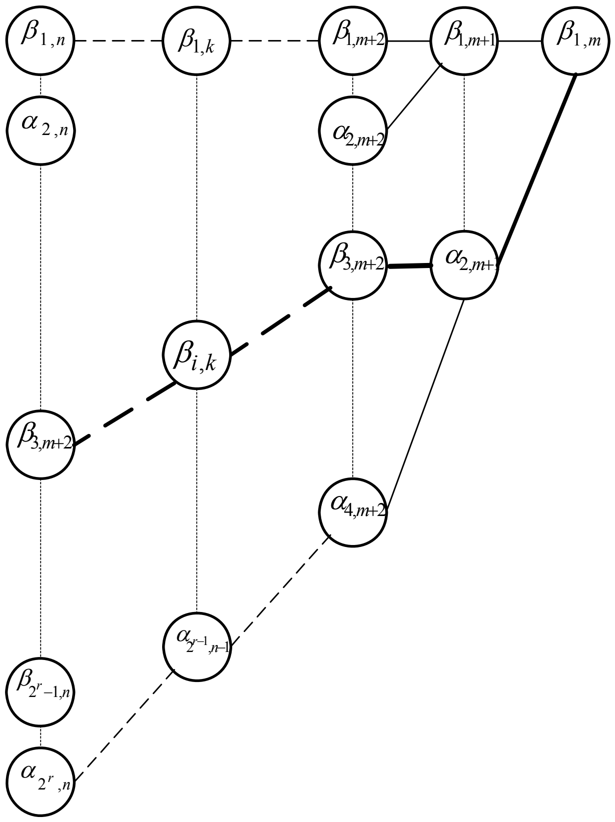


Fig. 5.1.1. Binary Tree.

Let *m* and *n* be the numbers of the extreme right and left tiers, respectively. Then *k=(n, n*-1,..., *m*+1, *m); i*=(1, 2,..., 2^{k-m}); the number of tiers *r=(n-m*+1); the number of tree nodes *u*=(2^r -1); the number of vertexes in the *n*-tier $N = 2^{r-1}$. Let us designate as $\alpha_{i,k}$ the vertex with even number *i* and as $\beta_{i,k}$ - the vertex with odd number *i*.

Let us call the path within the tree that connects vertexes $\beta_{1,m}$ and $\boxed{\quad}$ the ***p*-path**. Obviously, each *p*-path can be described by a series of symbols $\alpha_{i,k}$ and $\beta_{\iota,\kappa}$. For instance, the path emphasized in Fig. 5.1.1 is the *p*-path with the following corresponding series

$$\beta_{p,n} \cdots \beta_{i,k} \cdots \beta_{3,m+2}\, \alpha_{2,m+1}\, \beta_{1,m}.$$

Let us refer to each symbol α or β of a series depicting a certain *p*-path in the tree as ***k*-digit of the *p*-path** or **(*i, k*)-digit of the tree**. If we set, for each of the digits in the *p*-path, a corresponding 1 for an α-digit or a 0 for a β-digit, then the *p*-path can be represented by a binary code *K[p]*. In particular, for Fig. 5.1.1 *K[p]* = 0...0...010. The number of the *p*-path equals the number of the digit in the *n*-tier with which this path ends. Let us now agree that α and β are binary values, i.e. α = (0,1) and β = (0,1). Let us call the *p*-path **open** if the value of all its digits is 1, and **closed** if the value of at least one of its digits is 0. For illustration, Fig 5.1.2 shows a binary digit tree in which the paths are open (indicated in parentheses is the binary code that corresponds to the given path)

$$\alpha_{43}\, \alpha_{22}\, \beta_{11}\, \beta_{10} \qquad (K[4]=1100),$$
$$\beta_{53}\, \beta_{32}\, \alpha_{21}\, \beta_{10} \qquad (K[5]=0010),$$
$$\alpha_{63}\, \beta_{32}\, \alpha_{21}\, \beta_{10} \qquad (K[6]=1010),$$
$$\beta_{73}\, \alpha_{42}\, \alpha_{21}\, \beta_{10} \qquad (K[7]=0110),$$

or, in other words, this tree represents 4 binary codes. It should be noted that the open path depicted in the tree by 1-digits only has a corresponding binary code that includes, in the general case, 0-digits as well.

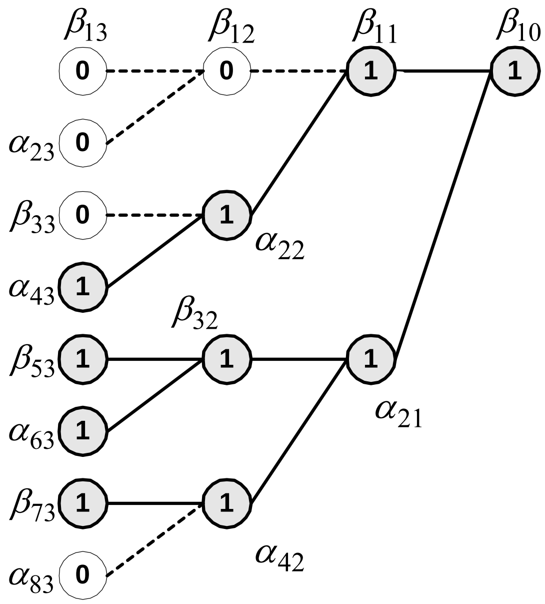


Fig. 5.1.2. Examples: a tree of binary digits

Let us refer to the binary digits tree thus constructed, which depicts a set of binary codes , as **primary geometrical code** **PGC** (in this section the adjective "primary" is going to be omitted and we shall talk about geometrical code **GC**), and the binary codes comprising it – as **linear codes**. Let us refer to the number of the digit in the senior tier of the geometrical code as the **address** of the corresponding linear code. The reduction of the number of binary digits when depicting *a* binary codes as geometrical code amounts to $g=ra/(2^{r+1}-2)$.

In particular, if all the tree's paths are open, then it depicts all *r*-digit binary codes. It follows from the above formula that the efficiency of the geometrical code increases in proportion to the value of *a*. However the main advantages of the geometrical code are that it is fairly simple to use for performing a variety of operations. Therefore, it is advisable to use geometrical code in those cases when there is a sufficiently large group of binary codes with which it is necessary to perform identical, **group operations**, for instance multiply all codes by one and the same number. In addition it is possible (as will be demonstrated further on) to use the geometrical code to depict random figures and interpret different transformations of these figures as operations with geometrical code.

5.1.2. Arithmetic operations with geometrical codes in a real radix

5.1.2.1. Introduction

Operations with geometrical codes reviewed below are, as a rule, equivalent to a certain logic or arithmetic operation involving the known, **basic**, binary code and each of the linear codes included in the set represented by the geometrical code. Besides, the operations are connected with the propagation of **carries** from the right-hand – lower tiers into the left-hand – upper tiers of the tree. Let us designate:

- $\beta_{i,k}$ - (*i, k*)-digit of geometrical code with *i* - odd;
- $\alpha_{i,k}$ - (*i, k*)- digit of geometrical code with *i* - even;
- $\pi_{i,k}$ - general carry to $\beta_{2i-1,k+1}$ and $\alpha_{2i,k+1}$ digits (*i*-odd*)*;
- $\mu_{i,k}$ - carry p from the $\beta_{i,k}$ digit;
- $\eta_{i,k}$ - carry p from the $\alpha_{i,k}$ digit;
- δ_k - *k*-digit of the basic code;
- $\tau_{i,k}$ - transposition signal of the code with the (*i, k*)-digit as its angle digit.

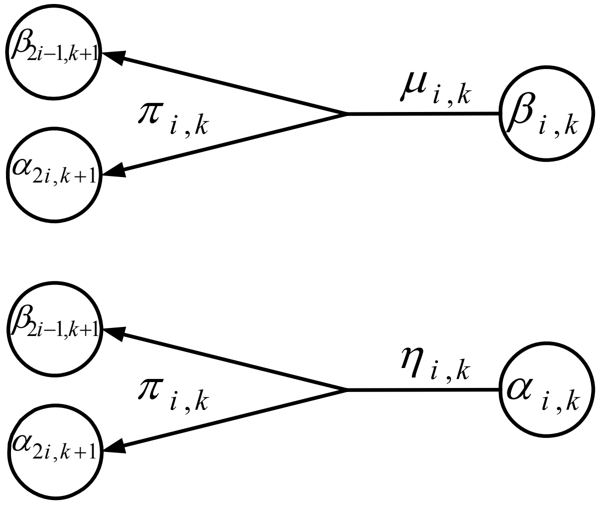


Fig. 5.1.2a. Carry propagation pattern in GC

The carry $\eta_{i,k}$ from the digit $\alpha_{i,k}$ or the carry $\mu_{i,k}$ from the digit $\beta_{i,k}$ arrives at the digits $\beta_{2i-1,k+1}$ and $\alpha_{2i,k+1}$ as the carry

$\pi_{i,k}$ according to the following pattern represented in Fig. 5.1.2 a. The transposition signal $\tau_{i,k}$ **precedes** the signals $\mu_{i,k}$ and $\eta_{i,k}$ that are the logical functions of the digit values $\beta_{2i-1,k+1}$ and $\alpha_{2i,k+1}$ obtained **after** the transposition.

The basic code and linear codes represented by the geometrical code can be viewed as binary codes with a ρ radix of a certain number or vector. In this case all digits of the geometrical code included in the *k*-tier must be accorded the weight of the *k*-digit of the linear code.

The number of linear codes comprising the geometrical code does not change during arithmetic operations.

5.1.2.2. Writing of Base Code

When the path is formed in GC with a linear code, equal to base code δ, the carry propagation process is determined by the following formulas:

$$\mu = \pi \wedge \overline{\delta}, \qquad \eta = \pi \wedge \delta.$$

For μ=1 the digit β assumes the value "1" regardless of its previous value. Similarly, for η=1 the digit α assumes the value "1".

5.1.2.3. Transposition

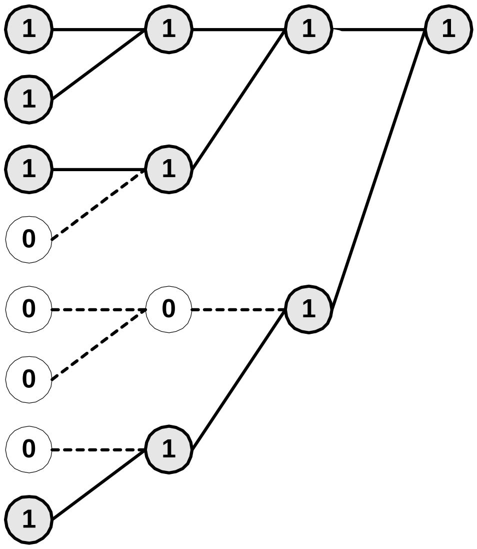


Fig. 5.1.3. Example: a transposed code.

Transposition of geometrical code will be the name of a transformation whereby the lower and upper halves of the geometrical code switch places. To be more precise, the digits of the initial code are connected with the digits of the transposed code (marked with a dash above the symbol) in the following manner:

$$\alpha_{i,k} = \overline{\alpha}_{j,k}, \quad \beta_{i,k} = \overline{\beta}_{j,k}, \quad j = rest(1+2^{k-i}) \bmod 2^{k-i+1}.$$

For instance, the code in Fig. 5.1.2 after transposition becomes the code in Fig. 5.1.3.

5.1.2.4. Addition of Geometrical and Basic Codes when ρ=2 is described in Table 1.1.1, wherefrom it follows that

$$\tau = \delta \oplus \pi, \quad \eta = \alpha \wedge \delta \wedge \pi, \quad \mu = (\delta \vee \pi) \wedge \beta,$$

Having distributed the indexes, we come up with the following formulae:

$$\tau_{i,k} = \delta_{k+1} \oplus \pi_{i,k}, \qquad (5.1.1)$$

$$\eta_{i,k} = \alpha_{i,k} \wedge \delta_{k} \wedge \pi_{i-1,k-1} \quad \text{with } i \text{ - even,} \qquad (5.1.2)$$

$$\mu_{i,k} = (\delta_{k} \vee \pi_{i,k-1}) \wedge \beta_{i,k} \quad \text{with } i \text{ - odd.} \qquad (5.1.3)$$

Table 5.1.1a. Addition of geometrical and basic codes with ρ=2

α	β	δ	π	τ	η	μ
0	0	0	0	0	0	0
0	1	0	0	0	0	0
1	0	0	0	0	0	0
1	1	0	0	0	0	0
0	0	0	1	1	0	0
0	1	0	1	1	0	1
1	0	0	1	1	0	0
1	1	0	1	1	0	1
0	0	1	0	1	0	0
0	1	1	0	1	0	1
1	0	1	0	1	0	0
1	1	1	0	1	0	1
0	0	1	1	0	0	0
0	1	1	1	0	0	1
1	0	1	1	0	1	0
1	1	1	1	0	1	1

Example 5.1.1 of addition when ρ=2. Let the basic code be K=<2> or K=10, and the geometrical code G depict a set of linear codes {1100, 0010, 1010, 0110} or, which is exactly the same thing, a set of numbers {12, 2, 10, 6}. Let us find the geometrical code $R=G+K$ – see Fig. 5.1.4. The process of carry proliferation stops. The resulting code $R=G_4$ depicts a set of codes {1110, 0010, 1100, 1000}, i.e. a set of numbers {14, 4, 12, 8}, which is what we needed to obtain. Thus, addition of geometrical and basic codes when ρ=2 is reduced to repeated transpositions.

	$m =$	3 2 1 0	numbers of digits
	$K =$	0 0 1 0	basic code
1)	$G_1 =$	0 0 1 1	$\pi_{10} = 0$
		0	
		0 1	$\tau_{10} = \delta_1 = 1$
		1	
		1 1 1	
		1	
		1 1	
		0	
2)	$G_2 =$	1 1 1 1	$\pi_{11} = (\delta_1 \vee \pi_{10}) \wedge \beta_{11} = 1$
		1	
		1 1	$\pi_{21} = \alpha_{21} \wedge \delta_1 \wedge \pi_{10} = 0$
		0	
		0 0 1	$\tau_{11} = \delta_2 \oplus \pi_{11} = 1$
		0	
		0 1	$\tau_{21} = \delta_2 \oplus \pi_{21} = 0$
		1	
3)	$G_3 =$	1 1 1 1	$\pi_{12} = 1$
		0	
		1 1	$\pi_{22} = \pi_{32} = \pi_{42} = 0$
		1	
		0 0 1	$\tau_{12} = 1$
		0	
		0 1	$\tau_{22} = \tau_{32} = \tau_{42} = 0$
		1	
4)	$G_4 =$	0 1 1 1	$\pi_{i,3} = 0$
		1	
		1 1	$\tau_{i,3} = 0$
		1	
		0 0 1	
		0	
		0 1	
		1	

Fig. 5.1.4. For example 5.1.1

5.1.2.5. Algebraic Addition of Geometrical and Basic Codes when ρ=2

This operation is possible only if the initial numbers are represented in the form of additional codes. In this case algebraic addition is described by the same equations. It is impossible to use inverse codes because there is no way to organize a chain of cyclical carries within the geometrical code.

5.1.2.6. Algebraic Addition of Geometrical and Basic Codes when ρ=-2

This operation consists of a repeated sequence of inversion (multyplication by '-1') operations and inverse addition (calculation according to the *c=-a-b* formula). The inverse addition operation is described in Table 5.1.1, wherefrom it follows that

$$\tau = \delta \oplus \pi, \quad \eta = \alpha \wedge (\delta \vee \overline{\pi}), \quad \mu = \beta \wedge \delta \wedge \overline{\pi}.$$

Table 5.1.1b. Inverse addition of geometrical and basic codes with ρ=-2

α	β	δ	π	τ	η	μ
0	0	0	0	0	0	0
0	1	0	0	0	0	1
1	0	0	0	0	0	0
1	1	0	0	0	0	1
0	0	0	1	1	0	0
0	1	0	1	1	0	0
1	0	0	1	1	0	0
1	1	0	1	1	0	0
0	0	1	0	1	0	0
0	1	1	0	1	0	1
1	0	1	0	1	1	0
1	1	1	0	1	1	1
0	0	1	1	0	0	0
0	1	1	1	0	0	1
1	0	1	1	0	0	0
1	1	1	1	0	0	1

Where δ = 0, those same formulae describe the inversion of geometrical code. Here we can also use the formulae in (5.1.1) and

$$\eta_{i,k} = \beta_{i,k} \wedge \delta_{k} \wedge \overline{\pi}_{i-1,k-1} \quad \text{with } i \text{ - even,} \quad (5.1.4)$$

$$\mu_{i,k} = (\delta_k \vee \overline{\pi}_{i,k-1}) \wedge \alpha_{i,k} \quad \text{with } \boldsymbol{i} \text{ - odd.} \qquad (5.1.5)$$

<u>Example 5.1.2 of inverse addition with ρ = -2.</u> Let the basic code be ***K*** = < 2 > or ***K*** = 110, and the geometrical code ***G*** depict a set of linear codes {0000, 0100, 0010, 0110} or, which is the same thing, a set of numbers { 0, 4, -2, 2 }. Let us find the geometrical code ***R = - G - K –*** see Fig. 5.1.5. The process of carry propagation stops. The resulting code ***R*** $= G_4$ depicts a set of codes { 0000, 1100, 0010, 1110 }, i.e. a set of numbers { 0, -4, -2, -6 }, which is what we needed to obtain. Thus the addition of geometrical and basic codes with ρ = -2 is reduced to repeated transpositions.

	$m =$	3 2 1 0	numbers of digits
	$K =$	0 0 1 0	basic code
1)	$G_1 =$	1 1 1 1	$\pi_{10} = 0$
		0	
		1 1	$\tau_{10} = \delta_1 = 1$
		0	
		1 1 1	
		0	
		1 1	
		0	
2)	$G_2 =$	1 1 1 1	$\pi_{11} = \beta_{11} \wedge \delta_1 \wedge \overline{\pi}_{10} = 1$
		0	
		1 1	$\pi_{21} = \alpha_{21} \wedge (\delta_1 \vee \overline{\pi}_{10}) = 1$
		0	
		1 1 1	$\tau_{11} = \delta_2 \oplus \pi_{11} = 0$
		0	
		1 1	$\tau_{21} = \delta_2 \oplus \pi_{21} = 0$
		0	
3)	$G_3 =$	1 1 1 1	$\pi_{12} = \pi_{32} = 0$
		0	
		1 1	$\pi_{22} = \pi_{42} = 1$
		0	
		1 1 1	$\tau_{12} = \tau_{32} = 0$
		0	
		1 1	$\tau_{22} = \tau_{42} = 1$
		0	
4)	$G_4 =$	1 1 1 1	$\pi_{i,3} = 0$
		0	
		0 1	$\tau_{i,3} = 0$
		1	
		1 1 1	
		0	
		0 1	
		1	

Fig. 5.1.5. For example 5.1.2.

5.1.2.7. Multiplication of Geometrical and Basic Codes

When describing this operation, let us limit ourselves to a case where the basic code is whole because the other case is easily reduced to this one by shifting the product. Thus let us suppose that the basic code is the multiplicand, and the geometrical code – the multiplier. The essence of multiplication is in replacing all digits $\alpha_{i,k} = 1$ of the multiplier with the basic code. In order to effect such substitution, it is necessary

- to identify within the geometrical code G the geometrical code $G_{i,k}$, in the lowest digit of which the $\alpha_{i,k} = 1$ digit and the remainder code G_0 are found;
- to add the $G_{i,k}$ code to the basic code, assuming that the vertex of the $G_{i,k}$ code lies in the <u>zero</u> tier – as a result of this operation, a certain code $G'_{i/2,k-1}$ is formed;
- superimpose the $G'_{i/2,k-1}$ code obtained in the previous step over the remainder code G_0.

Multiplication as a whole is a successive substitution of the multiplier's $\alpha_{i,k} = 1$ digits, which begins with the upper tier digits and proceeds from left to right. The carries during this addition propagate in the opposite direction and do not distort those multiplier digits that have not yet undergone the substitution process. Let us note that the $G_{i,k}$ code consists of (r,s)-digits of the code, where $s > k$, $i2^{r-k} \geq r \geq i(2^{r-k-1} + 1)$. For instance, if $\alpha_{i,k} = \alpha_{21}$, then

$$G_{i,k} = G_{21} = \begin{array}{lll} \dots\beta_{53} & \beta_{32} & 0 \\ \alpha_{63} & & \\ \beta_{73} & \alpha_{42} & \\ \alpha_{83} & & \end{array}$$

In order to eliminate overflow of the digit net, which is something that can occur during multiplication, it is necessary to make use of the rounding-off operation described below.

This method of multiplication is not applicable with ρ=2 if there are negative numbers among those numbers represented by the linear codes.

<u>Example 5.1.3 of multiplication with ρ = -2.</u> Let us find the product of the basic code *K*=<-2> or *K*=10 and the geometrical code – see Fig. 5.1.6.

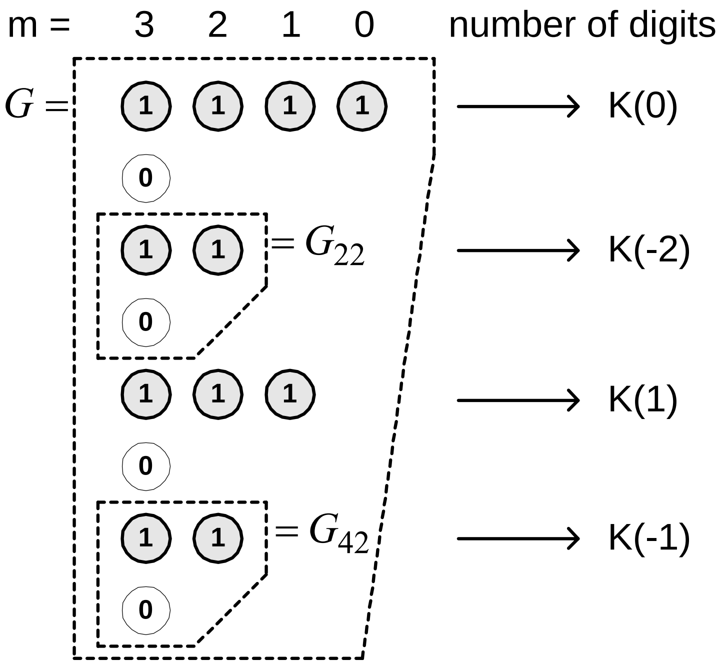


Fig. 5.1.6. For example 5.1.3.

In the third tier of code G there are no digits α = 1. Thus we proceed to analyze the second tier, where $\alpha_{22} = \alpha_{42} = 1$. We identify codes G_{22}, G_{42}, G'_0. Let us perform the addition of codes G_{22}, G_{42} and 10, as a result of which we obtain the codes G'_{11} and G'_{21}. Once we superimpose these codes onto the code G'_0, we obtain the code – see Fig. 5.1.7.

$G'_0 =$	$G'_{21} =$	$G'_{11} =$	$G' =$
1 1 1 1	0 1 1	0 1 1	1 1 1 1
0	1	1	1
0 0	0 0	0 0	0 0
0	0	0	0
1 1 1			1 1 1
0			1
0 0			0 0
0			0

Fig. 5.1.7. For example 5.1.3.

We then review the first tier of the resulting code G' and identify from it the codes – see Fig. 5.1.8.

$G''_{21} =$	$G''_0 =$
1 1 1	1 1 1 1
1	1
0 0	0 0
0	0
	0 0 0
	0
	0 0
	0

Fig. 5.1.8. For example 5.1.3.

Let us add the code G''_{21} with the code 10, and as a result we shall obtain the code G''_{11}. We then superimpose that code G''_{11} onto the code G''_0 and we obtain the final code G'' – see Fig. 5.1.9.

$G''_{11} =$
0 0 1 1
0
1 1
1
0 0 0
0
0 0
0

$G'' =$	→ K(…)
1 1 1 1	→ K(0)
1	→ K(4)
1 1	→ K(-2)
1	→ K(2)
0 0 0	
0	
0 0	
0	

Fig. 5.1.9. For example 5.1.3.

Thus $G'' = -2G$. It is easy to verify the correctness of this multiplication. Indeed, the code G describes the set of numbers {-2, -1, 0, 1}, and the code G'' - the set of numbers {-2, 0, 2, 4}, which is obtainable from the first by multiplying by '-2'.

Let us review a special case when the basic code contains «1» in the lowest digit. In this case the multiplication algorithm is simplified and consists of the following:

- identify within the geometrical code G the geometrical code $G_{i,k}$, in the lowest digit of which the digit $\alpha_{i,k} = 1$ is located;
- add the code $G_{i,k}$ to the basic code in which the *lowest digit has been zeroized*, assuming that the vertex of the code $G_{i,k}$ lies in the *zero* tier – as a result of this operation, a certain code $G'_{i,k}$ is formed;
- superimpose the code $G'_{i,k}$ obtained in the preceding step onto the code G.

Such an algorithm is equivalent to all codes $\{G_{i,k},\ i - \text{var}\}$ of the *k*-tier being added to the basic code in which the *lowest digit has been zeroized.* The carries in this case originate from the $\alpha_{i,k}$ digits, and this digit itself does not change its value since it is added to the zero digit of the basic code.

Example 5.1.3a of multiplication with ρ = -2. Let us find the product of the basic code *K*=<-1> or *K*=11 and the geometrical code – see Fig. 5.1.9a. In the third tier of the code G there are no digits $\alpha = 1$. Thus we shall proceed with the analysis of the second tier where $\alpha_{22} = \alpha_{42} = 1$. After the code G is added to the altered basic code 10, we obtain the code G' – see Fig. 5.1.9b.

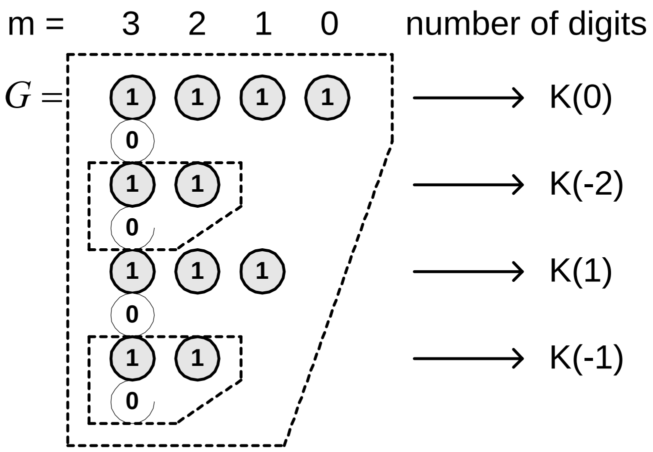


Fig. 5.1.9a. For example 5.1.3a.

$G'=$	→ K(…)
1 1 1 1	→ K(0)
0	
0 1	
1	→ K(2)
1 1 1	→ K(1)
0	
0 1	
1	→ K(3)

$G''=$	→ K(…)
1 1 1 1	→ K(0)
0	
0 1	
1	→ K(2)
1 1 1	→ K(1)
0	
1 1	→ K(-1)
0	

Fig. 5.1.9b. For example 5.1.3a.

In the first tier of this code $\alpha_{12}=1$. After the code G' is added to the altered basic code 10, we obtain the code G'' – see Fig. 5.1.9b. Thus $G''=-G$. This multiplication is easily verified. Indeed, the code G describes the set of numbers {-2, -1, 0, 1}, while the code G'' - the set of numbers {-1, 0, 1, 2}, which is obtainable from the first by multiplying by '-1'.

5.1.2.8. Division of the geometrical code by the basic code of a certain number is replaced by multiplication of the geometrical code by the basic code of an inverse number.

5.1.2.9. Rounding-off of the geometrical code

This operation containing r tiers consists in discarding the lowest tier. As a result, two codes are formed containing (r-1) tiers each. The operation ends by superimposing the resulting codes. Thus, as a result of superimposition, the lesser (or to be more precise, not the greater) number of linear codes of a shorter word length remains. This is due to the fact that when lower digits are discarded, equal linear codes may form, which are fixed in the resulting geometrical code as a single code.

5.1.3. Geometrical Codes in a Complex Radix.

Complex numbers can be used as the radix for coding linear codes. Similarly, attribute geometrical codes can be constructed in a complex radix. Unlike the preceding, the path value in such codes is the linear binary code with a complex radix. Such codes are described to part 3.1- see Table 3.1.1, where the existing systems of complex numbers are outlined. Further we are going to review arithmetic operations with geometrical codes in coding systems 1, 2 and 3. Therefore, in the subsequent detailing of codes the results of the previous section can be used. Some operations do not depend on the coding radix at all, and they will not be reviewed here.

5.1.3.1. Algebraic Addition of Geometrical and Basic Codes.

In the indicated systems, the binary code of a complex number can be viewed (when algebraic addition is being performed) as two codes of the parts Im and Re with a radix $\rho = -2$, their digits alternating. Due to that, operations of inverse addition are described by the same equations as with $\rho = -2$, but the carries $\mu_{i,k}$ and $\eta_{i,k}$ from the *k*-tier arrive not into two digits of the (*k*+1)-tier, but into four digits of the (*k*+2)-tier. Addition formulae in this case take on the following appearance:

$$\tau_{i,k} = \delta_{k+1} \oplus \overline{\pi}_{(i+1)/2,k-1} \quad \text{with } i \text{ - even,} \qquad (5.1.6)$$

$$\tau_{i,k} = \delta_{k+1} \oplus \overline{\pi}_{i/2,k-1} \quad \text{with } i \text{ - odd,} \qquad (5.1.7)$$

$$\eta_{i,k} = \beta_{i,k} \wedge \delta_{k} \wedge \overline{\pi}_{j,k-2} \quad \text{with } i \text{ - even,} \qquad (5.1.8)$$

$$\mu_{i,k} = (\delta_{k} \vee \overline{\pi}_{j,k-2}) \wedge \alpha_{i,k} \quad \text{with } i \text{ - odd,} \qquad (5.1.9)$$

where $j=1+z[(i-1)/4]$, and the function $z[x]$ – is a whole part of the argument x.

<u>Example 5.1.4 of inverse addition where</u> $\rho = j\sqrt{2}$ Let the basic code be $\boldsymbol{K}$ = < $j\sqrt{2}$ > = 10, and the codes of the numbers 0 and $j\sqrt{2}$ be represented by the geometrical code *G*. Let us find the geometrical code *R*=-*G*-*K* – see Fig. 5.1.10.

m =	3 2 1 0	numbers of digits
$\boldsymbol{K}$ =	0 0 1 0	basic code
1) $G = G_1$ =	1 1 1 1	$\pi_{10} = 0$
	0	
	0 0	$\tau_{10} = \delta_1 = 1$
	0	
	1 1 1	
	0	
	0 0	
	0	
2) G_2 =	1 1 1 1	$\pi_{11} = \beta_{11} \wedge \delta_1 = 1$
	0	
	0 0	$\pi_{21} = \alpha_{21} \wedge \delta_1 = 1$
	0	
	1 1 1	$\tau_{11} = \delta_2 = 0$
	0	
	0 0	$\tau_{21} = \delta_2 = 0$
	0	
3) G_3 =	1 1 1 1	$\pi_{12} = \pi_{22} = \pi_{32} = \pi_{42} = 0$
	0	
	0 0	
	0	
	1 1 1	$\tau_{12} = \tau_{22} = \tau_{32} = \tau_{42} = 1$
	0	
	0 0	
	0	
4) G_4 =	0 1 1 1	$\pi_{i,3} = 0$
	1	
	0 0	$\tau_{i,3} = 0$
	0	
	0 1 1	
	1	
	0 0	
	0	

Fig. 5.1.10. For example 5.1.4.

The process of carry proliferation stops. The resulting code $R = G_4$ describes the codes 1010 and 1000 of the numbers $(-j\sqrt{2})$ and

$(-2j\sqrt{2})$, respectively. Thus with $\rho = j\sqrt{2}$ the process of addition is reduced to repeated transpositions as well.

Thus, multiplication of GC by the basic code consists in adding the fragments of the GC to the basic code. Where the radix is (-2), such addition consists of inverse addition and inversion of the resulting fragment.

5.1.3.2. Multiplication of geometrical and basic codes is performed as described above for a random radix. However, in this case, certain modifications of this operation are also possible in addition to that:

- multiplication of the real part of geometrical code (even-numbered tiers) by basic code, which involves the substitution of only those digits $\alpha_{i,k} = 1$ that belong to the real parts of linear codes;
- multiplication of the imaginary part of geometrical code (odd-numbered tiers) by basic code, which is performed similarly to the preceding;
- multiplication of the real and imaginary parts of geometrical code simultaneously by different basic codes.

Another difference has to do only with coding system 1, and it is as follows. In even-numbered tiers, the $\alpha_{i,k} = 1$ digits are substituted by linear codes of the complex number Z as described above for the general case. In odd-numbered tiers, the $\alpha_{i,k} = 1$ digits are substituted by linear codes of the complex number jZ as described above for the general case.

Example 5.1.5 of multiplication of the imaginary part of geometrical code with $\rho = j\sqrt{2}$ The basic code, $K = <1 + j\sqrt{2}>$ or K=11, and the geometrical code G are known – see Fig. 5.1.11.

m = 3 2 1 0 -1	m = 3 2 1 0 -1
G = 1 1 1 1 1	G'_0 = 1 1 1 1 1
0	0
0 0	0 0
0	0
1 1 1	1 1 1
0	0
0 1	0 1
1	0
0 0 1 1	0 0 1 1
0	0
1 1	1 1
0	0
1 1 1	1 1 1
0	0
0 1	0 1
1	0

Fig. 5.1.11. For example 5.1.5.

Let us find the geometrical code *R=K(*Im*G)*. First we analyze the upper odd (third) tier of the code *G*, and we find that $\alpha_{83} = \alpha_{163} = 1$. We identify the codes $G_{83} = G_{163} = 1$ and G'_0, then we perform the addition of codes G_{83} and *K*, and as a result we obtain these codes:

```
G'_73 = G'_153   0 0 1
                 0
                 0 1
                 1
```

Let us superimpose G'_0, G'_{73}, and G'_{153}, and obtain the code G' – see Fig. 5.1.12. By passing the second tier (since we are only multiplying the imaginary part), we analyze the first tier of the code G', and we notice that $\alpha_{21} = \alpha_{41} = 1$. We identify the codes G'_{21}, G'_{41}, and G''_0. Then we add the codes G'_{21} and G'_{41} to the code *K*, and as a result we obtain the codes G''_{10} and G''_{20}. We then superimpose those codes onto the code G''_0, and we obtain finally the code R – see Fig. 5.1.13. A geometrical interpretation of this example will be provided further on.

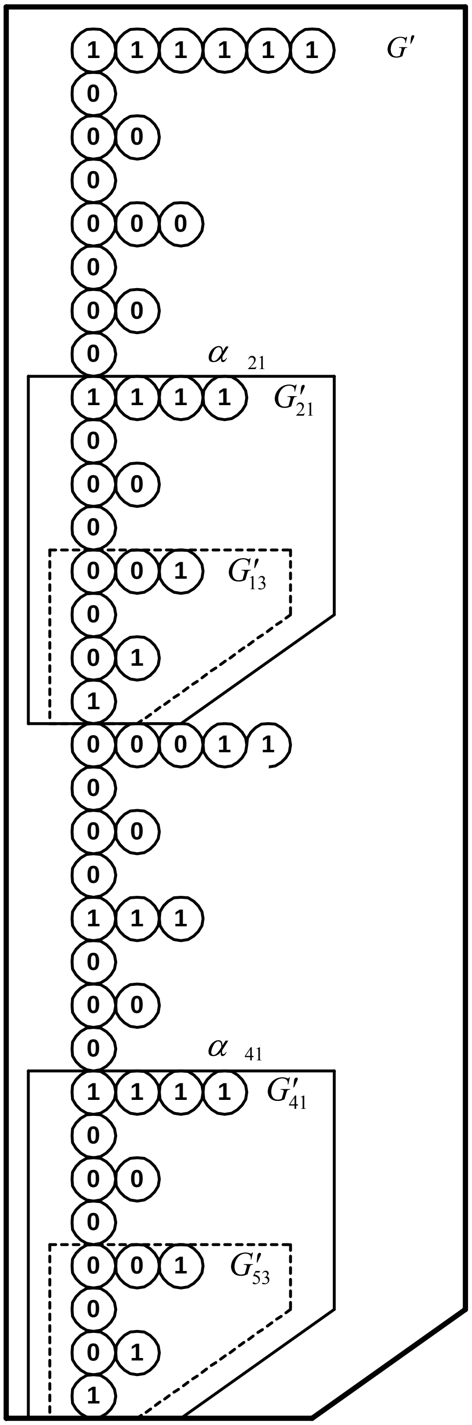


Fig. 5.1.12. For example 5.1.5.

$G''_0 =$	$G''_{10} =$	$G''_{20} =$	$R =$
1 1 1 1 1	0 0 0 1	0 0 0 1	1 1 1 1 1
0	0	0	0
0 0	0 0	0 0	0 0
0	0	0	0
0 0 0	0 1 1	0 1 1	0 1 1
0	1	1	1
0 0	1 1	1 1	1 1
0	0	0	0
0 0 1 1			0 0 1 1
0			0
1 1			1 1
0			0
0 0 0			0 1 1
0			1
0 0			1 1
0			0

Fig. 5.1.13. For example 5.1.5.

5.1.4. Coding and Transformation of Planar Figures

5.1.4.1. Method of coding

When coding planar figures, we are going to assume that

- identified on a plane are N points distributed evenly with step of Δx along the x axis and Δy along the y axis;
- each point can only be ascribed one of two values - 0 or 1;
- the shape is determined by a subset of points a, which are ascribed the value of one.

A trivial method of coding the figure might have been to determine pairs of coordinates x and y for all points, or the codes of complex numbers $x+jy$ corresponding to these points. Then the different transformations of the figure would have involved computations with complex numbers according to a certain routine. However the set a of binary codes of complex numbers can be represented by geometrical code. Such coding, first of all, requires less memory and, second, geometrical transformations of figures are easily interpreted as operations with geometrical codes.

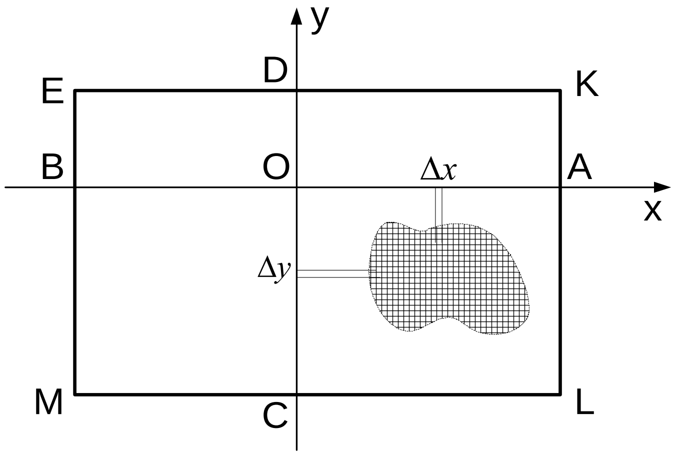


Fig 5.1.14. Coding planar figure.

The section of the plane being coded by geometrical code has the appearance of a rectangle EKLM, the sides of which pass through points A, B, C, and D perpendicular to the axes – see Fig. 5.1.14. $N = 2^r$ points are set out in the area EKLM, each of them corresponding to one of the complex numbers' linear codes, joined into a geometrical code. Distances between these points are determined by the values Δx and Δy, which depend on m and ρ: if $(m+1)$ is an even number or 0, then $\Delta x = |\rho|^{m+1}$ and $\Delta y = \Delta x\,|\rho|$, otherwise $\Delta y = |\rho|^{m+1}$ and $\Delta x = \Delta y\,|\rho|$ The size and position of the area being coded depends on Δx, Δy, n.

<u>**Example 5.1.6 of coding a plane with**</u> $\rho = j\sqrt{2}$ Let m=-1, n=3, r=n-m+1=5. The geometrical code for this case is shown in Table 5.1.2, which lists the linear codes corresponding to the paths in the geometrical code tree (it is assumed that all paths are open), and the values of complex numbers represented by these codes. In this table (and also in the next table 5.1.2) we shall use the following notations:

N – number of a point,
Z - value (a complex number) of this point,
L – code of this complex point – linear code
G – geometrical code.

Fig. 5.1.15 shows the points on the complex plane that correspond to these complex numbers. The section of the plane that is being coded by this geometrical code was thereby constructed.

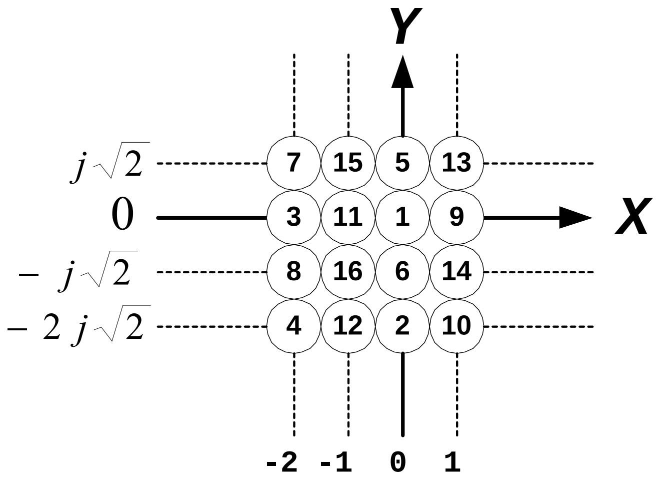


Fig 5.1.15. Coding plane at y=3, m=-1, r=4 for example 5.1.6.

Table 5.1.2. Geometrical code of a plane with $\rho = j\sqrt{2}$.

N	Z	L	G
1	**0+0**	**0000**	**11111**
2	**0-2j**$\sqrt{2}$	**1000**	**1**
3	**-2+0**	**0100**	**11**
4	**-2-2j**$\sqrt{2}$	**1100**	**1**
5	**0+j**$\sqrt{2}$	**0010**	**111**
6	**0-j**$\sqrt{2}$	**1010**	**1**
7	**-2+j**$\sqrt{2}$	**0110**	**11**
8	**-2-j**$\sqrt{2}$	**1110**	**1**
9	**1+0**	**0001**	**1111**
10	**1-2j**$\sqrt{2}$	**1001**	**1**
11	**-1+0**	**0101**	**11**
12	**-1-2j**$\sqrt{2}$	**1101**	**1**
13	**1+j**$\sqrt{2}$	**0011**	**111**
14	**1-j**$\sqrt{2}$	**1011**	**1**
15	**-1+j**$\sqrt{2}$	**0111**	**11**
16	**-1-j**$\sqrt{2}$	**1111**	**1**

Let us separate in this section of the plane the "black" (visible) and "white" (invisible) points – see Fig. 5.1.15a. The geometrical code will be on a form, presented in the Fig. 5.1.15b. The same code is described in the Table 5.1.2a, where (unlike the Table 5.1.2) the invisible points are shown without their coordinates

Table 5.1.2a. Geometrical code of a separated plane points with $\rho = j\sqrt{2}$

N	Z	L	G
1	**0+0**	**0000**	**11111**
2			**0**
3	**-2+0**	**0100**	**11**
4			**0**
5			**001**
6			**0**
7			**01**
8	**-2-j**$\sqrt{2}$	**1110**	**1**
9			**0111**
10	**1-2j**$\sqrt{2}$	**1001**	**1**
11			**01**
12	**-1-2j**$\sqrt{2}$	**1101**	**1**
13	**1+j**$\sqrt{2}$	**0011**	**111**
14			**0**
15	**-1+j**$\sqrt{2}$	**0111**	**11**
16			**0**

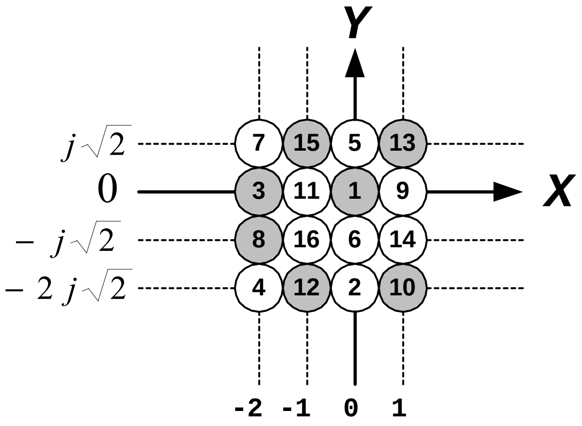


Fig 5.1.15a. Example: planar figure.

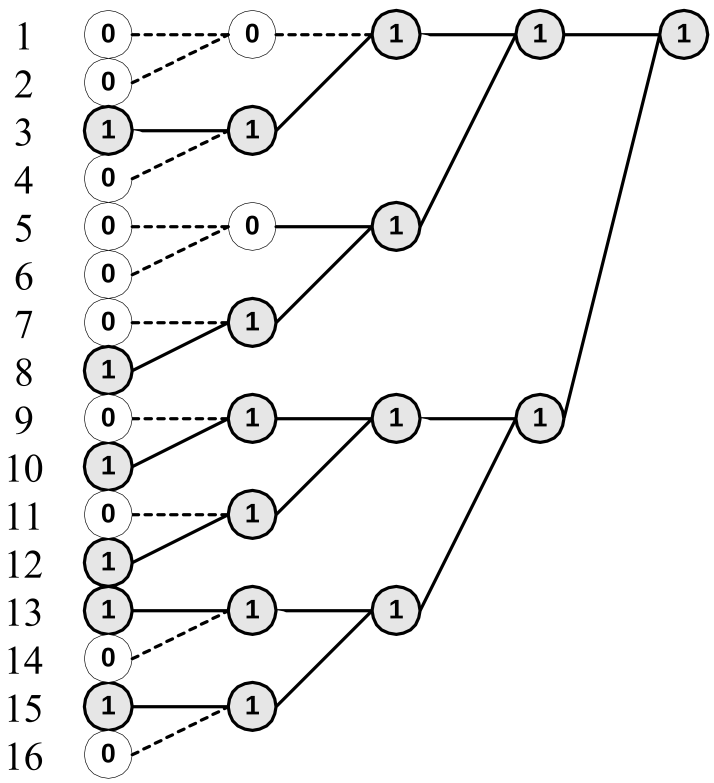


Fig 5.1.15b. Example: GC tree of planar figure.

Let *t=x+jy* be a random point on the plane, represented by a linear code within the geometrical code with a complex radix, and *b=|b|* $e^{j\varphi}$ *=(c+jd)* – a complex number represented by the basic code with the same radix. Let us review those geometrical transformations that are equivalent to arithmetic operations between the numbers *t* and *b*.

5.1.4.2. Carry

The carry of figures along the ray $e^{j\varphi}$ by *|b|* units is equivalent to the *t+b* operation, i.e. addition of the geometrical and basic codes.

5.1.4.3. Centroaffine transformation

Centroaffine transformation corresponds to multiplication of the real and imaginary parts of the geometrical code simultaneously by different basic codes *(c+jd) and (g+jh) (component-by-component multiplication)*. This operation is described by the formula *z+jv=x(c+jd)+jy(g+jh)* – here the point (*x, y*) changes into the point (*z, v*). In particular cases, centroaffine transformation becomes *a turn, a widening, a shift* (but not the carry reviewed earlier) or some combination of those transformations.

5.1.4.4. Affine transformation

Affine transformation is the product of centroafine transformation and the carry, and is performed in two stages:

1. component-by-component multiplication of the geometrical code by a pair of basic codes of the centroaffine transformation,
2. additon of the geometrical code resulting from the previous operation to the basic code of the carry.

<u>**Example 5.1.7 of deformation with the**</u> $\rho = j\sqrt{2}$ – see Fig. 5.1.16 and Table 5.1.3. Here we denote:

i – number of a point,

a_i - a point of the initial figure,

b_i -a point of transformed figure,

$L(a_i)$ - linear code of the point a_i ,

$L(b_i)$ - linear code of the point b_i .

Let us review a figure determined by 6 points a_i. Let us deform this figure in such a way, that the points $a_i = (x_i + jy_i)$ would change into the points b_i, noting that $b_i = (x_i + jy_i(1 + j\sqrt{2}))$. This deformation is equivalent to shifting the figure horizontally by an angle of $\Psi = 55^0$ $(tg\Psi = \sqrt{2})$. All number codes a_i are depicted by a single geometrical code *G* of the initial figure.

The above deformation of this figure is equivalent to multiplying the imaginary part of the geometrical code ***G*** by the basic code *K*=0011 of the number $(1 + j\sqrt{2})$. Such multiplication was performed in example 5.1.5. As a result, geometrical code ***R*** is formed. Decoding the code ***R***, we find that it combines the linear codes of b_i points of the deformed figure.

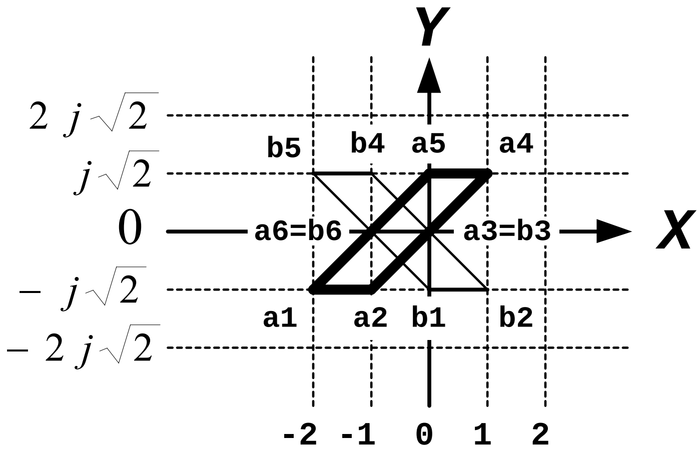


Fig. 5.1.16. Deformation of a figure for example 5.1.7

Table 5.1.3. Figure deformation with $\rho = j\sqrt{2}$

N	a_i	$L(a_i)$	b_i	$L(b_i)$
1	**2-j$\sqrt{2}$**	**1110**	**0-j$\sqrt{2}$**	**1010**
2	**-1-j$\sqrt{2}$**	**1111**	**1-j$\sqrt{2}$**	**1011**
3	**0+j0**	**0000**	**0+j0**	**0000**
4	**1+j$\sqrt{2}$**	**0011**	**-1+j$\sqrt{2}$**	**0111**
5	**0+j$\sqrt{2}$**	**0010**	**-2+j$\sqrt{2}$**	**0110**
6	**-1+0**	**0101**	**-1+0**	**0101**

5.1.5. Coding and Transformation of Spatial Figure

In this section we shall assume that linear codes to the radix $\rho = j\sqrt[3]{2}$ are used in the creation of geometrical code. Arithmetic operations with geometrical codes to this radix are in many respects similar to operations with geometrical codes to the radix $\rho = j\sqrt{2}$. The distinction is in the following:

- vector codes addition is equivalent to the addition of *three* components,
- componentwise multiplication of geometrical and base codes includes multiplication of each of the *three* linear code components by different base codes.

When coding three-dimensional figures we shall assume that

- we have singled out N points in three-dimensional space, distributed uniformly with step Δx, Δy, Δz on the Cartesian coordinate system's axes,
- each point may assume one of the values - 0 or 1;
- a figure is defined by a subset a of points assuming the value 1;

If m is the number of lowest tier, n – the number of highest tier, and $r=(n-m+1)$ – the number of all geometrical code's tiers, then

- the number of points singled out in the coded part of the space, is $N = 2^r$;
- for $(m+1)=3t$ (t-an integer number) the steps by coordinate axes are $\Delta x = |\rho \quad|^{m+1}, \Delta y = \Delta x|\rho \quad|, \Delta z = \Delta y|\rho \quad|$;
- the coded part of the space is a parallelepiped, with faces parallel to coordinate planes.

Let U_1 be a vector of an arbitrary point within a three-dimensional figure. Then by analogy with the above said, we have that:

- the addition of geometrical code to the base code U_5 is equivalent to addition of codes U and U_5, that is, to a *carry* of the figure by vector U_5;
- componentwise multiplication of a geometrical code by ordered three vectors U_2, U_3, U_4 is equivalent to a similar operation

with each vector U_1 and consists in computing vector U by the formula $U = x_1 i * U_2 + y_1 j * U_3 + z_1 k * U_4$, which means that it is equivalent to a *centroaffine transformation*.

An *affine transformation* is a product of a centroaffine transformation and a carry, and is performed in two stages:

1) componentwise multiplication of the geometrical code by three base codes of centroaffine transformation;
2) addition of the geometrical code that is the result of preceding operation, to the base carry code.

In particular cases this transformation is equivalent to a carry, a turn, a compression, a shift of a figure, or a vector multiplication of all the figure’s vectors by the base vector, or to other transformations of a figure.

In the same way as the codes of three-dimensional figures, geometrical codes of many-dimensional figures may be created, because, as indicated above, in the ring of many-dimensional vectors there also exists a positional system of binary codes. It means that a many-dimensional figure may be also coded by geometrical code, and affine transformations of this figure may be performed with this code. This fact is convenient to use, for instance, for creating pattern recognition devices, because the recognizable objects’ features are often invariant to certain types of geometrical transformations.

5.2. Attribute Geometrical Codes

5.2.1. Data Representation

It follows from the preceding that operations with PGC require multiple transpositions. The schemes that perform the transpositions must have a large volume since during transposition each PGC digit can switch places with any digit in its tier.

Let us review a PGC modification that is free of this shortcoming, and refer to it as **attribute GC – AGC** (in this section, the adjective "attribute" will be omitted if it is clear from the context that reference is not being made to primary GC). For each pair of digits $\beta_{2i-1,k}$ and $\alpha_{2i,k}$, there is an additional digit, $\gamma_{i,k}$ included in AGC, which is a modulus 2 counter of transposition signals $\tau_{i,k}$. When the $\gamma_{i,k}$ digits have a zero value (there was no transposition or it was performed an even number of times), the PGC and AGC codes are identical: each path in the geometrical code tree has a corresponding linear code wherein a "1" stands in place of an α-digit, and a "0" – in place of the β-digit. However, when the values of the digits $\gamma_{i,k}$ =(0,1), the linear code corresponding to this path in the AGC tree is determined as follows: in place of the $\alpha_{2i,k}$-digit stands the value $\bar{\gamma}_{i,k}$, and in place of the $\beta_{2i-1,k}$-digit – the value $\gamma_{i,k}$. Let us remind you that the values of the digits α and β determine only that the path in the geometrical code tree is open (or closed), and that the corresponding linear code is included (or not included) in the set of linear codes being coded.

The principal difference between PGC and AGC is in the following. Let a certain vector X have a corresponding p-path. During an operation with geometrical code, the value of this vector changes to Y. In PGC, after this operation is performed, q-path is going to correspond to this vector. Thus the *vector's position* in PGC depends upon its value. Not so in AGC: a given vector always has one and the same path that corresponds to it. One could say that a vector (and its corresponding point in space) *retains its individuality* regardless of the changing value of this vector (the position of the point). In this case the point can be accorded an attribute, which can be a name, a color, a weight, etc. This attribute must be connected with the terminal vertex of the same path where the linear code of the given vector (point) is written.

Let us consider the above statement more formally. A certain open path in the AGC tree has corresponding series of digits α=1, β=1, γ=(0,1):

$$\alpha_{p,n} \cdots \quad \beta_{2j-1,q} \cdots \quad \alpha_{2i,k} \cdots \quad \beta_{1,m}$$

This path depicts the linear code,

$$\bar{\gamma}_{p/2,n} \cdots \quad \gamma_{j,q} \cdots \quad \bar{\gamma}_{i,k},$$

that we are going to refer to as the value code, or simply the *value of the given path*. In this path, the pair of digits $\alpha_{2i,k}$ and $\gamma_{i,k}$ is represented by the value digit $\bar{\gamma}_{i,k}$, and the pair of digits $\beta_{2i-1,k}$ and $\gamma_{i,k}$ is represented by the value digit $\gamma_{i,k}$.

Where $\gamma \equiv 0$ for all digits of the given path, the linear code assumes the value

1 ... 0 ... 1 ... 0,

which we are going to refer to as the number code, or simply the *number of the given path*. In this code, "1" stands in place of the α-digit, and "0" – in place of the β-digit. Thus in AGC each path has a number, a value, and an attribute.

AGC is presented in Fig. 5.2.1. The number of digits in AGC is determined according to the following formula:

$$V = 1 + 3\sum_{k=0}^{n-m} 2^k = 3 \cdot 2^{n-m} - 2 \cdot$$

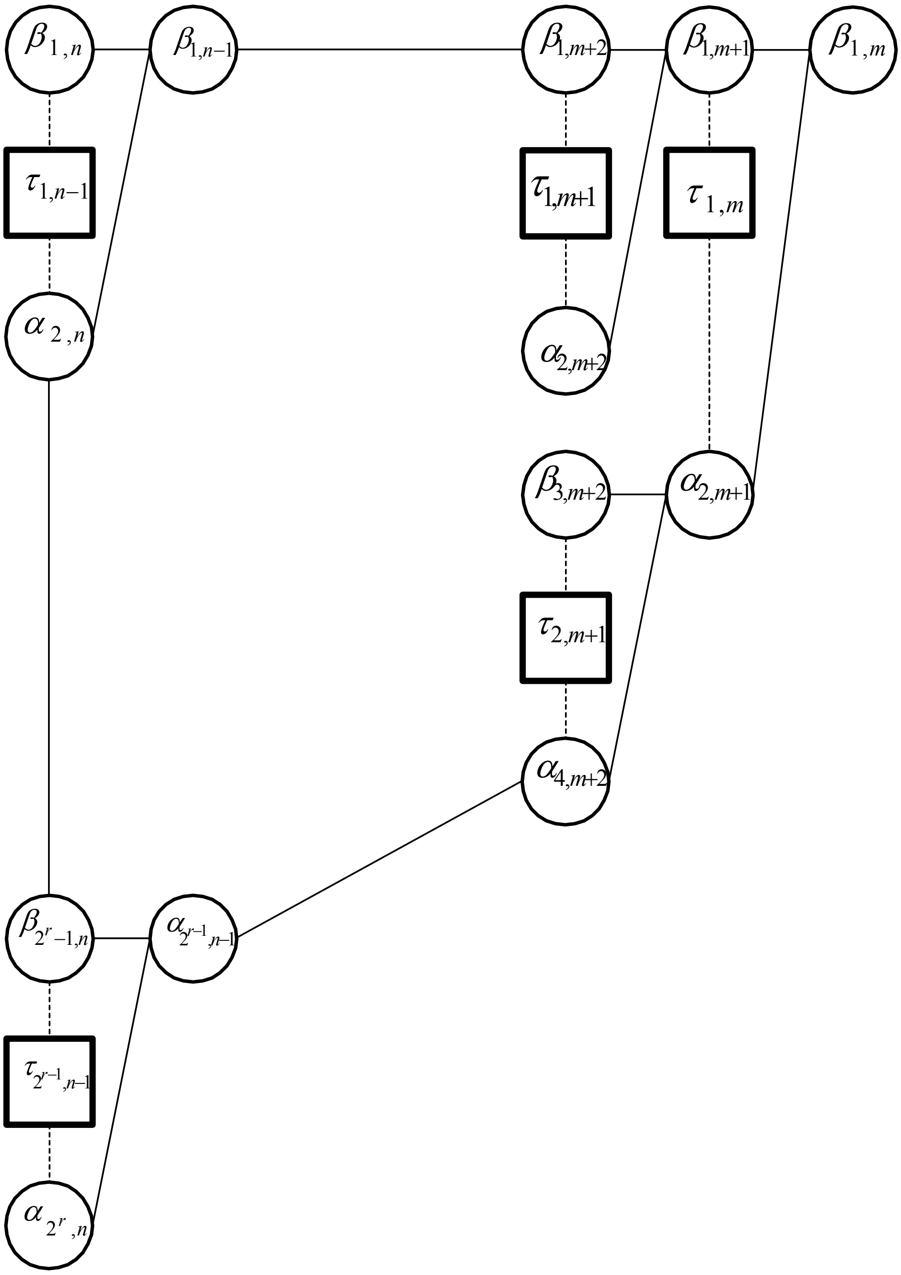


Fig. 5.2.1. Attribute Geometrical Code.

5.2.2. AGC in a Real Radix

Let us review the operations between the basic code and SGC with a real radix. While doing that, we shall use the designations from section 5.1.2.1.

5.2.2.1. Writing of a given Number

In this case a path is created in AGC with a number equal to the base code δ. <u>If all the digits $\gamma = 0$, the code of the value is coincident with the code of the number.</u> The process is determined by the following formulas:

$$\mu = \pi \wedge \overline{\delta}, \qquad \eta = \pi \wedge \delta.$$

Where μ=1, the β digit takes on the value of “1” regardless of its former value. Similarly, where η=1, the α digit takes on the value of “1”.

5.2.2.2. Writing of a given Value.

Табл. 5.2.1 describes the process of carry propagation when a value is written in AGC with the basic code δ. The carries are determined by the following formulae:

$$\mu = \pi \wedge \overline{(\delta \oplus \gamma)}, \qquad \eta = \pi \wedge (\delta \oplus \gamma).$$

Table 5.2.1. Writing a value with a given code.

π	δ	γ	μ	η
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Where μ=1, the β digit takes on the value of “1” regardless of its former value. Similarly, where η=1, the α digit takes on the value of “1”. In such way, or a new path is formed, and the given value is written in it, or a path is discovered, in which the given value is already written. In this case a search for the address of the given value is performed.

5.2.2.3. Reading the value of the path with a given number.

Let an open path (β≡1 and α≡1) have a number with the linear code δ. The process of carry propagation while reading the value of this path is described in Table 5.2.2. In it, ω - is the corresponding digit of the linear code of this path's value. The signal ω is created in the digit α or β, through which the carry signal has passed. Thus,

$$\mu = \pi \wedge \bar{\delta}, \qquad \eta = \pi \wedge \delta,. \qquad \omega = (\mu \wedge \gamma) \vee (\eta \wedge \bar{\gamma}).$$

Table 5.2.2. Reading the value of the path with a given number

π	δ	γ	μ	η	ω
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	1	0	1	0

5.2.2.4. Addition of AGC and the basic code when ρ=2

This operation is described in Table 5.2.3. Carry signals μ, η and the transposition signal τ are created as functions of π, δ, γ. After that, the signal τ=1 changes the value of γ into its opposite, and the signals μ and η propagate further (if β=1 and α=1 respectively).

Table 5.2.3. Addition of AGC and the basic code when ρ=2

π	γ	δ	μ	η	τ
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	1	1	0

Example 5.2.1 of addition when ρ=2. Same as in example 5.1.1, let the basic code be ***K***=<2> or ***K***=10, and the attribute geometrical code ***G*** depict a set of linear codes {1100, 0010, 1010, 0110} or, which is the same thing, a set of numbers {12, 2, 10, 6}.

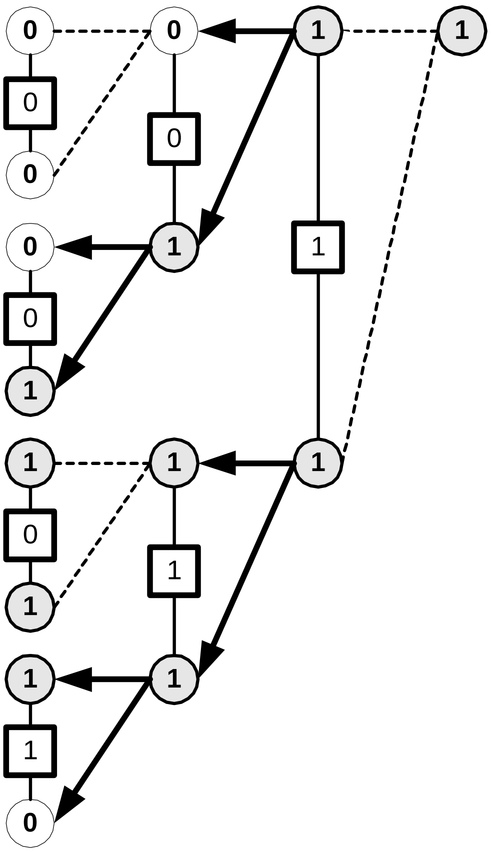


Fig. 5.2.2. For example 5.2.1.

The resulting attribute geometrical code is $R=G+K$. Fig. 5.2.2 shows code R. The thick arrows show those connections along which the carry $\pi=1$ propagated during addition. Square windows show the γ digits. They refer to that pair of digits α and β, which have been placed in round windows conjugated with the given square window. If all the γ digits were to be zeroized, the same figure would depict the primary code G – compare with the geometrical code G_1 in example 5.1.1. Thus the result differs from the primary code only by the values of the γ digits. Note that if the transposition is performed in accordance with the γ digit values, geometrical code G_4 of the result is formed as shown in example 5.1.1.

5.2.2.5. Inverse addition of AGC to the basic code when ρ=-2

This operation is described in Table 5.2.3a. The signals of carries μ, η and the signal or transposition τ are produced as functions of π, δ, γ. After that the signal τ=1 changes the value of γ into its opposite, and the signals μ and η propagate further (if β=1 and α=1, respectively).

Table 5.2.3a. Inverse addition of AGC and basic code when ρ=-2

π	γ	δ	μ	η	τ
0	0	0	0	1	0
0	0	1	1	1	1
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	0	0	1
1	1	1	1	0	0

Example 5.2.2 of inverse addition when ρ=-2. As in example 5.1.2, let the basic code be *K*=<2> or *K*=110, and the attribute geometrical code *G* depict a set of linear codes {0000, 0100, 0010, 0110} or, which is the same thing, a set of numbers {0, 4, -2, 2}. The resulting attribute geometrical code is *R=-G-K.* Fig. 5.2.3 shows code *R.* The thick arrows show those connections along which the carry π=1 proliferated during addition. Square windows show the γ digits. They refer to that pair of digits α and β, which have been placed in round windows conjugated with the given square window. If all the γ digits were to be zeroized, the same figure would depict the primary code *G* – compare with the geometrical code G_1 in example 5.1.2. Thus the result differs from the primary code only by the values of the γ digits. Note that if the transposition is performed in accordance with the γ digit values, geometrical code G_4 of the result is formed as shown in example 5.1.2.

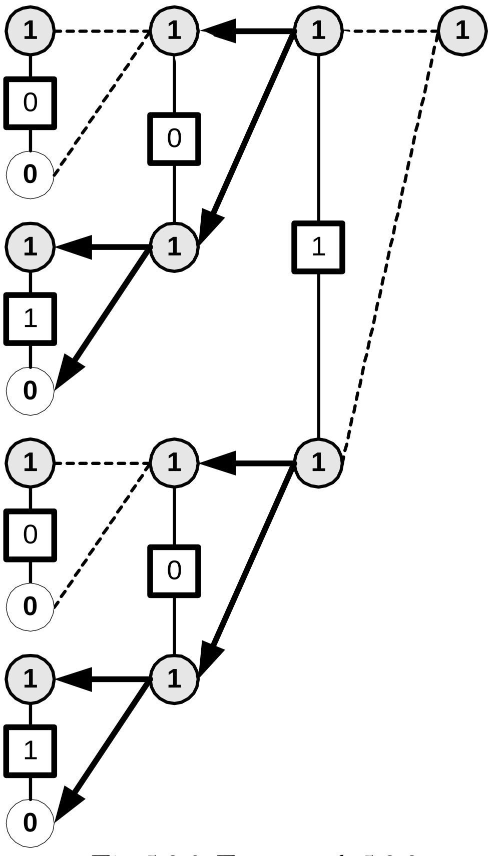


Fig. 5.2.3. For example 5.2.2.

Let us consider the circuit of forming the carries signals μ, η and the transposition signal τ in the involved adder – see Fig. 5.2.3a. On this scheme

π - the input carry signal,
μ, η - the output carry signals,
β - trigger of the digit β ,
α - trigger of the signal α ,
γ - trigger of the signal γ ,
δ - trigger of the signal δ of the basic code,
τ - trigger od the transposition signal τ ,
Sum – single-digit inverse addition circuit,
And – transposition signal key τ,
R – transposition enabling signal.

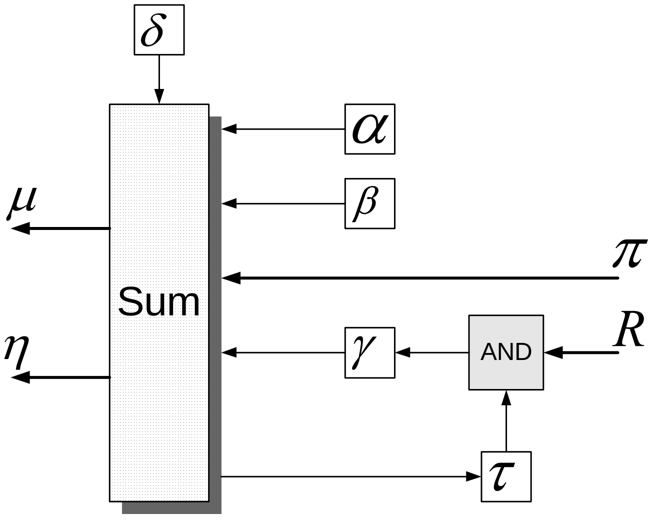


Fig. 5.2.3a. One-digit inverse addition circuit

This circuit covers the first threesome of geometrical code digits and produces carry signals to the two next threesomes of geometrical code digits. The transposition enabling code is common for all digits and comes from the control circuit afer the end of carries propagation though all the digits.

5.2.2.6. Inversion of AGC when ρ=-2.

Table 5.2.4 describes the process of carry propagation during the inversion of geometrical code with a radix "-2". In it

γ - is the value of the γ digit in the primary code;

τ - is the signal of transposition of the primary code to the resulting code (this code is generated in the primary code register).

Table 5.2.4. AGC inversion when ρ=-2.

π	γ	μ	η	τ
0	0	0	0	0
0	1	0	0	0
1	0	0	0	1
1	1	0	0	1

5.2.2.7. Algebraic addition of AGC splits into the operations of addend inversion and inverse addition.

5.2.2.8. Search for the Next Open Path, its Number and it's Value

Here it is assumed that a known path is determined by its number. The search consists of three consequently performed operations: 1) search for a path with a given number and fixation of its terminal vertex; 2) search for the next terminal vertex; 3) reading the number and path value with the given terminal vertex.

5.2.2.9. Multiplication of AGC by the basic code.

This multiplication is performed similarly to multiplication of the primary GC by the basic code – see section 5.1.2.7. The difference is that the analyzed digits are $\alpha=1$, if $\tau=0$, or $\beta=1$, if $\tau=1$ (and not the digits $\alpha=1$, as in the primary geometrical code). This multiplication is performed according to the following algorithm:

1. the primary geometrical code is shifted to the left one digit to the left;
2. the lower digit B_j of the basic code is analyzed, where $j=1$;
3. if $B_j=1$, then an inverse addition of the shifted code with the primary code is performed; a negative geometrical code of the partial product is generated;
4. the geometrical code of the partial product is shifted to the left 1 digit to the left;
5. the next digit B_j of the basic code is analysed;
6. if $B_j=1$, and the partial product was positive, then the inverse addition of the shifted code with the primary code is performed; a negative geometrical code of the partial product is generated;
7. if $B_j= 1$, and the partial product was negative, then the inverse addition of the shifted code with the negative primary code; a positive geometrical code of the partial product is generated;
8. if all the digits are exhausted, the multiplication is completed;
9. then go to point 3.

Evidently, this algorithm is in many ways similar to an algorithm of ordinary addition. The operations constituting the algorithm have already been described above. It is important to note that in a geometrical code of a product **the attribute's number is increased 2^n times** relative to the number of initial geometrical code's attribute (n is digit capacity of the base code, number of shifts).

5.2.3. Attribute Geometrical Codes in a Complex Radix

As indicated above (see section 3.1), complex numbers can be used as a basis for coding linear codes. In a similar fashion, attribute geometrical codes can be constructed using a complex radix - **AGCC**. Unlike the preceding, the value of the path in such codes is the linear binary code with a complex radix. But the most important difference lies in the algorithms of the arithmetic operations. Let us review algorithms of arithmetic operations with geometrical codes in systems of 1, 2 and 3 coding. In these systems, the digits representing the real and imaginary parts of a complex number alternate. Algebraic addition of each part is performed independently according to the rules of algebraic addition of codes of real numbers with a radix «-2». Therefore, in the further presentation of codes we can make use of the results from the preceding section. Some operations do not depend on the coding radix at all, and they will not be reviewed here.

5.2.3.1. Inverse addition of AGCC with the basic code

This operation is described in Table 5.2.3. Carry signals μ, η and the transposition signal τ are generated as functions of π, δ, γ. Thereafter the signal τ=1 changes the value of γ into its opposite, and the signals μ and η propagate further (if β=1 and α=1, respectively). These signals are transmitted to every other tier. A pattern of their propagation is shown in Fig. 5.2.4.

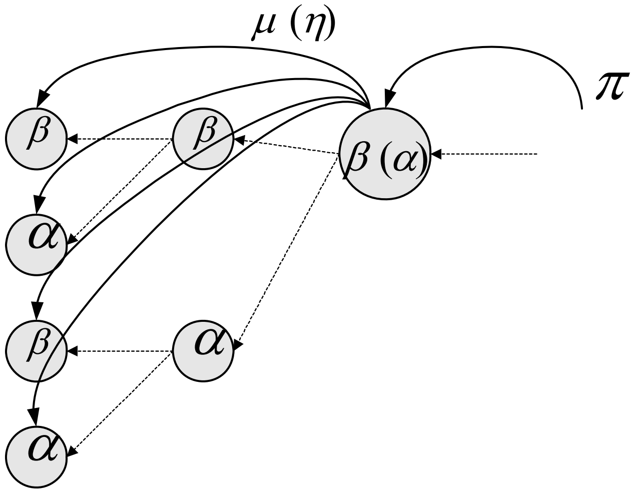


Рис. 5.2.4. Схема распространения переносов в комплексном GC

5.2.3.2. Inversion of AGCC.

This operation is performed similarly to the way the operation for codes with a radix «-2» is described in the preceding section.

5.2.3.3. Deformation of AGCC.

This operation is equivalent to the deformation of a figure, and is performed according to the following formula:

$$S = \operatorname{Re} G \cdot Z' + \operatorname{Im} G \cdot Z'',$$

where

G – is the primary geometrical code,

S – is the resulting geometrical code,

Z', Z'' - two linear codes (complex numbers).

This multiplication is performed similarly to the multiplication of prime GC by the basic code – see Section 5.1.2.7. The difference is that the analyzed digits are α=1, if τ=0, or β=1, if τ=1 (and not the digits α=1, as in prime GC). The difference is that during inverse addition and inversion, the carries are transmitted to every other tier.

Deformation is possible only if the complex codes Z', Z'' have a junior digit of 1. In order to have this case, these codes must be transformed according to the formula $Z \Rightarrow -\rho \cdot Z + 1$. After that the resulting code must be shifted one digit to the left.

Let us indicate some particular cases:

- if $Z = Z' = Z''$ we have ordinary multiplication: $S = G \cdot Z$,
- if $Z = Z' = Z'' = j$ we have a 90 degree turn: $S = G \cdot j$,
- if $Z = Z' = Z'' = -j$ we have a (-90) degree turn: $S = -G \cdot j$.

The last two operations are greatly simplified in the coding system 1, when the code of the number j is of the form “10”.

Example 5.2.3 of deformation with $\rho = j\sqrt{2}$. We shall consider the example of AGC deformation by analogy with the example 5.1.7 of GC deformation. Let us regard a figure determined by 6 points a_i, see Fig. 5.1.16, and Table 5.1.3. We shall deform this figure in such a way that the points $a_i = (x_i + jy_i)$ will turn into points b_i, where $b_i = (x_i + jy_i(1 + j\sqrt{2}))$. This deformation is equivalent to a shift of the figure horisontally by an angle $\Psi = 55^0$ $(tg\Psi = \sqrt{2})$. All the codes of numbers a_i are presented by a single code AGC of the whole figure. This code is shown of the Fig. 5.2.5 and it combines all the points of the initial figure. Al digits in this code are τ=0. Decoding this code we may make certain that the linear codes of all open paths have value a_i indicated in the Table 5.1.3. Code of the point a_i for every open path is indicated in the Fig. 5.2.5 opposite the corresponding terminal vertex.

This deformation of the figure is equivalent to the AGC imaginery part‘s multiplication by basic code of *K*=0011 of the number $(1 + j\sqrt{2})$. The result is the formation of AGC code of the deformed figure. This code is shown in the Fig. 5.2.6, and it combines the points of the deformed figure. The code changes for each open path's point, but the path's position does not change – compare Fig.5.2.5 and Fig. 5.2.6. In the deformed figure's AGC not all the digits are τ=0. Decoding this code we may see that linear codes of all open paths have value b_i, that is, it combines all codes of the deformed figure's points b_i. The point's codes b_i for every open path are indicated in the Fig. 5.2.6 opposite the corresponding terminal vertex.

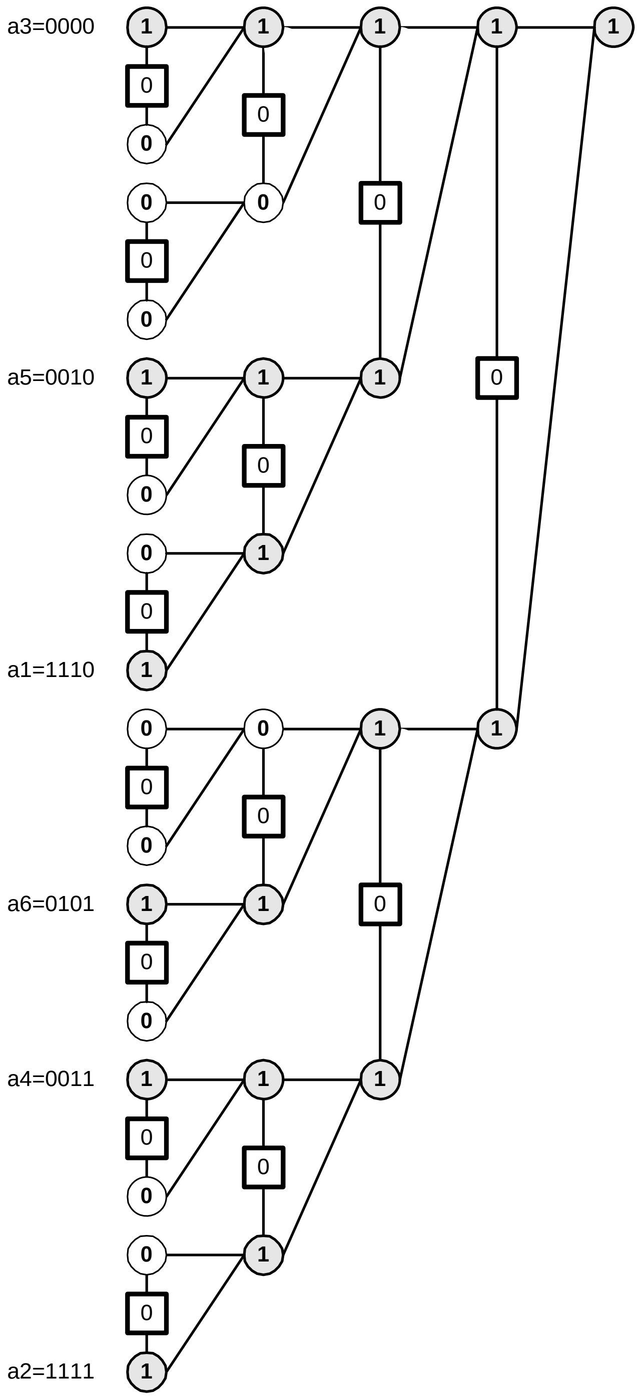


Fig. 5.2.5. For example 5.2.3: AGC of the initial figure.

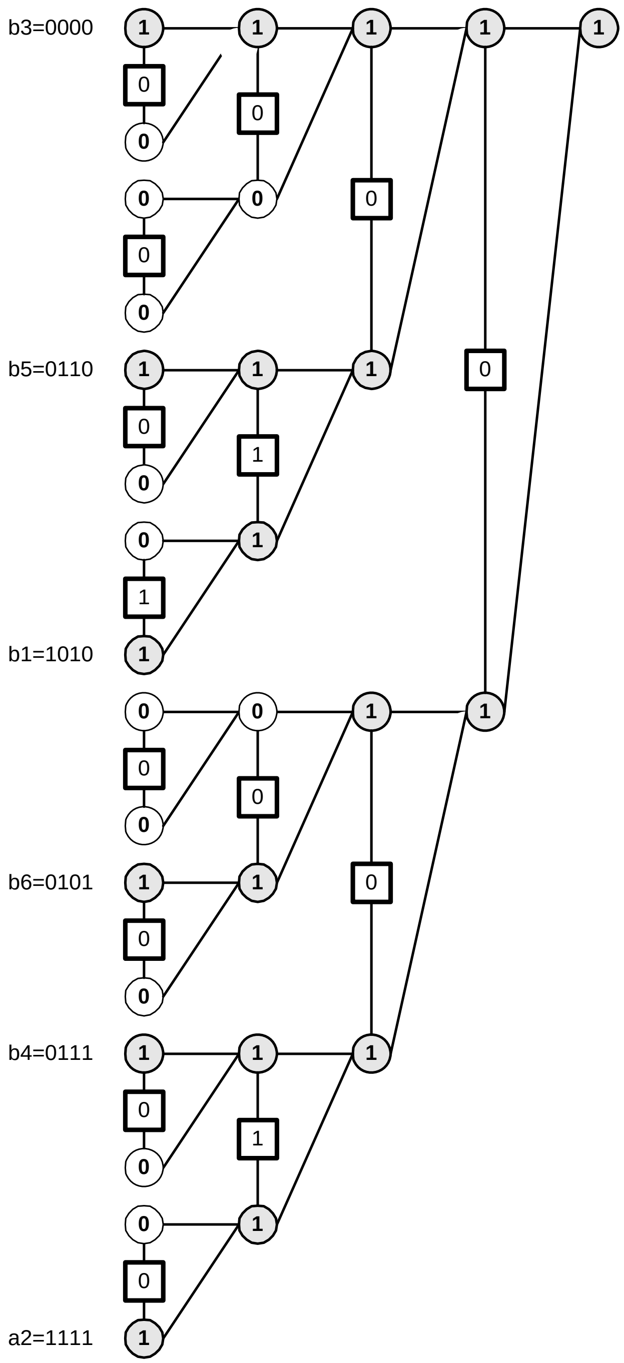


Fig. 5.2.6. For example 5.2.3: AGC of the deformed figure

5.2.4. Attribute Geometrical Codes of Spatial Figures

Attribute geometrical codes with a complex radix reviewed above represent planar figures. These codes may be used for affine transformations of planar figures. In the general case, it is also necessary to perform projection transformations of planar figures, as well as affine and projection transformations of 3-dimensional figures. It is a known fact that uniform coordinates are used for projection transformations. In this case a point on a plane is represented by three coordinates, while a point in 3-dimensional space – by four coordinates. If such representation is being used, a projective transformation of a planar figure includes an affine transformation of a three-dimensional figure, and projective transformation of a three-dimensional figure includes a n affine transformation of a four-dimensional figure.

Thus, attribute geometrical codes of planar, 3-dimensional and 4-dimensional figures that lend themselves to affine transformations may be used to solve all of the geometrical transformation problems. The method applied for such codes' synthesis is that of positional coding – see part 3. This method, similarly to the complex nymbers coding method, lets us represent a spatial vector by a single binary code. Moreover, this method allows to perform algebraic addition and multiplication of such codes.

Linear binary codes of vectors may be unified in GC. In this case, GC of a 3- or 4-dimensional figure is formed. Arithmetic operations with such GC are fully analogous to operations with the GC of a planar figure. Addition circuits differ only in that the carries propagate through every 2nd or 3rd tier (for a 3- or 4-dimensional figure, respectively).

Deformation of geometrical code in the general case is performed according to the formula

$$S = \mathrm{part1}(G)\cdot Z_1 + \mathrm{part2}(G)\cdot Z_2 + \mathrm{part3}(G)\cdot Z_3 + \mathrm{part4}(G)\cdot Z_4,$$

where

G – is the initial geometrical code,

S – is the resulting geometrical code,

part$p$ – a part of code G,

Z_p – linear codes (vectors),

$p = \{1, 2, 3, 4\}$,

i, j, k, m – orts of vectorial space,

h >= 0- integer,

r = tier number.

As usual, deformation involves substituting the digits $\alpha_r = 1$ by linear codes Z_p. When the vector coding method 2 is used, the Z substitution code is selected as follows:

for 3-dimensional vectors

$$Z = Z_1, \quad \text{if} \quad r = 3\,h+1,$$
$$Z = Z_2, \quad \text{if} \quad r = 3\,h+2,$$
$$Z = Z_3, \quad \text{if} \quad r = 3\,h+3;$$

for 4-dimensional vectors

$$Z = Z_1, \quad \text{if} \quad r = 4\,h+1,$$
$$Z = Z_2, \quad \text{if} \quad r = 4\,h+2,$$
$$Z = Z_3, \quad \text{if} \quad r = 4\,h+3,$$
$$Z = Z_4, \quad \text{if} \quad r = 4\,h+4.$$

When the vector coding method 1 is used, the Z substitution code is selected as follows:

for 3-dimensional vectors

$$Z = Z_1, \quad \text{if} \quad r = 3\,h+1,$$
$$Z = j \cdot Z_2, \quad \text{if} \quad r = 3\,h+2,$$
$$Z = k \cdot Z_3, \quad \text{if} \quad r = 3\,h+3;$$

for 4-dimensional vectors

$$Z = Z_1, \quad \text{if} \quad r = 4\,h+1,$$
$$Z = j \cdot Z_2, \quad \text{if} \quad r = 4\,h+2,$$
$$Z = k \cdot Z_3, \quad \text{if} \quad r = 4\,h+3,$$
$$Z = m \cdot Z_3, \quad \text{if} \quad r = 4\,h+4.$$

5.2.5. Contracted Attribute Geometrical Codes

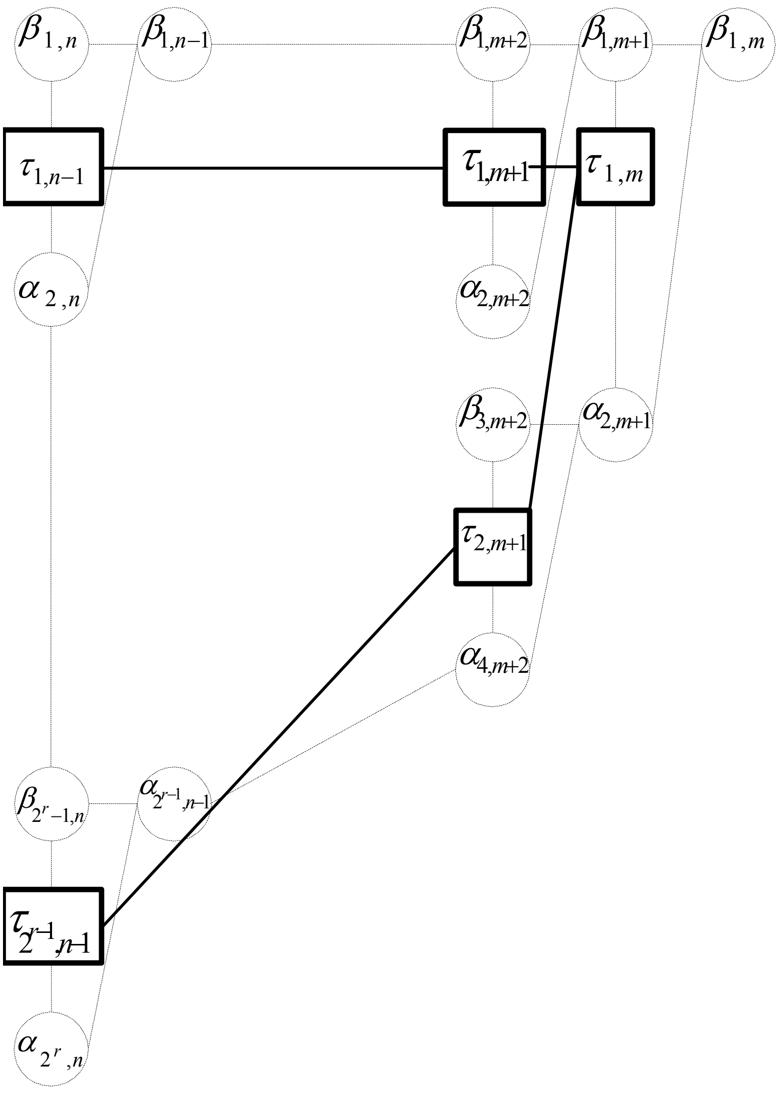


Fig. 5.2.7. Contracted Attribute Geometrical Code

In the previous section we have discussed AGC whose paths could be both open and close. The values of digits $\alpha=1$ (or 0) and $\beta=1$ (or 0) determine only the fact that the path in the geometrical code's tree is open (or close) and the corresponding linear code is included (or not included) into the coded set of linear codes. We shall now deal with the case when all linear codes of a given dimension are included into the coded set. In this case all α=1 and all β=1. So there is no need to actually include these digits to the geometrical code. In all manipulations with AGC we may assume that all α=1 and all β=1. AGC without digits will be called <u>contracted AGC</u>- **CAGC.**

Fig. 5.2.7 present a contracted AGC. The actually absent digits are denoted by dotted line. The number of digits in a contracted AGC is defined by the following formula:

$$V = \sum_{k=0}^{n-m} 2^k = 2^r - 1.$$

Each terminal vertex $\tau_{k,n-1}$ of the contracted AGC's tree has two corresponding attributes – the upper one, corresponding to the imaginary vertex $\beta_{k,n}$, and the lower one, corresponding to the imaginary vertex $\alpha_{k+1,n}$.

Evidently, the operation circuits for a contracted AGC are much shorter, and the code's volume is three times reduced. More accurately, if in an ordinary AGC the number of digits is $V = 3 \cdot 2^r - 2$, in a contracted GC it is equal to $V = 2^r - 1$.

6. Geometrical Processor

6.0. Data Presentation

We shall assume, as before, that a set (M) of points in a p-dimensional space is given. The points constitute the domain of definition, which is a p-dimensional cube, and they are distributed in this domain, located in the nodes of a uniform network. The node's coordinates are represented by a pn-digit code of a vector with fixed point. All the domain is defined by the set of these codes, with the overall number of $M=2^{pn}$. Each node is defined by a triad: *address-coordinate vector- attribute.*

Let us consider now the data representation in the random-access memory and in the arithmetic unit.

There are two possible methods of the random-access memory organization. The first, simple one involves building two interrelated arrays. The first of them contains the coordinate vectors, and the second – the attributes. For this method the random-access memory (RAM) may be realized as a dynamic one (DRAM), or as a static one (SRAM). The memory for realizing this method will be called traditional random-access memory **TRAM**.

The second method, using GC, assumes that all the codes of coordinate vectors are integrated in AGC. Each path in AGC corresponds to a value interpreted as "*coordinate vector*", and to a number, interpreted as "*address*". The attributes in this method are, as before, integrated into an array connected with AGC by means of the addresses. For this method the random-access memory should be realized as static one (SRAM), because certain logic of access to the AGC memory should be ensured. Further we shall call the random-access memory for realizing this method – specific random-access memory **SSRAM.**

In future we shall consider only SSRAM, since this memory (unlike traditional memory) makes possible to obtain the given code's address by one access. This access method is necessary for vector's attribute retrieval, and also increases essentially the processor speed. Besides, SSPAM is much more efficient than traditional random-access memory.

More precisely, the coordinate array in TRAM contains $2^{pn}pn$ digits, and the reduced AGC has only 2^{pn} digits. For example, if p=3 and n=12, then the memory volume is 36 times smaller.
Two schemes of memory organization for AGC may be applied:

- *full,* when the whole code AGC with carry propagation schemes is kept in **PSSRAM**
- *fragmentary*, when all the code is split into fragments, and there is **FSSRAM** with carry propagation schemes for one fragment.

Such AGC structure will be called *vertical fragmentation.*
Let us consider now data presentation in the arithmetic unit. The first method consists in creating a complete AGC of 2^{pn} digit capacity in the arithmetic unit. This is, however, insufficient, as when operating with AGC, carries may occur in any path from high-order digit (by analogy with vector processor). The resulting code may have a digit capacity of *pr*, whereas the initial code had digit capacity pn. The higher digits of the resulting code may be combined into a rectangular code of vectors RCV (similar to the way it was done in a vector processor for full-sized code). Thus, there must be register AGC and register RCV in the arithmetic unit. Let us call this pair of codes – a mixed code of a figure **MCF**. The register of mixed code has a digit capacity of $pr+2^{pn}$.

Arithmetic unit with MCF register will serve simultaneously as random-access memory. Let us call it a *maximal arithmetic unit for geometrical figures* - **MGAU**. Evidently, this unit has a very large volume, and the possibility of its realization is at the limit of modern technology potential. Therefore we shall view another variant. Let us divide MCF into several fragments MCF$_q$, each of which consists of fragment AGC$_q$ and fragment RCV$_q$. MCF$_q$ combines Q *paths in the code* MCF, i.e. Q resulting codes of digit capacity $(n+r)$. If Q=*2*, then the lower tiers of the AGC$_q$ code fragment will concentrate in one path, and MCF$_q$ will have a structure as shown in the Fig. 6.0.1.

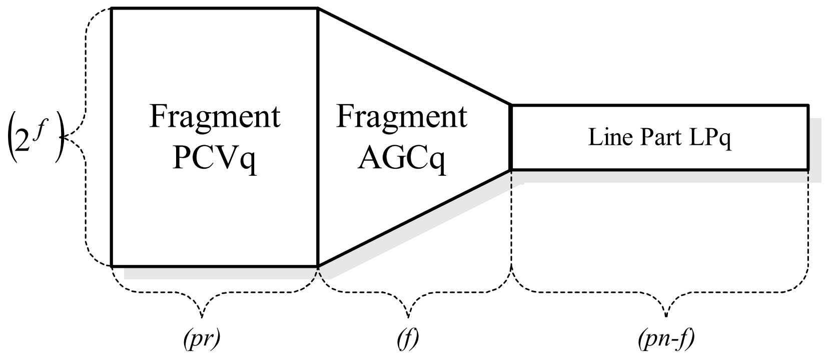


Fig. 6.0.1. MCF structure

Such MCF structure will be called *horizontal fragmentation*. In this case

- the linear part LP_q of the code MCF_q contains $(pn\text{-}f)$ digits,
- the rectangular part RCV_q of the code MCF_q contains (Qpr) digits,
- the code MCF_q contains $(pn\text{-}f)+Qpr+Q)$ digits,
- the geometrical part AGC_q of the code MCF_q contains Q digits,
- the code MCF consists of 2^{pn-f} fragments MCF_q and contains $((pn\text{-}f)+Qpr+Q)*2^{pn-f}$ digits.

Accordingly, the affine transformation for such data structure consists of 2^{pn-f} operations. An arithmetic unit with register of such structure will be called fragmentary GAU – **FGAU**.

Notice that horizontal fragmentation is convenient for arithmetic unit, and vertical fragmentation – for random-access memory.

6.1. Full Specific Random-access Memory

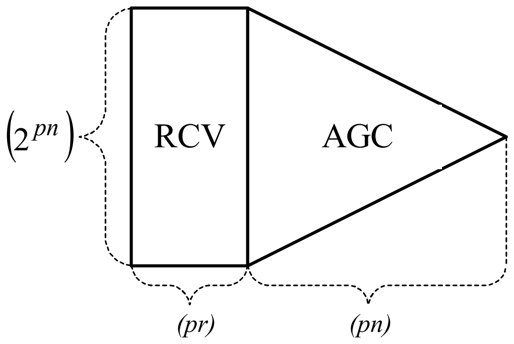


Fig. 6.1.1. Full specific random-access memory

Full specific random-access memory **PSSRAM** of the code MCF is shown in the Fig. 6.1.1 and it consists of two parts – memory for rectangular code **RCV** and memory for AGC. It is complemented by a certain scheme of carry propagation, and this makes it possible to perform the following operations

S1. Writing the vector code – see section 5.2.2.1.

S2. Determining the address where this vector code is located – see section 5.2.2.2. This operation is necessary for searching the vector's attribute.

S3. Reading the vector code according to the address where it is located – see section 5.2.2.3.

S4. Writing a code fragment MCF_q according to its number q.

S5. Reading a code fragment MCF_q according to its number q.

S6. Determining the number of the first non-zero digit in the code MCF, which is necessary for rounding off.

6.2. Fragmentary Specific Random-access Memory

In this case for representation in the memory, the geometrical code *G* is split into fragments joined into *F* tiers. The fragment of each tier contains *r* tiers of geometrical code. This statement is illustrated on Fig. 6.2.1, where fragments of the geometrical code are shown as triangles. Their numeration has the following meaning: *"number of tier"."number of fragment in the tier"*.

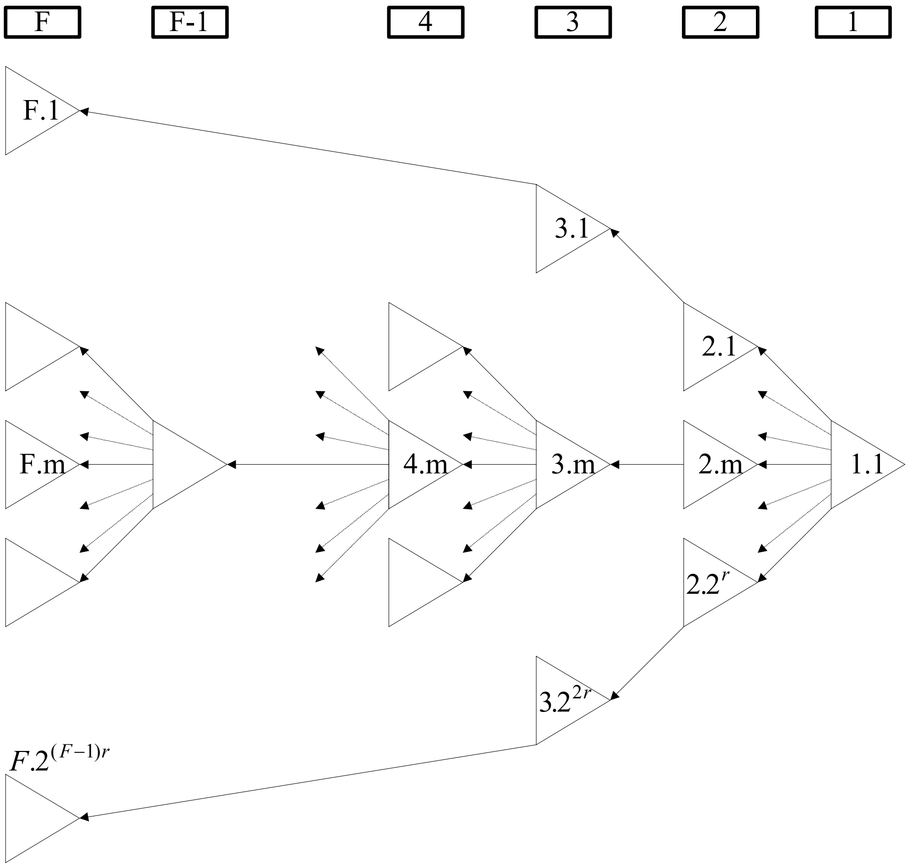


Fig. 6.2.1. Tiers of geometrical code and segments of linear code.

In the memory, the fragments are placed in a series, tier after tier:

$$1.1,$$
$$2.1,\ 2.2,\ \ldots,\ 2.2^{r},$$
$$3.1,\ 3.2,\ \ldots,\ 3.2^{2r},$$
$$\ldots$$
$$k.1,\ k.2,\ \ldots,\ k.m,\ \ldots,\ k.2^{(k-1)r},$$
$$\ldots$$
$$F.1,\ F.2,\ \ldots,\ F.2^{(F-1)r}$$

Note that the number j of the fragment in this series is connected with the number k of the tier and the number m of the fragment in the tier by the following dependence:

$$j = m + \sum_{a=1}^{k-1} 2^{(a-1)r} = m + \frac{1-2^{(k-1)r}}{1-2^{r}}. \qquad (6.2.1)$$

Given:

- number k of the tiers of fragment, where the processed fragment is placed,
- number m of the processed fragment in the tier k,
- number v of a vertex in the last tier of the processed fragment the carry arrives at,

the number f of the fragment is the next *(k+1)*-tier, where this carry arrives, may be determined according to the formula:

$$f = m + v - 1. \qquad (6.2.2)$$

Number of the fragment the carry arrived at, is

$$j = f + \frac{1-2^{kr}}{1-2^{r}}. \qquad (6.2.3)$$

Fragments numbered in accordance with (6.2.1), are saved in the ordinary random-access memory of fragments, retrieved into the unit FSSRAM for processing, and after processing returned to the fragments memory. A fragment is written or read from the fragment memory according to the fragment's number in this memory. Unit FSSRAM realizes the above named commands S1-S6; in order to do so it accesses some of the fragments. The following operations in the unit FSSRAM are performed directly with fragments:

F1. Reading a fragment with a given number of the fragments memory.

F2. Writing a given vector code SEGMENT according to its number of the same code – see section 5.2.2.1. This operation is applicable only if all the digits $\gamma = 0$, i. e. with initial formation of AGC.

F3. Determining the address where this vector code SEGMENT is located – see section 5.2.2.2.

F4. Reading a vector code SEGMENT at the address where it is located – see section 5.2.2.3.

F5. Writing a fragment with a given number into the fragments memory.

F6. Determining the number of the highest nonzero digit in a fragment (used only in highest tier fragments).

F7. Determining by the formula (6.2.3) the number j of the next fragment the carry arrives at.

Let us now compare the volume and performance speed for different methods of random-access memory organization. When evaluating the performance speed we shall assume that all the paths in the GC tree are open, i. e. it integrates all the codes of the same digit capacity. Let us denote:

r – number of tiers in a fragment,

F – number of fragments' tiers in GC.

Then we have:

$n = r \cdot F$ - digit capacity of linear codes and number of GC tiers,

2^r - number of terminal vertexes in a fragment,

$\left(2^{2r} - 1\right)$ - total number of vertexes in a fragment,

$2^{(F-1)r}$ - number of terminal fragments in GC.

As was shown above, number of digits in reduced AGC is

$$V = 2^n - 1.$$

This code joins all n – digit codes. Total number of bits for storing these codes is

$$V' = n \cdot 2^n.$$

<u>Hence using AGC reduces the data volume n times</u>.

Let us consider now the number of elementary operations in writing the fragmentary AGC. In the lowest fragments tier 1 operation with fragment is being performed, in the second tier - 2^r operations, the third one - 2^{2r} operations, …, in the highest tier - $2^{(F-1)r}$ operations. So the total number of elementary operations with a fragment is

$$a_1 = 1 + 2^r + 2^{2r} + \ldots + 2^{(F-1)r} = \frac{1 - 2^{Fr}}{1 - 2^r}$$

or

$$a_1 \approx 2^{(F-1)r} = 2^{n-r} \qquad (6.2.4)$$

The ratio of memory capacity for all fragment in SSRAM to one SSRAM fragment's capacity is equal to

$$R \approx 2^{n-r} \qquad (6.2.5)$$

The volume of FSSRAM unit is approximately 3 times larger than one fragment's volume, because the unit contains carries schemes in every digit. Hence the random-access memory complexity is characterized by the value

$$\theta \approx 2^n + 3 \cdot 2^r = \left(3 + 2^F\right) \cdot 2^r \qquad (6.2.6)$$

6.3. Maximal Arithmetic Unit of Geometrical Figures

Maximal arithmetic unit of geometrical figures **MGAU,** similarly to full specific random-access memory PSSRAM, operates with code MCF, presented in Fig. 6.1.1. This unit contains a well-developed scheme of carries propagation, and due to this it can perform the following operations:

M1. Writing the given vector code – see section 5.2.2.1.

M2. Reserve.

M3. Reading the vector code according to its address – see section 5.2.2.3.

M4. Reserve.

M5. Reserve.

M6. Determining the highest nonzero digit in the code MCF, which is necessary for rounding off.

M7. Adding MCF to given vector code – see section 5.2.2.7.

M8. Multiplying MCF by transformation matrix – see section 5.2.2.9.

M9. Determining the length of vector code by its address.

M10. Reading from MGAU a vector code according to the first\next address – see section 5.2.2.8.

6.4. Fragmentary Arithmetic Unit of Geometrical Figures

The arithmetic unit **FGAU** for operations with code MCF_q is shown in the Fig. 6.4.1

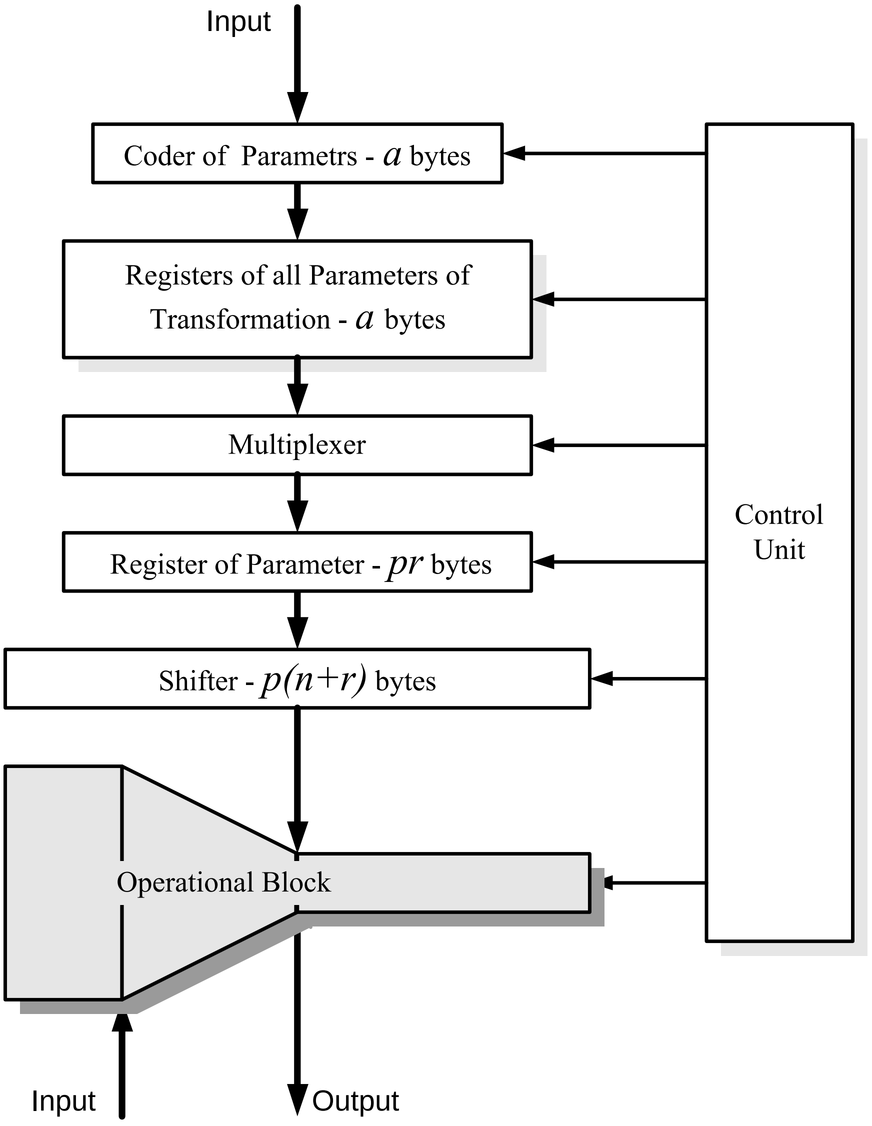

Input
Coder of Parametrs - a bytes
Registers of all Parameters of Transformation - a bytes
Multiplexer
Register of Parameter - pr bytes
Shifter - p(n+r) bytes
Operational Block
Control Unit
Input
Output

Fig. 6.4.1. Fragmentary arithmetic unit

It is in many ways similar to the unit FVAU. The difference is in the fact that FVAU includes operational unit of digit capacity:

$R_6 = Qp(n+r),$ (6.4.1)

and FGAU includes operational unit of digit capacity

$R_7 = (pn-f)+Qpr+2^{pn-f}.$ (6.4.2)

Operational unit consists of register MCF_q and carry propagation schemes. The carry schemes in the linear and the rectangular parts are organized according to the rules of vector arithmetic, and in geometrical part – according to the rules of geometrical codes arithmetic.

The ratio between digit capacity of the units FVAU and FGAU is:

$R_6 / R_7 \approx (n+r)/r$ (6.4.3)

Let us consider the list of co-processor commands realized in FGAU:

A1. Receiving the transformation parameters.

A2. Adding MCF_q to carry vector.

A3. Multiplying MCF_q by carry matrix.

A4. Yielding the code MCF_q.

A5. Receiving the code MCF_q.

A6. Yielding the vector by address – without rounding off and with rounding off.

A7. Determining the vector code length according to the address.

A8. Determining a vector adjacent to the given vector, according to a given coordinate and given direction on the coordinate axis.

6.5. Processor with a Maximal Arithmetical unit

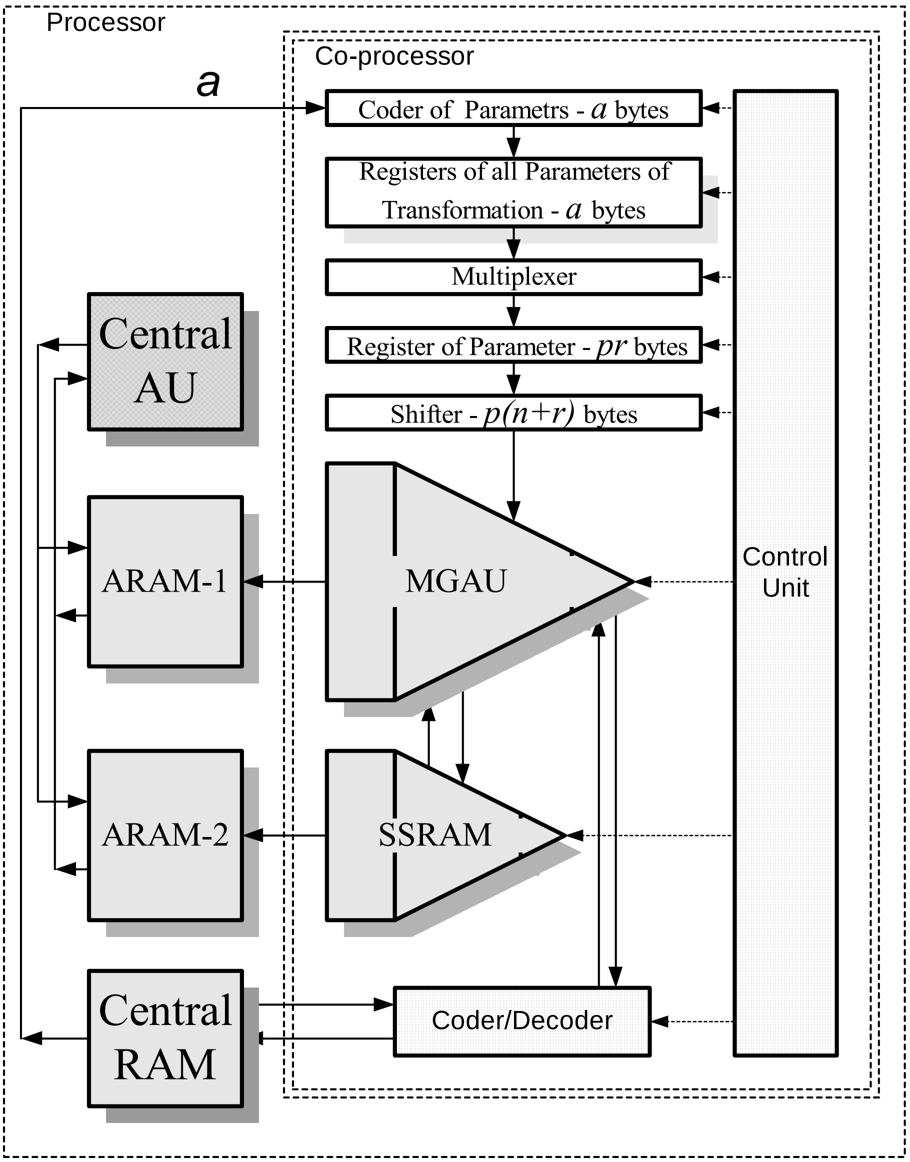


Fig. 6.5.1. Processor with a maximal arithmetical unit

Processor with a maximal arithmetic unit **PMGAU** contains an arithmetic unit MGAU for operation with codes MCF that combines the functions of arithmetic unit and of random-access memory. The co-processor PMGAU and its place in the central processor are shown in the Fig.6.5.1.

Thus, PMGAU contains the arithmetic unit MGAU and a unit of additional specific random-access memory SSRAM, coder/decoder of vectors and control unit, as well as other units, similarly to the unit FVAU. Coder/decoder is connected with the main memory of the central processor. MGAU and SSRAM are connected with attribute memory units ARAM-1 and ARAM-2, which are parts of the central processor.

It should be noted that the co-processor entirely releases the central processor from solving respective problems, so that the latter may solve other problems simultaneously with co-processor. Let us consider now the list of co-processor's commands and the units used in performing these commands:

R1. Receiving (by the bus *a*) and coding the carry parameters.
R2. Adding MCF to carry vector (see operation *M7*).
R3. Multiplying MCF by transformation matrix (see operation *M8*).
R4. Reserve.
R5. Reserve
R6. Determining the number of the highest non-zero digit in MGAU for rounding off (see operation *M6*).
R7. Transmitting the rounded k-vector from MGAU into SSRAM.
R8. Determining a vector code length according to the address (see operation *M9*).
R9. Reading a vector by its address in SSRAM (see operation *S3*).
R10. Determining a vector adjacent to a given vector, by a known coordinate and known direction.
R11. Determining an address of a known vector in SSRAM (see operation *S2*).
R12. Transforming coordinates into vector, writing it to MGAU and determining its address (see operation *M1*). The coding of a point's coordinates into vector code is performed by the coder.

R13. Reading from MGAU vector's code by a given address (see operation *M3*) and transforming this vector into a point's coordinates – performed by the decoder.

R14. Reading from MGAU a vector code by the first/next address (see operation *M10*).

6.6. Processor with Fragmentary Arithmetic Unit

Co-processor **PFGAU** with fragmentary FGAU and its place in the central processor is shown in the Fig. 6.6.1. Co-processor contains an arithmetic unit FGAU, specific random-access memory unit SSRAM-1 and an additional specific random-access memory unit SSRAM-2, coder/decoder of vectors and a control unit.

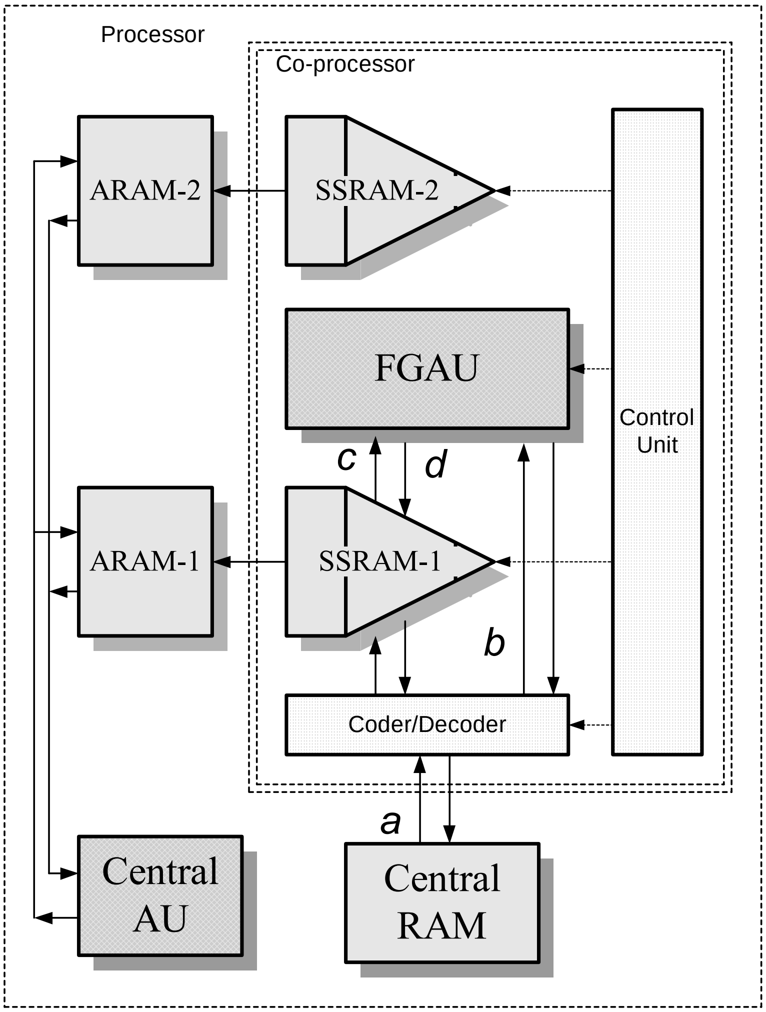


Fig. 6.6.1. Processor with fragmentary FGAU

Coder/decoder is connected with the main memory of the central processor. SSRAM-1 and SSRAM-2 are connected with attribute random-access memory units ARAM-1 and ARAM-2, which are parts of the central processor.

It should be noted that the co-processor entirely releases the central processor from solving respective problems, so that the main processor may solve other problems simultaneously with co-processor.

From further discussion it follows that in the SSRAM-2 memory only two operations: *S1* and *S3* should be provided.

Let us consider list of commands of co-processor and of units used in their performance. (operations of arithmetic unit FGAU are denoted by symbols A):

P1. Receiving and coding transformation parameters. These parameters are transmitted by buses *a* and *b*. Operation A1 is being used at that.

P2. Adding MCF_q to the carry vector GAU. Operation A2 is being used at that.

P3. Multiplying MCF_q by transformation matrix in GAU. Operation A3 is being used at that.

P4. Yielding code MCF_q from GAU to SSRAM-1 by bus *d*. Operations A4 and S4 are being used at that.

P5. Receiving code MCF_q from SSRAM-1 to GAU by bus *c*. Operations A5 and S5 are being used at that.

P6. Determining number of highest non-zero digit SSRAM-1 for rounding off. Operation S6 is being used at that.

P7. Transmitting a rounded *k*-vector from GAU to SSRAM-2. Operations A6 and S1 are being used at that.

P8. Determining the vector code length by its address. It is assumed that the corresponding fragment is in GAU and operation A7 is being performed.

P9. Reading a vector by its address. Operation S3 is being used in SSRAM-2.

P10. Determining a vector adjacent to a given vector, by a known coordinate and known direction. Operation A8 is being used at that.

P11. Determining an address by a known vector. Operation S2 is being used in SSRAM-2.

P12. Transforming coordinates into vector, writing it into SSRAM-1 and determining its address. The coding of the point's coordinates into the vector code is performed by the coder, and operation S1 is used for writing the vector and for determining its address.

P13. Reading a vector code from SSRAM-1 by the vector's address and transforming this vector into point's coordinates. Operation S3 is used for reading the vector code, and its decoding to point's coordinates is performed by the decoder.

P14. Reading a vector's code from SSRAM-1 by the first/next address. Operation S3 is used for reading the vector code.

Notice that for SSRAM-2 only the operations S1, S2, S3 may be used.

6.7. The Main Procedures

Here we shall use the following (described above) notations of the units and operations performed by the units.

Section	Unit	Type	The used units	Operations
6.1	PSSRAM	RAM	-	S1-S6
6.2	FSSRAM	RAM	-	S1-S6; F1-F7
6.3	MGAU	AU	-	M1-M10
6.4	FGAU	AU	-	A1-A8
6.5	PMGAU	processor	MGAU; PSSRAM	R1-R14
6.6	PFGAU	processor	FGAU; PSSRAM or FSSRAM	P1-P14

6.7.1. Affine Transformation

In the processor PFGAU:

1. Receiving and coding the transformation parameters - operation P1.
2. Enumerating all q-fragments (in the central processor).
   2.1. Receiving code MCF_q from SSRAM-1 - operation P5.
   2.2. Multiplying MCF_q by the transformation matrix – centroaffine transformation - operation P3.
   2.3. Adding MCF_q to the carry vector - operation P2.
   2.4. Yielding code MCF_q from GAU to SSRAM-1 - operation P4.

In the processor PMGAU:

1. Receiving and coding the transformation parameters - operation R1.
2. Multiplying MCF by the transformation matrix - operation R7.
3. Adding MCF to the carry vector - operation R2.

6.7.2. Rounding

We shall call so an operation of building an array of pairs "point's coordinates" – "point's attribute" with simultaneous compression of the figure in the direction of one or several coordinate axes. To do this the complex codes are read from GC without some of the lower-order digits. For example, for plane figures

- the absence of the lowest digit is equivalent to twofold compression along the abscissa axis,
- the absence of two lowest digits is equivalent to twofold compression along both axes,

- the absence of three lowest digits is equivalent to 4-fold compression along abscissa axis and two-fold compression along both axes.

During such compression one coordinate may correspond to one or several attributes. The attributes' joining (as was noted above) is determined entirely by their application meaning.
During the operation of rounding all vector codes are rounded off (their lower digits are discarded) and are written from SSRAM-1 to SSRAM-2. The algorithm is as follows

In the processor PFGAU:

1. Determining the highest non-zero digit in SSRAM-1 - operation P6.
2. Zeroing ARAM-2 (in the central processor)).
3. Enumerating all *q*-fragments (in the central processor).
   3.1. Receiving all MCF_q from SSRAM-1 into GAU - operation P5.
   3.2. Enumerating all local *k*-addresses in the code MCF_q (operation P14.
      3.2.1. Transmitting the rounded k-vector from GAU to SSRAM-2 - operation P7.
      3.2.2. Transmitting ((q-1)k)-attribute from ARAM-1 to ARAM-2 (in the central processor). It is significant that there is a possibility of attribute being *added* to already existing attributes of this point.

In the processor PMGAU:

1. Determining the number of the highest non-zero digit in SSRAM-1 - operation R7.
2. Zeroing ARAM-2 (in the central processor).
3. Transmitting the rounded k-vector from GAU to SSRAM-2 - operation R7.
4. Transmitting ((q-1)k)-attribute from ARAM-1 to ARAM-2 (in the central processor). It is significant that there is a possibility of attribute being *added* to already existing attributes of this point.

6.7.3. Rough rounding

During all arithmetic operations there may occur an overflow – i.e. there may appear tiers with a number exceeding the maximal. Such overflow is equivalent to the point going out of the limits of coding domain (for example, out of the screen bounds). At rough rounding all the out-of-the limits points are excluded from the figure's code. To do this would require only discarding the higher digits of the vector code and zeroing its attribute. The algorithm is as follows.

In the processor PFGAU:

```
1. Zeroing ARAM-2 (in the central processor).
2. Enumerating all $q$-fragments (in the central processor).
    2.1. Receiving code $MCF_q$ from SSRAM-1 to GAU - operation P5.
    2.2. Enumerating all local $k$-addresses in the code $MCF_q$ – operation P14.
        2.2.1. Analyzing the length of $k$-vector code in GAU - operation P8.
        2.2.2. If there is no overflow in this code, then $((q-1)k)$-attribute is transmitted from ARAM-1 to ARAM-2 (in the central processor). Otherwise it will be left equal to zero.
```

In the processor PMGAU:

```
1. Zeroing ARAM-2 (in the central processor).
2. Enumerating all $k$-addresses in code MCF - operation R14.
    2.1. Analyzing the length of $k$-vector code in GAU - operation R8.
    2.2. If there is no overflow in this code, then $((q-1)k)$-attribute is transmitted from ARAM-1 to ARAM-2 (in the central processor). Otherwise it will be left equal to zero.
```

6.7.4. Attributes Correction

After the figure's code rounding it may occur that in a certain node of the network several points are present. It means that by a certain address in the attributes memory there is a list of attributes present. Attribute of a node is defined as a function of all attributes of all points found in this node. This procedure is known and is performed in the central processor.

6.7.5. Attributes Calculation

After rough rounding of the figure's code it may occur that in a certain node of the network a certain point is absent – its attribute is equal to zero. Attribute of a node is defined as a function of all attributes of all points found in this node. This procedure is also known and is performed in the central processor. But in this case it is necessary to access the co-processor. The algorithm is as follows.

1. The memory ARAM-2 is scanned (in the central processor)
2. If at a certain address the attribute is equal to zero, then

- 2.1. Vector V_0 of the point C_0 is determined at the given address A_0 - see operation P9 (or R9).
- 2.2. Coordinates and directions by the coordinates are enumerated (in the central processor). For every variant k
  - 2.2.1. Vector V_k of the adjacent point C_k is determined - see operation P10 (or R10).
  - 2.2.2. The address A_k of the point C_k is determined according to the known vector V_k - see operation P11 (or R11).
  - 2.2.3. The attribute T_k of the point in ARAM-2 is found by the known address A_k (in the central processor).
- 2.3. The attribute T_0 of the point C_0 is determined and written into ARAM-2 as a known function of attributes T_k of points C_k (in the central processor).

6.7.6. Coding a Figure.

By this term we mean a transformation of the connected arrays "attributes"-"coordinates" into mixed code of the figure. The algorithm is as follows:

The connected arrays are enumerated. For every address of the pair "attributes"-"coordinates" the following actions are performed

1. The coordinates of the point are transformed into vector code and this vector is written to SSRAM-1. A new address is yielded from SSRAM-1 . Operation P12 (or R12) is used for this purpose.
2. The point's attribute is written to ARAM-1 (in the central processor).

6.7.7. Decoding a Figure.

By this term we mean a transformation of mixed code of the figure into connected arrays "attributes"-"coordinates". The algorithm is as follows. The addresses of ARAM-1 are enumerated. For each address the following actions are performed:

1. By given address the point's attribute is written to ARAM-1 into "attribute array" (in the central processor).
2. The vector code is read by the address from SSRAM-1, and then is transformed into point's coordinates. Operation P13 (or R13) is used for this purpose.
3. These coordinates are written into "coordinate" array (in the central processor).

6.8. Operational Units

Following is the description of operational units included in the arithmetic unit and specialized random-access memory. These units constitute the patterns of carries propagation in AGC. Algorithms of the corresponding operations were presented above. When outlining the patterns we shall use the following notations:

π - input carry signal,

β - β digit trigger,

α - α digit trigger,

μ, η - output carry signals arriving at the β and α digits, respectively,

γ - γ digit trigger,

δ - basic code δ digit trigger,

τ - transposition signal τ trigger.

A general pattern of carry propagation is shown in Fig. 5.1.2a and Fig. 5.2.4. Algorithms of the corresponding operations were described above.

6.8.1. Writing unit for the number with the given code

Fig. 6.8.1 shows a fragment of the circuit for writing a number with the given basic code into the AGC – see section 5.2.2.1. Shown in this Figure are units that calculate the μ, η signals according to specific formulae. One of these signals always equals “1” and is transmitted further in the form of signal π. The signal $\mu=1$ or $\eta=1$ establishes the appropriate trigger in “1”. The signal $\mu=0$ or $\eta=0$ does not change the condition of the corresponding trigger.

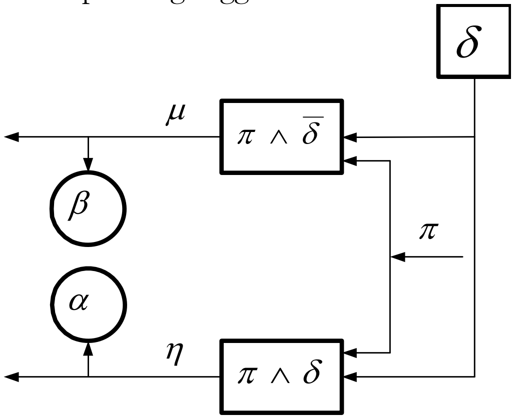


Fig. 6.8.1. Writing unit of number with given code.

6.8.2. Writing unit for the value with the given code

Fig. 6.8.2 shows a fragment of the circuit for writing a number with the given basic code into the AGC – see section 5.2.2.2. Shown in this Figure are units that calculate the μ, η signals according to specific formulae. One of these signals always equals “1” and is transmitted further in the form of signal π. The signal μ=1 or η=1 establishes the appropriate trigger in “1”. The signal μ=0 or η=0 does not change the condition of the corresponding trigger.

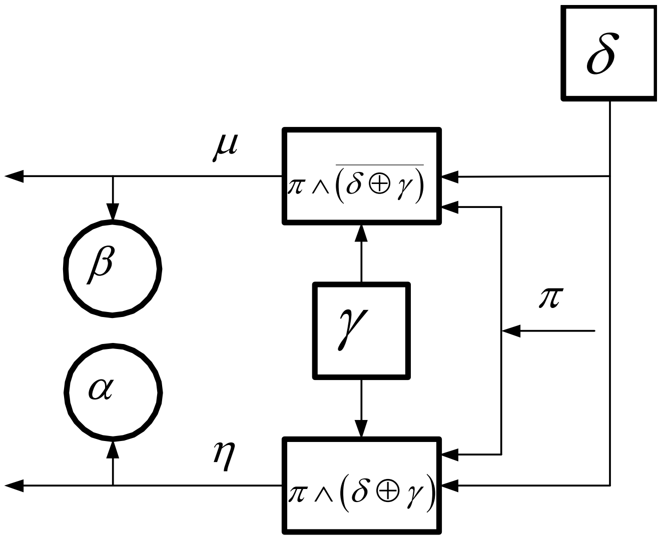

δ
μ
$\pi \wedge \overline{(\delta \oplus \gamma)}$
β
γ
π
α
η
$\pi \wedge (\delta \oplus \gamma)$

Fig. 6.8.2. Writing unit of value with given code.

6.8.3. Reading unit for path value of the given number

Fig. 6.8.3 shows a fragment of the circuit for reading the value of the path of a known number with the given basic code – see section 5.2.2.3. The path value is formed as the second basic code with ω digits. Shown in this Figure are units that calculate the μ, η signals according to specific formulae. One of these signals always equals “1” and is transmitted further in the form of signal π. The unit that calculates the ω signal writes it into the same-name trigger of the value code register. The ω signal is generated in the α or β digit that the carry signal has passed through.

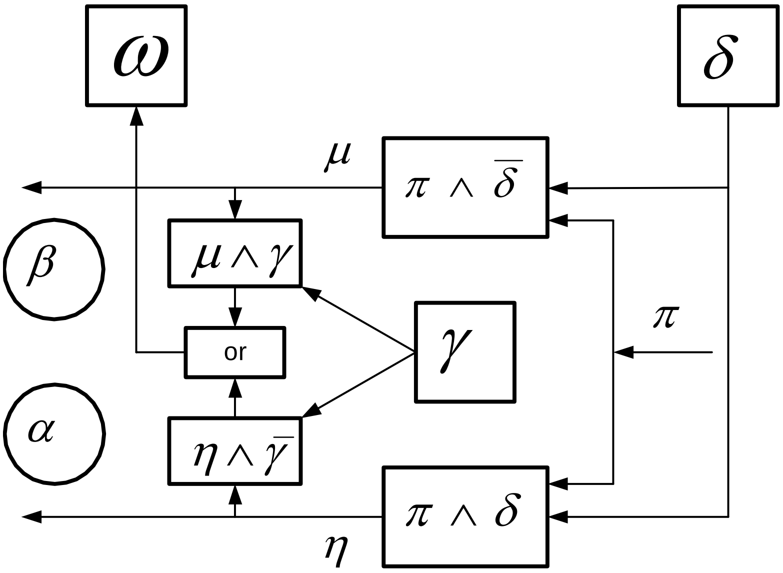


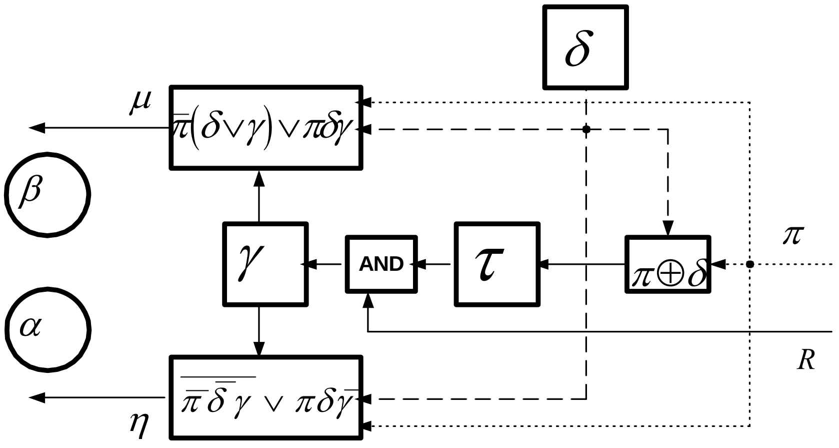
Fig. 6.8.3. Reading unit of value with given code.

6.8.4. Inverse adder

Fig. 6.8.4 shows a fragment of the adder for inverse addition of AGC to the basic code with a radix (–2) – see section 5.2.2.5. This Figure shows units that calculate the μ, η, τ signals according to specific formulae in accordance with Table 5.2.3a. One of these signals μ, η always equals "1", and is transmitted further in the form of signal π. The signal τ is written into the trigger that has the same name. The transposition enabling signal R is common for all digits and is received from the control circuit once the carry propagation through all the digits is finished. After that, the value τ is set in the digits γ.

The same circuit is used for multiplication. The difference is that the addition start signal is sent to the root vertex; during multiplication – to all vertexes of a specific tier in which α=0.

In a particular case where δ≡0, the same adder performs the function of an inverter.



δ
μ
$\bar{\pi}(\delta\vee\gamma)\vee\pi\bar{\delta}\gamma$
β
γ
AND
τ
$\pi\oplus\delta$
π
α
R
$\overline{\bar{\pi}\bar{\delta}\gamma}\vee\pi\delta\bar{\gamma}$
η

Fig. 6.8.4. Inverse adder.

6.8.5. Search unit for the first open path, its numbers and its values

Fig. 6.8.5 shows a fragment of the circuit for searching out the first open path and reading its number and its value. The number code is formed as the first basic code with δ digits, and the value code is formed as the second basic code with ω digits. The figure shows units that calculate the signals μ, η according to certain formulae. Only one of these signals may equal “1”, and is transmitted further as a π signal. If both of these signals equal zero, then the Null signal is generated, and the carry propagation process stops. Units calculating the δ and ω signals write them into the same-name trigger of the corresponding code’s register. This circuit finds the uppermost open path in the GC tree.

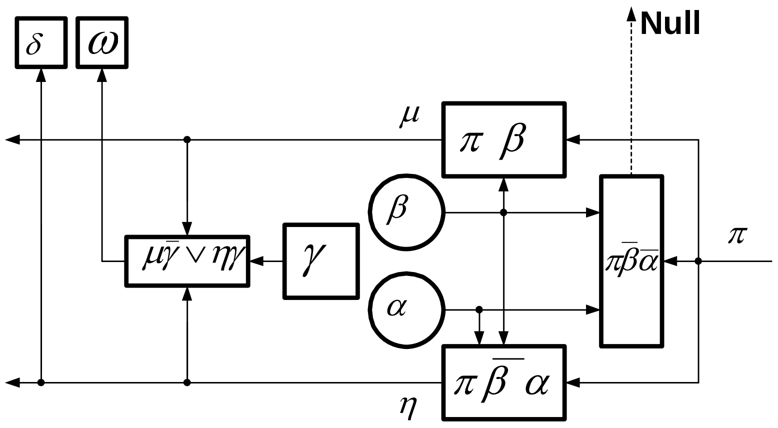
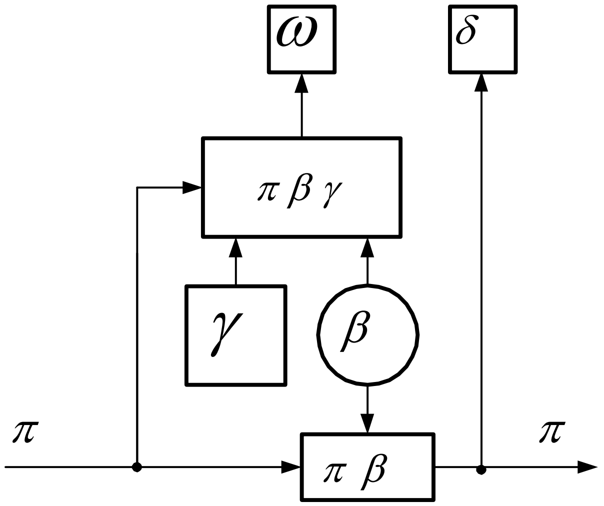


Fig. 6.8.5. Search unit for first open path, its number and value.

6.8.6. Reading unit of the number and value of the path with a given terminal vertex

Fig. 6.8.6 shows a fragment of the circuit for reading the number and value of the path ending in the given terminal vertex. Unlike the previous circuits, here the carries propagate from left to right. At the same time, the number code is formed as the first basic code with δ digits, and the value code is formed as the second basic code with ω digits. Circuits conjugated with the β and α digits are differing.



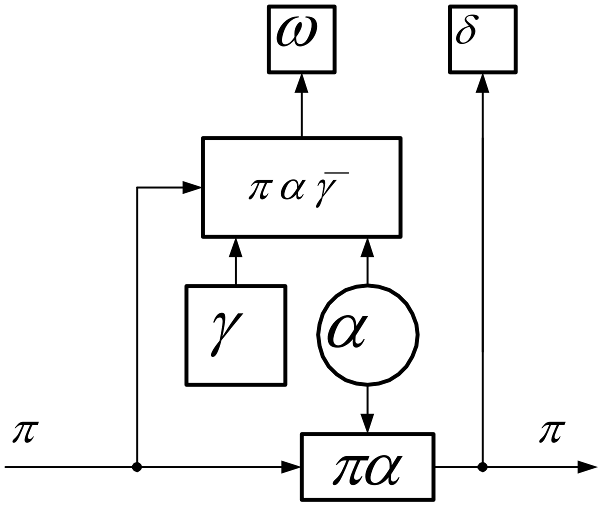


Fig. 6.8.6. Reading unit of number and value of path with a given terminal vertex.

6.8.7. Next terminal vertex search unit

This unit scans (with Carry_In and Carry_Out signals) the terminal vertexes starting with the given one (upon the InPut signal) and up to the first vertex with a value of one. The output signal is generated in such a vertex (OutPut) – see Fig. 6.8.7.

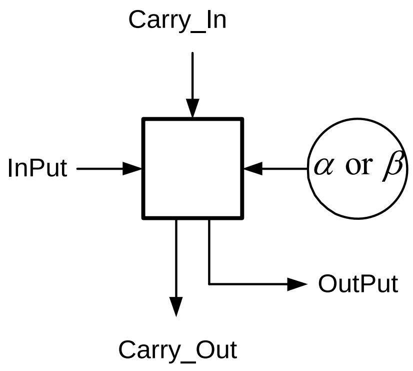


Fig. 6.8.7. Next terminal vertex search unit.

7. Comparative Analysis

This section is concerned with comparison between the characteristics of arithmetic units and random-access memory units that were presented above. Table 7.1 gives the list of the units, and Table 7.2 shows their characteristics, where

- T – reading/writing time
- S – time of attribute search by known coordinates
- R – digit capacity for random-access memory and equivalent digit capacity of AU
- A – number of operations for affine transformation
- n – digit capacity of one coordinate code
- r – digit capacity of transformation parameter
- a – total digit capacity of all transformation parameters (see (2.1.1.)
- p – space dimension
- $M=2^{pn}$ – number of points in the space - see (2.1.2)
- F – number of fragments tiers when using vertical fragmentation
- $Q=2^{f}$ – number of points in a fragment when using horizontal fragmentation
- $D=p(p-1)$ – number of adding in an affine transformation – see (2.2.1)
- In p. 1 it is assumed that the search is performed in an unordered array TRAM

Table 7.1. The list of compared units

№		
1	**TRAM**	Random-access memory in traditional performance
2	**PSSRAM**	Full specific random-access memory
3	**FSSRAM**	Specific fragmentary random-access memory
4	**SAU**	Simplest arithmetic unit
5	**MSAU**	Arithmetic unit with rectangular codes
6	**FSAU**	Arithmetic unit with fragmentary rectangular codes
7	**VAU**	Vector arithmetic unit
8	**MVAU**	Vector arithmetic unit with rectangular codes
9	**FVAU**	Vector arithmetic unit with fragmentary rectangular codes
10	**FGAU**	Arithmetic unit with fragmentary geometrical codes
11	**MGAU**	Arithmetic unit with fragmentary geometrical codes combined with random-access memory

Table 7.2. Characteristics of compared units

№		T	S	R	A
1	**TRAM**	1	$M/2$	$Mp(n+r)$	-
2	**PSSRAM**	1	1	$Mpr+M$	-
3	**FSSRAM**	F	F	$F\left(\frac{M}{F}pr + \sqrt[F]{M}\right)$	-
4	**SAU**	-	-	$7(n+r)+a$	$M(D+p^2)$
5	**MSAU**	-	-	$7M(n+r)+a$	$D+p^2$
6	**FSAU**	-	-	$7Q(n+r)+a$	$(D+p^2)M/Q$
7	**VAU**	-	-	$7p(n+r)+a$	M
8	**MVAU**	-	-	$7Mp(n+r)+a$	1
9	**FVAU**	-	-	$7Qp(n+r)+a$	M/Q
10	**FGAU**	-	-	$((pn-f)+Qpr+Q)$	M/Q
11	**MGAU**	1	1	$(Mpr+M)$	1

The performance time τ of one operation is practically independent of the type of each listed unit. Therefore the performance time of affine transformation in each of these units is $t = A\tau$. In a unit time each unit solves $z = 1/t = 1/A\tau$ problems of affine transformation. It is reasonable to define *the quality of* an arithmetic unit by the unit's volume necessary for solving a certain number of affine transformation problems, or by *relative volume* of a unit: the less is the relative volume W, the higher is the unit's quality. Clearly, the relative volume of a unit is $W \equiv R/z$ or $W=AR$.

Similarly, the quality of random-access memory may be described by the ratio of its volume to the number of access operations performed in a unit time. Speaking of a reading/writing operation, the relative volume of random-access memory is $W1=TR$. Considering the operation of a point's search in an unordered array, the relative volume of random-access memory will be equal to $W2=SR$. It would be useful to consider a given mixture of these operation, but for that the statistics of access operations must be known.

Table 7.3 shows the relative volume of all above described units.

Table 7.3. Relative volume of compared units.

№		AR	SR
1	TRAM		$\frac{M^2 p(n+r)}{2}$
2	PSSRAM		$F^2\left(\frac{M}{F}pr + \sqrt[F]{M}\right)$
3	FSSRAM		Mpr
4	SAU	$14Mp^2(n+r)$	
5	MSAU		
6	FSAU		
7	VAU	$7Mp(n+r)$	
8	MVAU		
9	FVAU		
10	FGAU	Mpr	
11	MGAU	Mpr	Mpr

Based of this table we have built a more illustrative Table 7.4 of the relative volumes of the main units.

Table 7.4. Relative volumes of the main units

№	Unit	*W=AR* for processor; *W1=TR* or *W2=SR* for RAM
1	TRAM - traditional random access memory	$W_1 = Mp(n+r)$ $W_2 = M^2 p(n+r)/2$
3	SSRAM - specific static random access memory	$W_1 = W_2 = F^2\left(\frac{M}{F}pr + \sqrt[F]{M}\right)$
5	SAU - scalar arithmetic unit	$14Mp^2(n+r)$
8	VAU - vector arithmetic unit	$7Mp(n+r)$
10	GAU – geometrical arithmetic unit	Mpr

Fig. 1.1 (in the Introduction) gives a bar graph of the quality of all considered arithmetic units with $n=r$. The measuring unit in this bar graph is $14*M$. For instance for $p=3$ the ratio of quality characteristics is **(84:14:1)**.

The relative volume $W2$ of the unit TRAM for $n=r$ is M times larger than relative volume $W2$ of the unit PSSRAM. For $F>1$ the ratio of relative volumes $W2$ of the units TRAM and FSSRAM is $\frac{M(n+r)}{2Fr}$. For example, at $n=r$ the ratio of these volumes is M/F.

The relative volume $W1$ of the unit TRAM at $n=r$ is twice as large as the relative volume $W1$ of unit PSSRAM. For $F>1$ the ratio of relative volumes $W1$ of the units TRAM and FSSRAM is $\frac{(n+r)}{Fr}$. For example, at $n=r$ their ratio is $2/F$, i.e. the relative volume $W1$ of the unit TRAM is $F/2$ times smaller than the relative volume $W1$ of the unit FSSRAM.

Let us assume now that in a given problem the reading/writing operations occur H times more often than search operations.

Then the relative volume of random-access memory should be defined by the formula $W = R(TH + S)$. This value can be found in the Table 7.2. For the units TRAM and FSSRAM the relative volume is accordingly

$$W_T = Mp\,(n + r)\left(H + M/2\right)$$

and

$$W_F = F\left(\frac{M}{F}pr + \sqrt[F]{M}\right)(FH + F) \approx FMpr\,(H + 1).$$

Ratio $\frac{W_F}{W_T} \approx \frac{FMpr\,(H+1)}{Mp\,(n+r)(H + M/2)}$.

For $1 \ll H \ll M$, $n = r$ this ratio $\frac{W_F}{W_T} \approx \frac{HF}{M}$. Thus, the relative volume of FSSRAM is smaller than that of TRAM, if $\frac{W_F}{W_T} \approx \frac{HF}{M} < 1$ or $HF < M$. The average number of tiers during vertical fragmentation is $F \approx 10$. Consequently,

> The relative volume of a specialized storage device is $\frac{M}{10H}$ times smaller than the relative volume of a traditional storage device.

References

1. H. Haberdar. Affine Transformation Example, University of Houston, 2012.
2. E. Angel, D. Shreiner. Interactive Computer Graphics : A Top-Down Approach with Shader-Based OpenGL (6th Edition), 2011.
3. Sébastien Roy, Daniel Lefebvre and Henri H. Arsenault. Recognition invariant under unknown affine transformations of intensity, Optics Communications, Volume 238, 2004.
4. Shih-Hsuan Yang, Chun-Yen Liao, and Chin-Yun Hsieh. Watermarking MPEG-4 2D Mesh Animation in Multiresolution Analysis, Computer Science, Volume 2532, 2002.
5. Khmelnik S.I., The Planar Figures Coding. Avtomatika i Vychislitelnaya Tekhnika, Ac. of Sc. of Republic of Latvia, 1970, №6 (in Russian)
6. Khmelnik S.I., Algebra of Many-dimensional Vectors and Spatial Figures Coding. Avtomatika i Vychislitelnaya Tekhnika, Ac. of Sc. of Latvia Republic, 1971, №1 (in Russian)
7. Khmelnik S.I., High-performance Search Procedures. The Third International Symposium in Information Theory, part II, Tallinn, 1973 (in Russian)
8. Khmelnik S.I, Multi-criteria Assignment Problem. Isvestiya of USSR Ac. of Sc., Tekhnicheskaya Kibernetika, №4, 1977 (in Russian)
9. Khmelnik S.I, Search Procedures with Geometric Codes, journ. "Kibernetika", Ac. of Sc., Republic of Ukraine, 1990, №6 (in Russian)
10. Khmelnik S.I, Digital Device for Image Geometrical Transformations Author's Certificate. 333573, 1972 (in Russian)
11. Khmelnik S.I, A Device for Image Geometrical Transformations. Author's Certificate 1030816, 1983 (in Russian)
12. S. Khmelnik. Method and System for Processing Geometrical Figures. International patent application under PCT. PCT/CA02/00835, WO 02 099625 A2. Priority 07.06.01.
13. S. Khmelnik. Computer Arithmetic of Vectors, Figures and Functions, Publishing «Mathematics in Computers Comp.», Moscow– Tel-Aviv, 1995 (in Russian)
14. S. Khmelnik, I. Doubson. Positional codes of complex numbersand vectors. Printed in USA, Lulu Inc., ID 10834718, 2011, ISBN 978-1-257-83907-0.

15. S. Khmelnik. A Method and System for Processing Complex Numbers. International patent application under PCT. PCT/CA01/00007, WO 01 50332 A2. Priority 05.01.00.
16. Khmelnik S.I., The Vectors Coding, journ. "Kibernetika", Ac. of Sc., Republic of Ukraine, 1969, №5 (in Russian)

Designation

Add – M-codes Adder
AGC – attributic geometrical code
AGCC – attributic geometrical complex code
ARAM - attribute traditional random-access memory
AU – arithmetic unit
CAGC – contracted attributic geometrical code
CoderPM - coder of positive P-code into M-code
C-code – complex code in complex radix
DecoderMP - decoder of M-code into P-code
Deven – one-digit decoder circuit for even-numbered digit
Dodd - one-digit decoder circuit for odd-numbered digit
DRAM - dynamic random access memory
FGAU – fragmetntary geometrical arithmetic unit
FSAU – fragmetntary scalar arithmetic unit
FSSRAM – fragmetntary specific static random access memory
FVAU – fragmetntary vectorial arithmetic unit
GAU – geometrical arithmetic unit
GC - geometrical code
Inv - M-code inverter
InvAdd – M-codes inverse adder
LP – linear part of MCF
MCF - mixed code of figure
mDecoderMP - full decoder of M-code into P-code
Meven – one-digit coder circuit for even-numbered digit
MGAU – maximum geometrical arithmetic unit
Modd - one-digit coder circuit for odd-numbered digit
MSAU – maximum scalar arithmetic unit
MVAU - maximum vectorial arithmetic unit
M-code – real numbers code in the radix “-2”
nSign – M-code sign determinant
Partitioning - partitioning unit for code’s parts
PFGAU - processor with FGAU
PGC – primary geometrical code
PMGAU - processor with MGAU
PreCoder - precoder of P-code into M-code

PSSRAM – perfect specific static random access memory
P-code – traditional code in the radix "2"
RAM - random access memory
RCS - rectangular code of scalars
RCV - rectangular code of vectors
RGP – raster geometrical processor
SAU - scalar arithmetic unit
Seven – one-digit sign determinant circuit for even-numbered digits
Sodd - one-digit sign determinant circuit for odd-numbered digits
SRAM - static random access memory
SSRAM - specific static random access memory
Sub – M-codes subtractor
TRAM - traditional random access memory
VAU - vector arithmetic unit

List of Examples

Example 5.1.1 of addition when ρ=2 \ 67
Example 5.1.2 of inverse addition with ρ=-2 \ 70
Example 5.1.3 of multiplication with ρ=-2 \ 73
Example 5.1.3a of multiplication with ρ=-2 \ 75
Example 5.1.4 of inverse addition where $\rho = j\sqrt{2}$ \ 78
Example 5.1.5 of multiplication of the imaginary part of geometrical code with $\rho = j\sqrt{2}$ \ 80
Example 5.1.6 of coding a plane with $\rho = j\sqrt{2}$ \ 84
Example 5.1.7 of deformation in the $\rho = j\sqrt{2}$ \ 88
Example 5.2.1 of addition when ρ=2 \ 96
Example 5.2.2 of inverse addition when ρ=-2 \ 98
Example 5.2.3 of of deformation when $\rho = j\sqrt{2}$ \ 103

List of Tables

Table 3.1.1. Systems of complex numbers coding \ 20
Table 3.1.2. Binary coding systems \ 20
Table 3.2.1. 3-dimensional vectors multiplication \ 21
Table 3.2.2. Compex numbers multiplication \ 24
Table 3.2.3. 4-dimensional vectors multiplication\ 24
Table 3.4.2. One-digit inverting circuit \ 30
Table 3.4.3. One-digit inverse adder circuit \ 30
Table 3.4.4. One-digit adder circuit \ 31
Table 3.4.5. One-digit subtractor circuit \ 32
Table 3.4.6.1. One-digit sign determinant circuit for even-numbered digits \ 33
Table 3.4.6.2. One-digit sign determinant circuit for odd-numbered digits \ 34
Table 3.6.1. One-digit scalar multiplication \ 39
Table 3.6.2. One-digit vector multiplication \ 40
Table 3.6.3 Carries in scalar multiplication \ 41
Table 3.6.4 Carries in scalar multiplication \ 42
Table 3.6.5. Carries in vector multiplication \ 43
Table 3.7.5.1. One-digit coder circuit for even-numbered digit \ 48
Table 3.7.5.2. One-digit coder circuit for odd-numbered digit \ 48
Table 3.7.6.1. One-digit decoder circuit for even-numbered digit \ 50
Table 3.7.6.2. One-digit decoder circuit for odd-numbered digit \ 50
Table 3.7.8. Precoder of P-code into M-code \ 52
Table 3.7.9. Partitioning Unit for Code's Parts \ 52
Table 4.2.1. Comparative features of AU \ 57
Table 4.2.2. Numerical characteristics of AU with p=2, r=6, M=10^6, n= 12, Q=256, a=72 \ 58
Table 4.2.3. Numerical characteristics of AU with p=3, r=6, M=10^6, n= 12, Q=256, a=90 \ 58
Table 4.2.4. Numerical characteristics of AU with при p=4, r=6, M=10^6, n= 12, Q=256, a=240 \ 58

Table 5.1.1a. Addition of geometrical and basic codes with ρ=2 \ 65
Table 5.1.1b. Inverse addition of geometrical and basic codes with ρ=-2 \ 68
Table 5.1.2. Geometrical code of a plane with $\rho = j\sqrt{2}$ \ 84
Table 5.1.2a. Geometrical code of a separated plane points with $\rho = j\sqrt{2}$ \ 85
Table 5.1.3. Figure deformation with $\rho = j\sqrt{2}$ \ 88
Table 5.2.1. Writing a value with a given code \ 94
Table 5.2.2. Reading the value of the path with a given number \ 95
Table 5.2.3. Addition of AGC and the basic code when ρ=2 \ 95
Table 5.2.3a. Inverse addition of AGC and basic code when ρ=-2 \ 97
Table 5.2.4. AGC inversion when ρ=-2 \ 99
Table 7.1. The list of compared units \ 138
Table 7.2. Characteristics of compared units \ 138
Table 7.3. Relative volume of compared units \ 139
Table 7.4. Relative volumes of the main units \ 140

List of Figures

Fig. 1.1. Bar graph of the relative volume of the named arithmetic \ 11
Fig. 2.2.1. The simplest arithmetic unit \ 14
Fig. 2.2.2. Arithmetic unit with fragmentary rectangular codes \ 18
Fig. 3.4.1. Multidigit algebraic addition circuit \ 29
Fig. 3.4.2. One-digit inverting circuit \ 30
Fig. 3.4.3. One-digit inverse adder circuit \ 31
Fig. 3.4.4. One-digit adder circuit \ 31
Fig. 3.4.5. One-digit subtractor circuit \ 32
Fig. 3.4.6.1. One-digit sign determinant circuit \ 33
Fig. 3.4.6.2. Sign determinant \ 34
Fig. 3.6.1. Adder in the scalar multiplication unit \ 42
Fig. 3.6.2. Adder in vector multiplication unit \ 44
Fig. 3.7.5.1. Coder of positive P-code into M-code \ 47
Fig. 3.7.5.2. One-digit coder circuit \ 48
Fig. 3.7.6.1. Decoder of M-code into P-code \ 49
Fig. 3.7.6.2. One-digit decoder circuit \ 50
Fig. 3.7.7. Full decoder \ 51
Fig. 4.1.1. Vector arithmetic unit \ 54
Fig. 4.1.2. Vector arithmetic unit with rectangular codes \ 56
Fig. 5.1.1. Binary Tree.\ 60
Fig. 5.1.2. Examples: a tree of binary digits \ 62
Fig. 5.1.2a. Carry propagation pattern in GC \ 63
Fig. 5.1.3. Example: a transposed code \ 64
Fig. 5.1.4. For example 5.1.1 \ 67
Fig. 5.1.5. For example 5.1.2 \ 70
Fig. 5.1.6. For example 5.1.3 \ 72
Fig. 5.1.7. For example 5.1.3 \ 73
Fig. 5.1.8. For example 5.1.3 \ 73
Fig. 5.1.9. For example 5.1.3 \ 73
Fig. 5.1.9a. For example 5.1.3a \ 75
Fig. 5.1.9b. For example 5.1.3a \ 75
Fig. 5.1.10. For example 5.1.4 \ 78
Fig. 5.1.11. For example 5.1.5 \ 80
Fig. 5.1.12. For example 5.1.5 \ 81

Fig. 5.1.13. For example 1.1.5 \ 82
Fig 5.1.14. Coding planar figure \ 83
Fig 5.1.15. Coding plane at *y*=3, *m*=-1, *r*=4 for example 5.1.6 \ 84
Fig 5.1.15a. Example: planar figure \ 85
Fig 5.1.15b. Example: GC tree of planar figure \ 86
Fig. 5.1.16. Deformation of a figure for example 5.1.7 \ 88
Fig. 5.2.1. Attribute Geometrical Code \ 93
Fig. 5.2.2. For example 5.2.1 \ 96
Fig. 5.2.3. For example 5.2.2 \ 98
Fig. 5.2.3a. One-digit inverse addition circuit \ 99
Fig. 5.2.4. Carry propagation pattern in complex GC \ 101
Fig. 5.2.5. For example 5.2.3: AGC of the initial figure \ 104
Fig. 5.2.6. For example 5.2.3: AGC of the deformed figure \ 105
Fig. 5.2.7. Contracted Attribute Geometrical Code \ 108
Fig. 6.0.1. MCF structure \ 112
Fig. 6.1.1. Full specific random-access memory \ 113
Fig. 6.2.1. Tiers of geometrical code and segments of linear code \ 114
Fig. 6.4.1. Fragmentary arithmetic unit \ 119
Fig. 6.5.1. Processor with a maximal arithmetical unit \ 121
Fig. 6.6.1. Processor with fragmentary FGAU \ 124
Fig. 6.8.1. Writing unit of number with given code \ 131
Fig. 6.8.2. Writing unit of value with given code \ 132
Fig. 6.8.3. Reading unit of value with given code \ 133
Fig. 6.8.4. Inverse adder \ 134
Fig. 6.8.5. Search unit for first open path, its number and value \ 134
Fig. 6.8.6. Reading unit of number and value of path with a given terminal vertex \ 135
Fig. 6.8.7. Next terminal vertex search unit \ 136