# Alluvion –
# A Language for Computer Arithmetic Algorithms

Rade Kutil*

**Abstract.** *Computer arithmetic algorithms usually are represented by circuit diagrams which do not seem to be the first choice for* algorithms. *Rather, this is a consequence of the fact that these algorithms are implemented as circuits in almost all cases. However, it is possible to represent these algorithms in a procedural way, as is shown in this work. A specialized programming language is developed to formalize these algorithms efficiently. Advantages of this approach are shorter representations, easier understandability, code reuseability and better flexibility and adaptability. Real circuits can automatically be constructed through interfaces to hardware definition languages.*

## 1. Introduction

Computer arithmetic algorithms [2, 4] include solutions to arithmetic operations such as addition with all sorts of carry propagation enhancements, multiplication, division, square roots and elementary mathematical functions applied to numbers represented in the binary number system. Representations such as integer 2-s complement for negative numbers, fixed point and floating point numbers are possible.

They are usually implemented as hardware circuits [3] with VLSI methods or with FPGAs [6]. Therefore, such algorithms are usually represented by logic circuit diagrams. All calculations have to be done through elementary logic functions such as *and*, *or* and *not*.

Similar to normal algorithms, the efficiency (complexity) of a computer arithmetic algorithm is defined in terms of space and time, where space is basically measured by the number of logic gates needed in the hardware implementation and – mostly more important – time is measured by the biggest delay of all output bits. This delay is the sum of all gate delays in the longest path of gates from an input pin to an output pin. It largely determines the clock frequency of processing units. As with normal algorithms, time complexity can be reduced by accepting higher complexity in space, i.e. more gates working in parallel.

It is important to note that the fanout, i.e. the number of gate inputs that must be driven by a single gate output, usually imposes further limits on the algorithm. Although fanout limits can be avoided by the use of buffers, these cause additional delays and, therefore, reduce the performance.

The representation of computer arithmetic algorithms as circuit diagrams can be messy and hard to understand for large and complex algorithms. However, these algorithms can be written in a procedural form. The problem with this is the loss of parallelity which makes it hard to transform an

*Department of Scientific Computing, University of Salzburg, Austria, email: rkutil@cosy.sbg.ac.at

algorithm into its circuit representation. This work shows how restrictions on the language can avoid this problem. Strict separation of data and control variables allows the generation of a fixed data flow graph through evaluation of the control flow (calls, loops, ...) which depends on control variables only. The resulting data flow graph is a representation of the corresponding circuit.

Another important aspect is the use of alternative representations of knowledge for educational purposes. In fact, the language proposed in this work was developed for teaching computer arithmetic algorithms. This approach was chosen for the following reasons. The language has to be easy to understand. Therefore, it has to be simple and to provide only necessary features. In this way, short and clear formulations of algorithms can be found, which enables to compare the algorithms easily. They are reproducible and adaptable. Interactive usage can improve the learning process as one can quickly test modifications.

Traditional hardware description languages such as Verilog [5] and VHDL [1] use a more conventional approach by describing the algorithms directly in terms of data flow. Although design units can be arranged as simple chains and arrays, this does not provide the flexibility and comfort of iteration and selection statements in programming languages. *Alluvion* represents an approach to combine the two worlds of hardware and software to enable unified development.

## 2. Basic Syntax

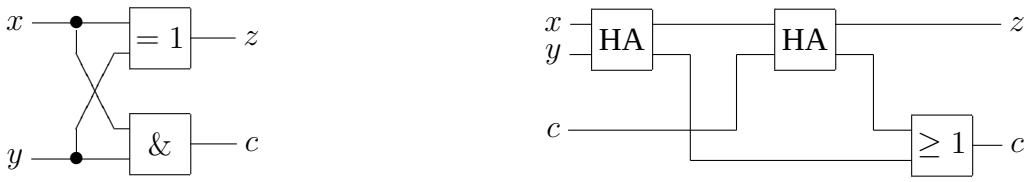
One major idea of Alluvion is to provide an easily comprehensible way to describe algorithms. Therefore, syntactical constructs were chosen that are well known from existing programming languages such as C or Java. The following example demonstrates the basic syntax of an Alluvion program.

```
(c, z) = HalfAdd (x, y)
{
  z = xor (x, y);
  c = and (x, y);
}

(cout, z) = FullAdd (x, y, cin)
{
  (c1, s1) = HalfAdd (x, y);
  (c2, z) = HalfAdd (s1, cin);
  cout = or (c1, c2);
}
```

In this example two functions are defined. The first one takes two arguments, i.e. two bits `x` and `y`, and performs a half adder operation. It calculates two output bits `z` and `c` (the carry bit) by calling the built-in functions `xor` and `and`. The second function is a full adder. It is implemented by two calls of the half adder function. Three intermediate variables (`c1`, `s1`, `c2`) are created by specifying them as part of result vectors. They do not have to be declared.

Figure 1 shows the circuit representation of the program example. A connection is drawn wherever the output of a function is read as input to another function. Note that this can easily produce an enormous




(a) half adder (HA)

(b) full adder (FA)

**Figure 1. Atomic adder units in circuit representation**

amount of crossing lines, where Alluvion prevents confusion when accessing named variables at any point in a function.

The program can be called by a command line such as `FullAdd('1','0','1')`, leading to the result `cout` $= 1$ and `z` $= 0$ and some program statistics such as gate count ($= 5$) and total gate delays ($= 3$ for `cout`) which are discussed later on in this paper.

## 3. Data Variables and Control Variables

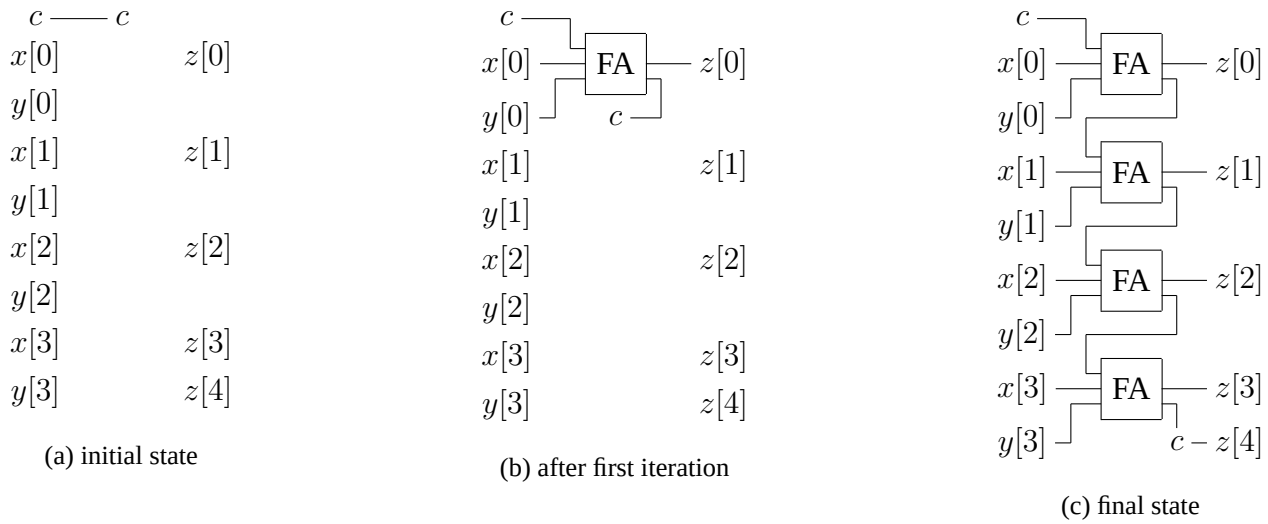
The most important decision in the design of Alluvion is to distinguish data variables and control variables. While the former basically correspond to electrical connections (often called signals in HDLs), the latter parameterize the algorithm and, as a consequence, the resulting circuit. Control variables appear as signal counts, signal numbers, stage depths and other characteristic values in those circuits.

Control variables can influence values of data variables since control variables control how data variables are created – hence the name. On the other hand, data variables cannot influence control variables since the values of control variables are considered constant with respect to the data flow in data variables. This is important in order to avoid the need to translate control flow statements into a circuit representation, which would be difficult since there is no “electronic while unit” for instance.

During the evaluation of an Alluvion program, control variables are not constant, though. This apparent contradiction can be resolved by imagining two phases of evaluation. In the first phase functions are instantiated following the control flow of a given program. This can be done with the sole knowledge of control variable values. In the second phase data values are supplied to function instances and their data variables in the order of instantiation.

The following example demonstrates all this. It implements the usual ripple-carry adder.

```
z[0,n] = Add<n> (x[0,n-1], y[0,n-1], c)
{
  for i = [0,n-1]
  {
    (c, z[i]) = FullAdd (x[i], y[i], c);
  }
  z[n] = c;
}
```




(a) initial state

(b) after first iteration

(c) final state

**Figure 2. How a loop creates an array of units and connections in the Add<4> function**

In this example `n` and `i` are control variables. Control variable `n` denotes the word size of the two words `x[0,n-1]` and `y[0,n-1]` that should be added. It is passed to the function not as normal argument but in <> brackets similar to generic parameters in C++ . The number of input and output bits, which are specified as arrays, depends on this control variable. Thus, a function can be parameterized by the caller so as to process arbitrary sized data. Note that setting `i = c` would produce an error because data variables cannot be assigned to control variables.

The control variable `i` serves as bit index. It counts from the least significant bit to the most significant bit in a `for` loop. At each bit position the addition is performed by calling the above `FullAdd` function.

Note that the current carry bit is passed to `FullAdd` and is immediately overwritten by the new carry as it is output by the function. What this means for signals in the resulting circuitry, is explained in the next section.

## 4. Writing to Variables Creates New Nodes

A circuit can be viewed as a set of elementary units with input and output connectors, and a set of nodes. Each node is connected to exactly one output connector and one or more input connectors.

Alluvion virtually maps function instances to elementary units or circuit blocks. These blocks are connected corresponding to input and output arguments. Whenever a function instance reads from a variable, it is – in terms of circuitry – connected to the function instance that last wrote to the variable. This means that variables serve as connection nodes.

However, this is not entirely true. In fact, it is the value of a variable that corresponds to a single node. To see the difference, consider a variable that is overwritten with a new value. This value is produced by the instance of a function. The output of this instance should not be connected to the node that is currently associated with the variable. Instead, a new node has to be created. As the variable gets the new value, it is associated with the new node.

If a variable is copied to another variable (as in `z[n] = c`), two variables contain the same value and, consequently, are associated with the same node. Therefore, a node is being manipulated until the last associated variable is overwritten or runs out of scope, i.e. its function instance is destructed.

This principle allows Alluvion to pass signals from one loop iteration to another as well as into and out of a loop, as shown in Figure 2. The carry bit `c` in the above `Add` function is first passed to the function and then used in the first iteration. Each iteration generates a new carry bit and passes it on to the next iteration by overwriting `c`. In the circuit diagrams in Figure 2 the current value of the carry bit is represented by the non-connected output $c$. The final carry bit is passed from the last iteration to the function result vector by copying `c` to `z[n]`.

## 5. Statistics

If the above adder function is called by the command line

```
Add<8> ('11010000', '00001111', '0')
```

then Alluvion produces the following output:

```
z[0,n]:
  [8]   0       (17, 7)
  [7]   1       (16, 6)
  [6]   1       (14, 4)
  [5]   0       (12, 11)
  [4]   1       (10, 9)
  [3]   1       (8, 7)
  [2]   1       (6, 5)
  [1]   1       (4, 3)
  [0]   1       (2, 2)

Statistics:
  and: 16 instances, max. fanout = 1
  or: 8 instances, max. fanout = 2
  xor: 16 instances, max. fanout = 2
  HalfAdd: 16 instances, 32 gates (32 internal, 0 external)
  FullAdd: 8 instances, 40 gates (8 internal, 32 external)
  Add: 1 instances, 40 gates (0 internal, 40 external)
```

The first part represents the output vector. It consists of the array `z[0,8]`, where `z[8]` is the outgoing carry bit. Each line shows the index of the array element, it's bit value and the two delay values in parentheses. The first one is the total gate delay, i.e. the number of gate levels involved in the calculation of the output bit. The second delay is the effective delay. It represents the time (measured in gate delays) at which the output bit is guaranteed to contain the final value. The effective delay can be smaller than the normal gate delay depending on input data. For instance, the expression `or('1',and('0','0'))` yields a gate delay of 2 but an effective delay of 1 since the value of the `and(..)` subexpression does not change the output of the `or`.

The second part shows overall statistics. First, the number of times a function is instantiated is counted. For built-in functions this shows the number of gates of a certain type used in the corresponding circuit. For other functions, the total gate count (i.e. the sum of the gate counts of every instance) is calculated additionally, divided into the number of internal and external gates. Internal gates are created through the use of built-in functions directly in the function. External gates are created when another function is called. The external gate count is also contained in the total gate counts of the called functions.

## 6. Implementation

Although the semantics of Alluvion is closely related to circuits, Alluvion does not produce any kind of circuit representation when a program is evaluated, at least not in its current implementation. Instead, each variable contains additional information about virtually associated circuit nodes which is sufficient to calculate program statistics. Needless to say that the data values (bits) can be calculated in usual language interpreter style, i.e. by replacing variable's values on assignment. In the following the way each of the statistical values is computed is explained.

Each node of a circuit has a certain gate delay, i.e. the longest distance to an input node in terms of intermediate gates. As values of variables correspond to nodes, Alluvion associates a gate delay to each value. Whenever a built-in function (a gate) is called, it calculates the maximum gate delay of all input values, adds one to it and writes the result together with the computed data value into the destination variable. If a value is copied from one variable to another, its gate delay is also copied. This is especially important for function calls when input values are copied into local variables of the called function's instance. Output values are handled the same way. Thus, gate delays are passed into and out of functions.

The effective delay is calculated in a similar way. The only difference is that the effective output delay is not necessarily calculated via the *maximum* of effective input delays. Actually, the behaviour depends on the input values themselves. If, for instance, both input values of an and-function are zero, the *minimum* of the input delays is used.

Fanout values are more complicated to determine. The reason is that a value corresponding to a single node can be spread over several variables if it has been copied. If each of the copied values would count by itself the number of times it is supplied to a built-in function, all these counts must be added in the end, which is difficult because corresponding values are hard to find and possibly already destroyed because they went out of scope or have been overwritten. The solution is to create a node object for each new value and let the values reference the object. Copied values reference the same object. Now, the node object can count the input uses for all corresponding values. Node objects can be destroyed after the last referencing value is destroyed.

Finally, gate counts can easily be determined by counting calls to built-in functions. Each called function has to return its gate count which can then in turn be added to the gate count of the calling function.

## 7. Example

To see how Alluvion works for more complicated algorithms than the ones we have seen so far, let us look at the well known carry look-ahead add algorithm.

```
(g, p) = GP_Group<n> (x[0,n-1], y[0,n-1])
{
  if n > 1
  {
    m = n/2;
    (g0, p0) = GP_Group<m> (x[0,m-1], y[0,m-1]);
    (g1, p1) = GP_Group<n-m> (x[m,n-1], y[m,n-1]);
    g = or (g1, and (g0, p1));
    p = and (p1, p0);
  }
  else
  {
    g = and (x[0], y[0]);
    p = xor (x[0], y[0]);
  }
}

z[0,n] = CLA_Add<n,k> (x[0,n-1], y[0,n-1], c)
{
  for l = [0,n-k,k]
  {
    (cdead, z[l,l+k-1]) = Add<k> (x[l,l+k-1], y[l,l+k-1], c);
    (g, p) = GP_Group<k> (x[l,l+k-1], y[l,l+k-1]);
    c = or (g, and (c, p));
  }
  z[n] = c;
}
```

The function `CLA_Add<n,k>` implements a carry look-ahead adder for `n` bits. It groups the `n`-bit words into blocks of size `k` (e.g. `x[l,l+k-1]`) and applies the usual ripple-carry adder to the blocks. As the maximum gate delay primarily depends on the propagation of the carry bit, the carry look-ahead algorithm ignores the carry bit that is output by the `Add` function. Instead, it computes carry bits through `c=or(g,and(c,p))`, where `g` is the generate-bit that indicates whether the current block generates a carry bit, and `p` is the propagate-bit that indicates whether the the current block propagates an incoming carry bit. These two bits are calculated by the `GP_Group<n>` function. This function is implemented in a recursive way in order to minimize the gate delay of its outputs. It first calculates the GP-bits for the lower and upper half of the block and then combines the results. This leads to a binary tree of gate blocks. The height of this tree depends logarithmically on the word length `n` and, thus, is optimal in terms of gate delay.

When the carry look-ahead function is called for 16 bits in groups of 4 (`CLA_Add<16,4>`), it yields a delay of 17 for `z[15]` and a gate count of 156, while the ripple-carry adder (`Add<16>`) yields a delay of 32 and a gate count of 80.

## 8. Conclusion

Alluvion is a simple and yet powerful language that helps to design and convey computer arithmetic algorithms. Instead of messy diagrams and lengthy explanations it provides short and precise representations of the algorithms. Alluvion calculates characteristic values such as gate delay and gate count. Therefore, one can easily test new and modified algorithms without having to develop a full circuit. Because algorithms are usually easier to understand in their procedural representation, Alluvion can also be used in education.

## References


[1] P. J. Ashenden. *The Designer's Guide to VHDL*. Academic Press, 2002.

[2] I. Koren. *Computer Arithmetic Algorithms*. A.K. Peters Ltd., 2001.

[3] M. Morris Mano and C. R. Kime. *Logic and Computer Design Fundamentals*. Prentice Hall, 2000.

[4] A. R. Omondi. *Computer Arithmetic Systems: Algorithms, Architecture, and Implementation*. Prentice Hall, 1994.

[5] S. Palnitkar. *Verilog HDL*. Prentice Hall, 2003.

[6] B. Zeidman. *Designing with FPGAs and CPLDs*. CMP Books, 2002.