MATHEMATICAL ASPECTS OF VLSI DESIGN

Martin Rem
Eindhoven University of Technology
and
California Institute of Technology

```
0 0 1 0   1 1 0 1   0 0 0 1
1 0 0 0   0 0 0 0   0 1 0 0
0 0 1 1   1 0 1 0   1 0 0 1
0 0 1 1   0 0 1 0   1 1 1 1
1 1 0 1   1 1 0 1   1 0 0 0
1 1 1 0   1 1 0 1   0 1 0 1
0 0 1 0   0 0 1 1   1 1 1 0
```

The above is a computer program, written in the way we would program in the fifties. The program is represented in the very same way it is stored in the computer: it is the lowest level description of a program. Nowadays a program will be written in a notation more like the following.

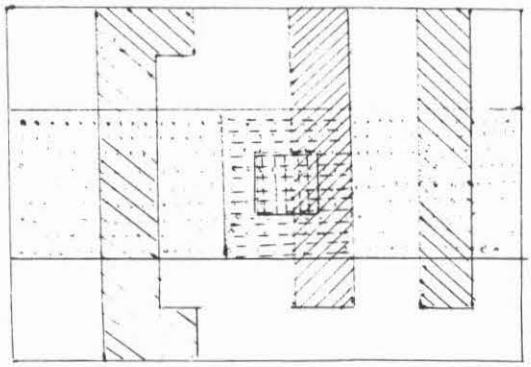
```
do x > y → x:= x - y
 ▯ y > x → y:= y - x
od
```

Although the program is written in a modern notation the algorithm it expresses is quite old. It actually dates back to the Greeks: it is Euclid's algorithm to determine the greatest common divisor of two numbers, x and y in this case. We can write our programs in such a clear way because we can make compilers that transform them into the required binary code. We don't want to know what binary code is produced, we don't even wish to know the binary code. Nor do we want the compiler to generate any messages that refer to the binary code. It should behave as if the above text is directly executed and all (error)messages should be phrased in terms of that program text.

There is more to it than just clarity. The program is expressed in well-defined constructs, each construct having a well-defined meaning. That

allows us to prove properties of our programs by which we can gain understanding in what is involved in programming and by which we can raise the confidence level of our products (cf. [2]). This enables us to construct larger programs or systems in a correct way. These larger systems will then consist of a hierarchy of smaller systems.

This is all very well-known, but let us now look at the following picture.



The above is a chip layout, or at least part of it. It also expresses some computing system, a program if you like. As in the case of the binary code it is represented in the same way it can actually be found in the computer. It is again the lowest level description of a system. But it is still the level at which we design. You may actually encounter designers drawing these figures with colour pencils on large sheets of paper. In a more modern environment you may find television screens drawing the figures for them, but it is in terms of these pictures that the designer understands his system.

The moral of this observation should be clear. We wish to have an algorithmic notation for computing structures in which we can express what should happen rather than how it should happen, together with a compiler that, if we so desire, can generate the chip layout: a silicon compiler.

+ + +

There are a number of respects in which chips are new. I want to mention three of them.

(1) New balance between logic and communication.

VLSI provides a homogeneous medium in which both logic and storage can be realized. The transistors come almost for free, it is the "wires", the communication, that determine the cost, both in area and in speed, of the chip [8]. The consequence is that traditional switching theory and complexity theory are not directly applicable to VLSI design as they don't take communication requirements into account.

(2) Invitation to high concurrency.

The expensive communication and the uniform technology form an invitation to introduce many local computations that are executed concurrently and that jointly carry out the required computation. The idea is to do the operations where the arguments are, rather than shipping the arguments to processing units.

The design of such an ultraconcurrent computation is not an easy task. In this respect VLSI came too early: we are beginning to understand the theory of sequential programming, but we still have only a rudimentary knowledge of concurrent programming.

(3) Geometrical composition of constructs.

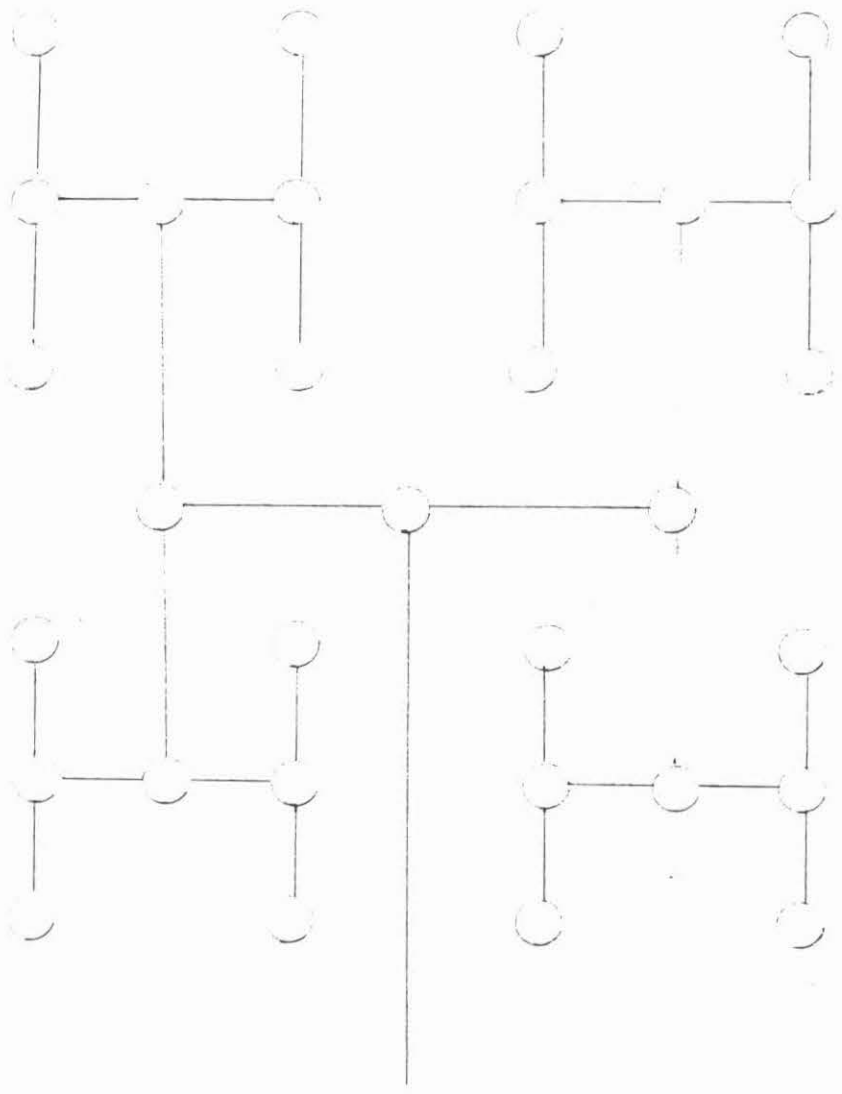
In programming we are used to think in terms of functional composition of constructs, compositions like recursion and the constructs of structured programming. We are beginning to understand them, but now there comes an additional constraint: the structures have to be mappable unto the plane. The structures should, therefore, be regular and they somehow must "fit". Ideally, they should resemble those plane-covering drawings by M.C. Escher.

The number of regular structures is limited. I shall mention some of them, more or less in the order of decreasing mappability unto the plane.

a. vector (pipe line)
b. ring
c. matrix
d. binary tree

In a. the maximal distance (in number of connections) between any two ele-

ments is n, or $n - 1$, for n elements. In b. it is $n/2$, in c. $\sqrt{n}$, and in d. $\log_2 n$. One may wonder whether the tree is mappable unto the plane. It turns out that that is not too bad.



The above picture must remind one of an Escher drawing. It is a binary tree of 5 levels and hence 31 nodes, 16 of them being leaves. Notice that the arcs get longer towards the root. That is fortunate as most of the arcs are at the lower levels, i.e. towards the leaves of the tree.

e. boolean k-cube

Let n be 2^k, number the elements 0 through $n - 1$ and write every element number in binary notation. An element is connected to every other element whose number is at Hamming distance 1, i.e. differs in one bit.

Every element has, consequently, $\log_2 n$ (=k) neighbours. This scheme is still mappable unto the plane, although not as well as the binary tree. Like the binary tree it has a maximal distance of $\log_2 n$. An important difference with the binary tree is that the cube does not have a designated root and this may prevent the congestion problems that may very well occur at the tree's root. However, the fact that the number of neighbours depends on the total number of elements is an awkward property: it precludes the modular composition of such a network. A scheme that looks like the cube without having this property is the following one.

f. perfect shuffle

Again the element numbers are coded in k (=$\log_2 n$) bits, but now an element is connected to four neigbours, viz. those with which it has k - 1 consecutive bits, i.e. all but the first all or all but the last bit, in common. Again the maximal distance between any two elements is $\log_2 n$. All nodes are equivalent, a property it shares with the cube. The problem with the perfect shuffle is that I don't know how mappable it is unto the plane. I have my doubts there.

+ + +

One of the problems with chips, nowadays, is that their initial design costs are very high. The consequence is that only those chips are produced for which a large market is expected. This phenomenon, of course, impedes progress. The hope is that the advent of the silicon compiler mentioned earlier will resolve this unfortunate situation. It will then become feasible to build small quantities of special purpose chips.

More interesting is the design of highly concurrent general purpose computing engines. In such a machine the computing elements will be connected by some pattern of "wires". How does one program such a machine? Does the programmer map his computation explicitly unto the connection pattern provided? Or does the programmer use an algorithmic notation, a programming language if you prefer, that guarantees the mappability of his computation unto the connection pattern? It seems that the latter solution is to be preferred.

The ideal is to have a uniform notation for computations. From the notation

one should not be able to tell whether the computation is meant to be

- a chip layout
- a computation for a graph of communicating processes like, e.g., a tree machine, or
- a computation for a sequential machine.

The only thing expressed by the notation is the computation and there should be compilers for all three realizations above.

An important problem is finding such a uniform notation for computations. A number of proposals have emerged recently. To mention just a few of them:

- Actors [4]
- Associons [7]
- Data driven nets [1]
- Communicating sequential processes [5]

The latter one is a rather nice notation and I would like to show an example of it. This particular one was written by David Gries. It expresses a computation of all primes less than 10000, using 102 processes and a print process. The code of the print process is not shown. It is again an old algorithm: the sieve of Eratosthenes.

The notation is a blend of Dijkstra's guarded commands [3] and synchronized communication. The sending and receiving of data are represented by an exclamation point and a question mark respectively. A matching pair of communication commands, one in the sending process and one in the receiving process, is executed simultaneously.

Each process SIEVE(i) sends one prime (the i-th prime) to the print process and sieves all multiples of that prime out of the stream of numbers it receives from process SIEVE(i-1). The stream is generated by process SIEVE(0).

```
SIEVE(i: 1 .. 100)::
      SIEVE(i-1)?p; PRINT!p;
      mp:= p;
      do SIEVE(i-1)?m →
         do m > mp → mp:= mp + p od;
         if m = mp → skip
          [] m < mp → SIEVE(i+1)!m
         fi
      od

SIEVE(0)::
      PRINT!2;
      m:= 3;
      do m < 10000 → SIEVE(1)!m; m:= m + 2 od

SIEVE(101)::
      SIEVE(100)?p; PRINT!p
```

What all these proposals lack is the notion of local computation. Every process, actor, or net element may in principle communicate with every other. To remedy this I propose the following concept of hierarchical processes, that takes locality into account.

> A process consists of a program, a state space, an initial state, and a (possibly empty) set of subprocesses.

This is a recursive definition defining a hierarchy of processes. It thus maps naturally unto a tree. The state space is the set of all possible states of the process. The program consists of sequencing primitives and instructions. Only instructions can change the state of the process. Analogous to [5] communication is performed pairwise synchronized. If P is a subprocess of Q then Q is called the environment of P . Two processes having the same environment are called coprocesses. Communication may take place only between coprocesses or between a process and its environment. The proposal is, therefore, more general than just a tree. In a tree we only have communication between a process and its environment. I am proposing to allow "horizontal" communication between processes with the same environment as well. The hope is that this more

general scheme resolves the congestion problem at roots and is still restricted enough to be in general well-mappable unto the plane.

A process' instruction is either private or public. All non-communicating instructions and communicating instructions with subprocesses are private instructions of the process. (The latter ones are public for the subprocesses.) Communicating instructions with coprocesses or with the environment of a process are public instructions. (The latter ones are private for the environment.)

A hierarchy is the only way to build complex systems with a high confidence level. They enjoy the nice property that we can prove assertions about the system by recursion over the hierarchy: assuming that the assertion holds for the subprocesses we prove that it holds for the process itself as well. During this proof we don't look inside the subprocesses, we only use their public instructions. Nor do we look at the coprocesses or the environment of the process, we only use the process' private instructions. That seems to be the only way to keep complex structures understandable.

We are as a matter of fact quite lucky here. There are physical reasons [6] why we want to design hierarchical systems, but also because of properties like understandability and inherent simplicity we wish to have hierarchies. What is mathematically attractive turns out to be physically attractive as well. Is this a violation of Murphy's law?

References

[1] Davis, A.L. "A maximally concurrent, procedural, parallel process representation". University of Utah, Salt Lake City, Utah, 1978.

[2] Dijkstra, Edsger W. "A Discipline of Programming". Prentice-Hall, Englewood Cliffs, N.J., 1976.

[3] Dijkstra, Edsger W. "Guarded commands, nondeterminacy and formal derivation of programs". Comm. ACM 18,8 (August 1975), 453-457.

[4] Hewitt, Carl & Henry Baker. "Laws for communicating parallel processes". Information Processing 77, North-Holland, Amsterdam, 1977, 987-992.

[5] Hoare, C.A.R. "Communicating sequential processes". Comm. ACM 21,8 (August 1978), 666-677.

[6] Mead, Carver A. & Martin Rem. "Cost and performance of VLSI computing structures". Japan-USA Computer Conference, San Francisco, 1978.

[7] Rem, Martin. "Associons and the Closure Statement". MC Tract 76, Mathematical Centre, Amsterdam, 1976.

[8] Sutherland, Ivan E. & Carver A. Mead. "Microelectronics and computer science". Scientific American 237,3 (September 1977), 210-228.