MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Computation Structure Group

Memo No. 28

GENERALIZED PACKAGE NOTATION

by

Fred Luconi

This memo attempts to define the syntax of a digital system design language. Many such languages have been discussed in the literature* Most of these languages represent registers and register transfer gating as basic to the logic design process. Some are more oriented towards simulation of the described structure than others, but all tend to allow easy specification of combinatory logic in terms of Boolean equations while treating the sequentially operating flip-flop as a special case. There are several problems with such approaches. First of all, the components used in modern design are not AND, OR, NOT-gates and flip-flops but highly complex combinatorial and sequential cells having several input and output terminals and providing a variety of functions. Designs must therefore be transformed either manually or automatically from the design specification to a specification reflecting the technology used. Another major problem arises if the design specification is to be used as input to a simulator for design checkout. System designs are usually the result of the cooperating efforts of a team of designers each working on a subsystem.

---

* Burnett, G. J., "A Design Language for Digital Systems", M.S. Thesis at M.I.T.; August, 1965.

Schlaeppi, H. P., "A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)" IEE Transactions, Vol. Ec-13, pp 439-448; 1963.

Schorr, H., "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language", IEE Transactions, Vol. EC-13, pp 730-737; 1964.

Proper simulation of any subsystem usually requires some modelling of the subsystem's environment. Moreover the various component subsystems at any one time may exist in a variety of developmental stages. One may only exist as an initial functional specification. Another may have been broken down into a more detailed block diagram representation, while still another may have been completed and exists as description of interconnected logical components. To allow subsystems at each of these stages of development to be represented within the same design language and to be "executed" (possibly concurrently) by a single simulation program is a problem which has been insufficiently dealt with.

Following the aforementioned arguments we will establish the following criteria for a design language:

1) The language should be easy and natural to use in the specification of logic at various stages of development. Yet descriptions must be precise, unambiguous and organized for machine simulation.
2) The language should be flexible enough so that components implemented in future technologies may be used in design.
3) The language must allow designs to be subdivided into subsystems with specified interfacing. A hierarchical language structure could allow referencing by name, parts of a system which have been previously defined.

Generalized Package Notation, GPN

GPN is a notation which involves the writing of "package" equations rather than Boolean logic equations and register transfer notation. The basic unit, the package, is considered to be any collection of logic elements

which are assumed to be grouped together as a named entity. Thus a package can be a single NAND gate in detailed design where each NAND gate represents a separate physical unit. Alternately a package can be an entire subsystem, e.g., an integrated circuit, an adder, or a CPU. Digital systems can thus be described as a network of interconnected packages.

Associated with every package in a design is a name which can consist of one or two components as shown:

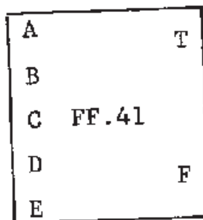
<Package Name>::= <Type Name> | <Type Name> . <Instance Name>

The type name is a designation for the particular logical function defined for the package. Examples of type names might be NAND for a nand-gate, FF for a flip-flop, or ADD for a combinational adder. If more than one package of a given type is used in a design, then each representative must possess a separate INSTANCE NAME to uniquely identify it.

All but the most trivial packages will have a number of input/output terminals. A particular terminal can be referenced by the following notation.

<Terminal Name> ::= <Package Name> : <Terminal Designation>

For example, a type FF flip-flop might have the following package input/output designations:



Future reference to this particular package would appear as FF.41. Reference to the "true"-output of this package would appear as FF.41:T

The simple package equation is a specification of the input to a particular package. The package name is written on the left hand side of an equal sign and the right hand side contains the list of input signals and the particular terminal to which each signal is connected. Formally a simple package equation can be defined as follows:

<Simple Eq> ::= <Package Name> =
{ <Terminal Designation> (<Terminal Name> {, <Terminal Name> }*) }†;

Note that several signals may be connected to a single terminal such as at a common-collector tie point.

Figure 1 shows a three-bit binary counter along with the corresponding GPN equations. The N type package contains four two-input NAND gates and the FF-type flip-flops are of the clocked, gated-input types.

For those who feel that the notation isn't very "graphical" in its presentation of information, figure 2 defines the identical circuit but a few syntax changes have been made. ← is substitutable for =. Commas, asterisks, and dashes are all interchangeable. For type FF packages, the terminals have been renamed as follows A → set 1, B → set 2, D → Reset 1, E → Reset 2. In some sense figure 2 may be more easily interpreted, but such changes to the language are simply "syntactic sugar" and do not alter the concepts involved. However it is freely admitted that syntax must be developed carefully if our first criterion is to be satisfied.

A system of simple package equations is not enough to specify a design. Since it is our desire to make as many details of the design as explicit as possible, no assumptions will be made about the operation of gates, flip-flops or any package type. Rather, the operation of each package type used in a design must be defined in detail. This definition is specified in the following format:

<u>define</u> <Package Name> [<list of terminal names>];

$$\left\{\begin{array}{c}\text{Specification of} \\ \text{package operation}\end{array}\right\}$$

<u>end</u> <u>definition</u> ;

The specification of package operation may take many forms. If the design specification only provides a format of communication between designers, then the specification of gates, flip-flops, and other basic building blocks may only refer to the circuit type used. If, however, it is desired to use GPN equations as direct input to a simulation routine, then specification of basic building blocks can take the form of subroutines for simulating the operation of the elements. A third alternative is to define a package by a system of package equations.

From the above discussion we can see that a digital system description appears as a hierarchical structure of package definitions. For example, an adder can be defined in terms of a system of equations using half-adder packages; the half-adder packages can be defined as a system of equations

using NAND-gates; the NAND-gate packages can be represented as primitives (subroutines for simulation purposes). Such an adder package can then be used as simply another package type in the construction of a multiplier.

The hierarchical structure of a digital system description enables the construction of a relatively simple simulator which need only operate on an interconnection of consistently defined packages according to their definitions. To speed simulation or to allow simulation when not all subsystems have been designed in detail, package definitions at any level can be replaced by simple, fast running modelling subroutines. The accuracy of simulation can be changed by merely altering the definitional subroutines.

Before terminating this discussion of basic GPN with a final example, (see Fig. 3) a simple embellishment is made to show one of the possible extensions to the language which simplifies design but removes none of the programmability of the language. The addition is the iteration statement which is constructed as follows:

<u>for</u> <integer ident> <u>in</u> <integer> <system of package equations> ;

3-bit Counter

```
FF.1 =  A(FF.1:F),  B(COUNT),  C(CLOCK),  D(FF.1:T),  E(COUNT)
FF.2 =  A(FF.2:F),  B(N:J),    C(CLOCK),  D(FF.2.T),  E(N:J)
FF.3 =  A(FF.3:F),  B(N:L),    C(CLOCK),  D(FF.3.T),  E(N:L)
N    =  A(COUNT),   B(FF.1:T), C(N.I), E(N.J), F(FF.2:T), G(N.K)
```

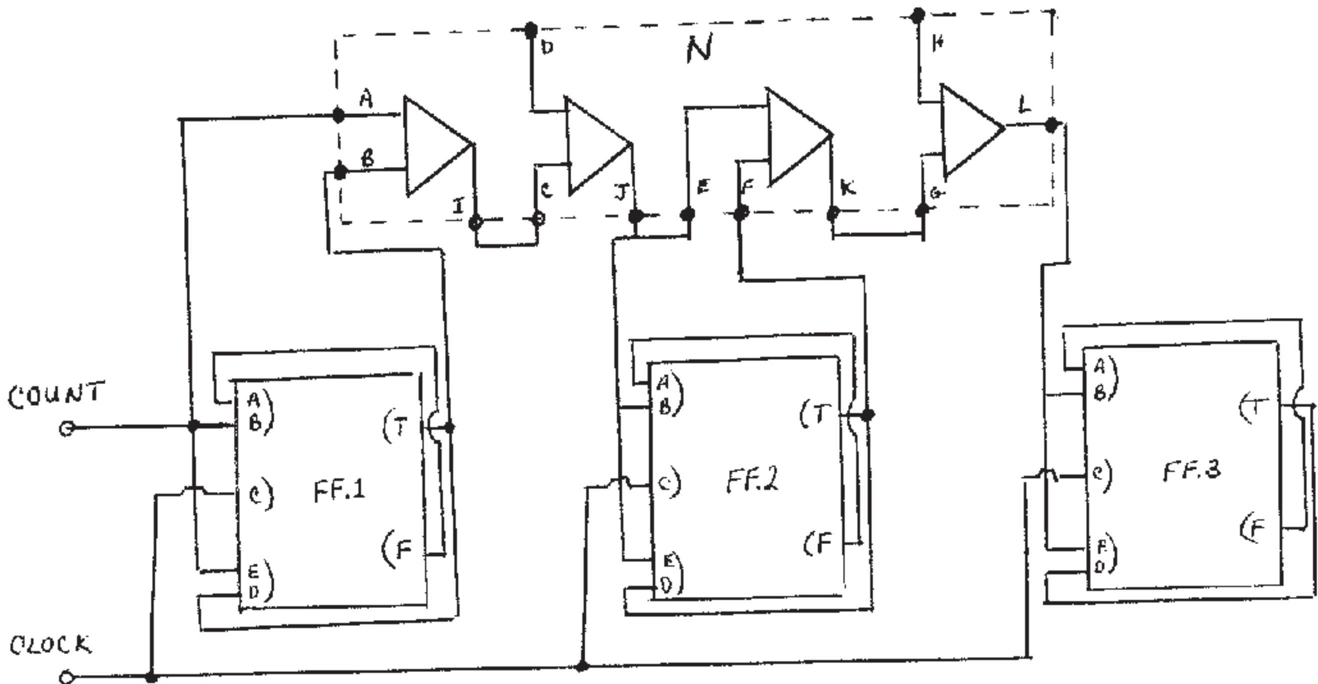


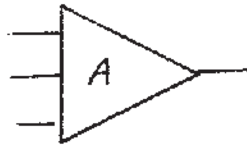
Figure 1.

```
FF.1  ← C(CLOCK)  -     Set 1 (FF.1:F) *   Set 2(COUNT),
                      Reset 1 (FF.1:T) * Reset 2(COUNT)

FF.2  ← C(CLOCK)  -     Set 1 (FF.2:T) *   Set 2(N:J),
                      Reset 1 (FF.2:T) * Reset 2(N:J)

FF.3  ← C(CLOCK)  -     Set 1 (FF.3:T) *   Set 2(N:L),
                      Reset 1 (FF.3:T) * Reset 2(N:L)
```

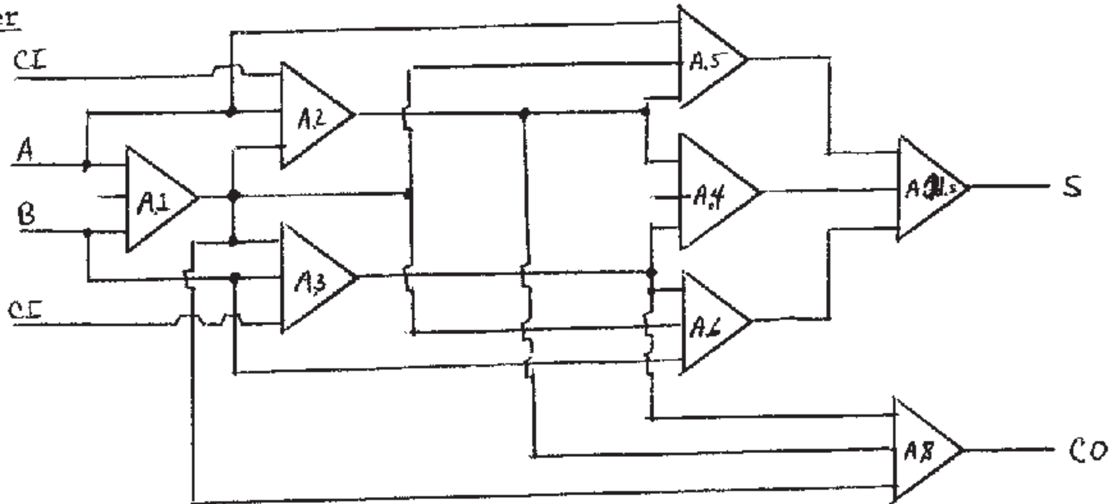
```
N  = A(COUNT) * B(FF.1:T),
       C(N.I),

     E(N.J) * F(FF.2LT),
       C(N.k)
```

Figure 2.

Nand Gate



Full Adder




```
Define   Add[A, B, S, CI, CO]

     A.1 =  (Add:A) * (Add:B); - (see footnote)
     A.2 =  (Add:A) * (Add:CI) * (A.1);
     A.3 =  (Add:B) * (Add:CI) * (A.1);
     A.4 =  (A.2) * (A.3);
     A.5 =  (Add:A) * (A.1) * (A.2);
     A.6 =  (Add:B) * (A.1) * (A.3);
     A.7 =  (A.4) * (A.5) * (A.5);
     A.8 =  (A.1) * (A.2) * (A.3);
   Add:S =  A.7;
  Add:CO =  A.8;
end definition;
```

If terminal designation is unspecified, any free input terminal may be assigned.

36-bit Adder

```
Define Adder [x, y, z];

    Add.1 = A(x[1]), B(y[1]), CI(zero);

    for k in 2, ..., 36

    Add.k = A(x[k]), B(y[k]), CI(Add.k-1:CO);

    for k in 1, ..., 36

    Adder:z[k] = Add.k:S;

end definition;
```