UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT

A Model for Register-Transfer Level Design Specification: The SDC Notation

Farhad Mavaddat

VLSI Group
CS-84-34

October, 1984

A Model for Register-Transfer Level Design Specification: The SDC Notation

Farhad MAVADDAT

CS-84-34
VLSI Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

A model for a register-transfer level design specification is proposed. Using this model, the data-path part is specified using three types of primitives, and the control part is simply a table associating the path-setting signals of the data-path with the control and status inputs.

The proposed model has an algebraic representation and its application to a number of design situations is discussed.

Keywords: register-transfer level, data-path, design representation, modeling.

A Model for Register-Transfer Level Design Specification: The SDC Notation

Farhad MAVADDAT

CS-84-34
VLSI Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

1. Introduction

Computer Aided Design (CAD) packages must operate on some form of design representation. These representations have a significant effect on the complexity of the CAD tools in the package. CAD packages should also provide their users with the means of specifying the design, the so-called design capture subsystem. The human friendly requirements of a design capture system, and the process friendly requirements of tools operating on the design, are often incompatible if not contradictory. This lack of compatibility between the user and tool requirements on the one hand, and the often incompatible representation requirements of the tools themselves on the other, has resulted in heterogeneous design environments, recognized as one of the major problems facing the CAD industry.

The availability of a single, multi-purpose intermediate representation to be used by a number of different application tools is useful to the development of an integrated CAD package. In this methodology, different design capture systems are viewed as the user interfaces to the intermediate representation, each presenting a view of the design which is friendly to the application.

Within such an environment, unique tools will be developed for different design tasks, all operating on the proposed notation. Different user interfaces are translated into this intermediate notation and use the variety of tools developed. This provides the designer with a number of choices in selecting the design capture system he may find more suitable to his style or requirements without needing to duplicate the tool development each time.

VLSI design is performed at several levels of abstraction, each with its own attractions and trade-offs [2,11,12,19]. Compared to conventional digital design, the register-transfer (RT) level abstraction of VLSI design has received more attention in recent years [1,3,4,7,8,9,13,14,18,20]. This enthusiasm can be attributed to the repetitive structure implied by the RT-based designs and the power of the RT-level abstraction in representing large systems in a concise and friendly form. Both these properties are of considerable importance in VLSI design.

While this attention to RT-level abstraction has already resulted in the

proposal of a number of RT-level design methodologies [1,9,14], less attention has been given to the study of the semantics of this abstraction [6]. This has resulted in a number of proposals [5,6,10]. We believe that while these studies propose some type of formal model for RT-level abstraction, their approaches are either too specific to certain applications to be of general use, or too abstract to be helpful to real design situations.

In this report we have taken on ourselves to propose a set of desirable features for such semantic models, to propose one such model, and to demonstrate that our model satisfies most of the proposed features.

This model is being used as the form of intermediate design representation in a large VLSI research program at Waterloo, supported by the Canadian Government.

2. OBJECTIVES

The development of a model for RT-based specifications which satisfies the requirements of multiple applications requires careful study. In our study of some of these application areas, we found the following properties to be highly desirable.

[1] **Computer representability.** Design tools are computer programs operating on design specifications represented in computer memory. To simplify the complexity of tool development, it is important to have these internal representations in forms which have proven suitable to computer processing. Tabular and matrix representations are one kind of these efficiently handled representations. Others are those with a suitably simple and regular structure, often defined by one of the formal languages. On the other hand, human oriented graphical representations are difficult to process. Textual definitions which do not adhere to the constraints of one of the formal languages are also hard to process.

[2] **Derivability.** This property requires that the specification of a new system, obtained through the combination (interconnection) of two subsystems, be easily derivable from the specifications of the initial subsystems. This property is very important in hierarchical design and helps with the definition of the higher levels without the need for re-definition at every level. Stated in a more formalized way we require that: S(a+b) be derivable from S(a) and S(b), where S(a+b) is the specification of combined modules "a" and "b", S(a) is the specification of "a", and S(b) is the specification of "b".

[3] **Non-restrictive.** A non-restrictive specification is one which does not restrict the applications of a design to only those foreseen by the designer. A specification becomes restrictive when it is defined in terms of a large number of high-level elements. To remove the restriction it must be specified in terms of a few primitive elements.

For example, the specification of a counter as a "counter" may prevent its also being used as a "register", but when specified as a combination of a "register" and an "incrementer" it is not subject to this restriction.

[4] **Usage Assistance.** Most design components of interest have one or more control inputs which in addition to controlling their function, provide the component with a certain amount of generality. For example, a shift register needs at least two control inputs for the control of "load" and "shift" operations. The "shift" input, in addition to controlling the component's shift function, expands its power to multi-shift applications. More complex components often have a larger number of control inputs, leading to a higher degree of programmability.

In an automated design environment, a synthesizing program trying to assemble a number of components needs to "understand" these inputs to create the function specified by the designer. This is possible only if the specification of those controls is able to fully describe their effect on the system.

It is not difficult to see that some behavior specification methods fail to

exhibit this property. To satisfy the requirement, the specification of a control input should describe its internal effect on the system, down to the level of primitive elements, rather than explaining the kind of change at the output due to the enabling of every control input. In a design specification model which satisfies this requirement, it is possible to deduce the behavior which results from the enabling the controls, while the opposite is not always possible.

[5] **Storage and retrievability.** Predesigning of frequently-used parts and their use in future designs is an integral part of any design activity. Poly-cell libraries are the VLSI implementation of this concept. When the size of these libraries increases, or new additions are made, the designer can easily lose track of what may be available. Automating the storage of newly designed components, and helping the designer in retrieving those potentially useful to his design is a desirable feature for any CAD system. Storing components in a way which does not restrict their retrieval to only those applications foreseen by the creator of the data-base is not an easy task.

This property requires that the device model help with its systematic placement in a library of predesigned cells, and that the model help the designer by providing him with a query system to retrieve the cells suitable for his design.

[6] **Circuit optimization assistance.** Optimization is a search within the space of functionally equivalent designs, subject to constraints imposed by the requirements of the design. While one can not expect the model itself to embody the optimization algorithm, its ease of use in presenting the algorithm with the design alternatives and in helping to transform one design to another equivalent design is indeed a property of the model.

This property requires that the model should facilitate the transformation of one design to other designs which perform the identical function in a shorter time or with less components. This is possible if the rules of legal transformation are easily representable in the same specification language.

In the next section of this paper we propose a new model for the RT-based specifications of design. This is then followed by a more formal presentation of RT-based modeling with a discussion of its power. The paper ends with an informal presentation of the application of the model in most of the areas of application discussed in this section.

3. MODEL DESCRIPTION

Our model is based on the assumption that the interfacing signals of every module of a design, i.e. those used to interface the module to other modules of the system or the environment, are one of three types;: *data*, *control*, or *representation*. We call a module's specification in terms of only two of three axes, its "projection" onto the plane formed by those two. In this way every design has three projections, each showing certain aspects of the design, more simply than the one comprising all three. Certain features of a design can be represented more clearly in terms of a particular projection. The RT-level specification is a design's projection onto the plane of *data*−*control*, ignoring the *representation* coordinate. On the other hand the distinction between two n-bit register ICs, one with an individual "load" control for every flipflop, and the other with a single "load" for all the flipflops, is only shown by its projection onto the "control-representation" plane.

Through this abstraction we have followed the path of other engineering disciplines who have found it advantageous to specify their designs by projecting then onto the planes of an orthogonal coordinate system. In digital design also, the independence of any two types of the interface signals from the third leads to the suggestion of some abstract form of orthogonality, and therefore the three-dimensional space of the digital design. This suggestion of three-dimensionality is enforced by certain drawings of the design. In Figure 3-1 we have shown one such drawing.

We are using the projections of a representation as a vehicle for specifying designs in our design specification input language. But for this report, this is the extent to which the three-dimensional design will be discussed.

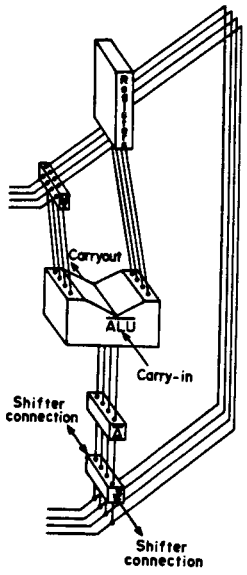


FIG. 3-1.

3.1. Model Primitives

We start by an informal discussion of the data-path and give a few examples of the data-paths of some well-known modules. We will have an even less formal presentation of the control function. The intention of this section is to introduce the reader to the proposed RT-level modeling techniques in an informal way and to demonstrate the motivations behind the more formal treatment to come in the next section.

A data-path is a network of three types of elements, each with one or more inputs and a single output, and interconnected according to certain rules to be defined in the next section. We refer to these elements as the *primitives*.

A *net* is a star connection of one or more "incoming" and "outgoing" edges connecting the input and the outputs of these *primitives* forming the

network. To every *net* we assign a value of a certain "type". The "types" of *net* values will depend on the *representation* techniques employed by the system. The "value" of a *net*, defined by the values of "incoming" edges, is passed to the succeeding primitives through the "outgoing" edges. Contradictory "incoming" edge values lead to undefined *net* values. The value of a *net* is referenced by the usual naming conventions. Following are the descriptions of the three *primitives*.

[1] *Selectors* do not have any interfacing signals along the *representation* axis and therefore are fully representable by their projection onto the *data*−*control* plane. A selector's projection onto the *data*−*control* plane has two inputs and one output along the *data* axis, and a single input along the *control* axis, Figure 3-2. The value of the data-output, at any time, is equal to the value of the data-input selected by the control-input at that time. Every control input has a "normal" setting which represents its choice of input under the control input's "disabled" condition. "Enabling" a *selector's* control-input forces its data-output to the other data-input value, for the duration of the "enabling" signal. The two inputs and the output of any selector are of the same type.

A *selector's* function can be shown algebraically by:

$$out = in1 \mid_j in2$$

where *out* is the value of the outgoing edge, $in1$ and $in2$ are the values of the incoming edges, and $\mid_j$ is the control-input, selecting either $in1$ or $in2$ for the value of *out*. The enabling signals are issued by the control unit. In this paper the normally-left convention is assumed; that is, when $\mid_j$ is enabled, then the right hand expression is selected.

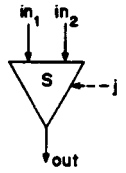


FIG. 3-2.

[2] *Combinationals* do not have any interfacing signals along the *control* axis and are therefore fully representable by their projections onto the *data*−*representation* plane. A *combinational's* projection onto the *data*−*representation* plane has one or more inputs and a single output along the *data* axis, and an equal number of inputs and outputs along the *representation* axis, Figure 3-3. The value of the *data* output, at any time, is a function of the *data* and *representation* inputs at that time. Similarly, the values of the *representation* outputs at the same time are a function of the *data* and

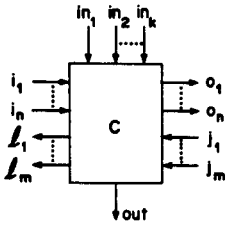


FIG. 3-3.

representation inputs. The signal carrying the value of "carry", from one slice of the "adder" to the next, is an example of input and output signals along the *representation* axis.

The value of *data* outputs as a function of *data* inputs can be shown algebraically, using the conventional function definition notations: $out = f(in_1, in_2, ...in_k)$, or can be listed in tables. We use a circle, with one or more inputs and a single output, to show the projection of a *combinational* element onto the *data*−*control* plane.

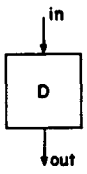


FIG. 3-4.

[3] *Delays* do not have any interfacing signals along the *control* and *representation* axes and have a single input and a single output along the *data* axis, Figure 3-4. In this sense they are one-dimensional elements. A *delay* element's output lags its input by one time unit.

A *delay* element's function can be shown algebraically by:

$$out = \Delta(in)$$

where *out* is the value of the outgoing edge, and *in* is the value of the *incoming* edge.

4. SPECIFICATION EXAMPLES

Here we present a few examples of module specifications using the proposed *primitives*. Treatment of these examples, especially where it deals with the definition of the control parts, will be informal. We present a more formal discussion of Register-Transfer type specifications in the next section.

4.1. Registers

We start by discussing the specification of a few kinds of registers. We will see later that by combining these registers with suitable multiplexors and de-multiplexors, different types of memories can be specified.

The simplest kind of register is the read-only register, to be known as an *ro−register*, specifying constant values. The data-path of one such register is a single *delay* element fed back on itself, shown graphically in Figure 4-1. This register is also shown by its algebraic form:

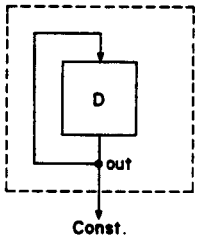


FIG. 4-1

$$out = \Delta(out)$$
$$out_{init} = Const.$$

where we assume that the *init* subscript refers to the placement of some initial constant value in the refreshing loop, at the time of manufacture.

The most common form of register is the read-write register, referred to as an *rw−register*. The projection of the data-path of an *rw−register* onto the *data−control* plane is shown grphically in Figure 4-2. Because neither the *selector* nor the *delay* element has any components along the *representation* path, this projection contains all the information necessary for the specification of such registers. The algebraic specification of *rw−registers* is shown by the following set of simultaneous equations.

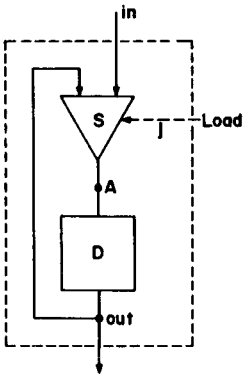


FIG. 4-2.

$$A = out \mid_{j} in$$
$$out = \Delta(A)$$

Under "normal" conditions the registered value is circulated in the hold loop from one

clock pulse to the other. This helps the register to refresh the value last placed in it. Enabling the j control-input places a new value in the *rw−register*. This leads to the equivalence of the "load" input to the register and the j control-input of the selector. We write this in the form of a table, associating the module-interfacing control-lines (here the "load" signal), with the control-inputs of the *selectors* in the data-path. For the *rw−register* this is simply shown as follows:

	j
load	Y

This table shows that for every "enabling" of the "load" signal, the j control-input of the data-path will also be "enabled".

Another kind of register to be specified is the programmable read-only register, to be called the "pro-register". The projection of a "pro-register" onto the *data−control* plane is shown graphically in Figure 4-3. The algebraic specification of its data-path is as follows:

$$A = out \mid_j B$$
$$B = f(in, out)$$
$$out = \Delta(A)$$

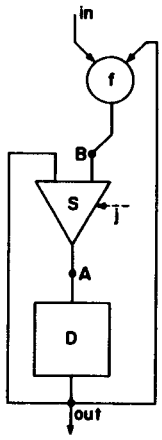


FIG. 4-3.

A "pro-register" is an "ro-register" in which the placement of the initial value in the refresh loop is under user control. "pro-registers" are manufactured with a certain value initialized in the refreshing loop. The user is able to modify that value, to his liking, through the proper application of the input values. The control table of a "pro-register" is similar to that of the "rw-register". This specification of a "pro-register" captures the reprogrammability of the programmable read-only memories, which is sometimes overlooked in discussion of this device. Use of f

combinatorial prevents full specification of "pro-registers" by their *data−control* projection. Howeever, further specification are beyond the scope of this example.

The last kind of register to be discussed is an erasable-programmable-read-only register, or an "epro-register". "pro-registers" lose their reprogrammability feature soon after one or two iterations. An "epro-register" enables the user to recreate the initial condition of programmability, by placing the initial value back in the register. The *data−control* projection of the data-path of an "epro-register" is shown in Figure 4-4. The existence of two *selectors* in its data-path makes its control-table more interesting. The following table relates the "enabling" of the i and j "control-signals" to the "enabling" of the *erase* and *program* control-lines.

	i	j
erase	E	X
program	D	E

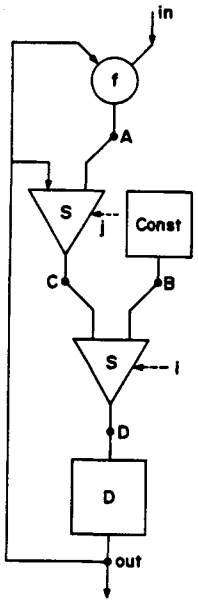


FIG. 4-4.

The following set of equations show the algebraic specification of the data-path.

$$A = f(out, in)$$
$$B_{init} = Const.$$
$$B = \Delta(B)$$
$$C = out \mid_j A$$
$$D = C \mid_i B$$
$$out = \Delta(D)$$

4.2. Multiplexors and De-multiplexors

A *selector* is a simple 2-input multiplexor. Larger multiplexors are made by forming an inverted tree of *selectors* such that the *selectors* of the same depth share the same control-input. Multiplexors are fully specifiable

by their projection onto the *data−control* plane and therefore their expansion does not require any connections along the *representation* axis. In Figure 4-5 the expansion of two n-deep (2^n way) multiplexors into a single (n+1)-deep multiplexor is shown.

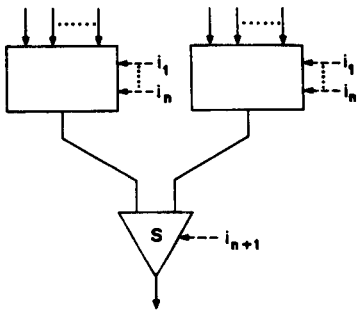


FIG. 4-5.

To construct a multi-way de-multiplexor, we must first construct a simple two-way de-multiplexor. Figure 4-6 shows the *data−control* projection of one such two-way de-multiplexor using two *selectors*. Following is the algebraic specification of the data-path of the two-way de-multiplexor:

$$out_1 = in_1 \mid_i in$$
$$out_2 = in \mid_i in_2$$

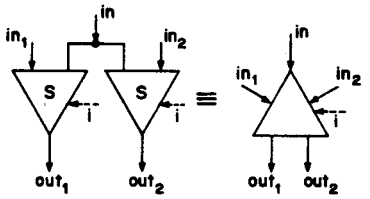
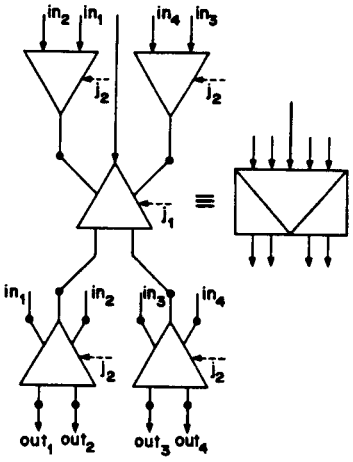


FIG. 4-6.

where the *in* signal is the main input to be multiplexed, and in_1 and in_2 are the alternative inputs.

An alternative input specifies the value of the corresponding output signals when it is not selected to receive the *in* signal. Two-way de-multiplexors can be expanded to multi-way de-multiplexors using the same tree expansion technique used for the multiplexors. This form of expansion leads to too many alternative inputs which are a nuisance in most applications. A more useful expansion of de-multiplexors is the one which uses a number of *selectors* to reduce the number of alternative inputs. We refer to these reduced input de-multiplexors as the homogeneous de-multiplexors. A homogeneous four-way de-multiplexor is shown in Figure 4-7. The following set of equations describe the values of the four output signals in terms of the four alternative and the main input signals.



These equations describe the data-path operations of the de-multiplexor. In these equations, the intermediate *net* values have been replaced in the expressions and only the input-output relationships are shown.

The following table shows the value of the four outputs under the different "enabling" conditions.

$$out_1 = in_1 \mid_{j_2} ((in_2 \mid_{j_2} in_1) \mid_{j_1} in)$$
$$out_2 = ((in_2 \mid_{j_2} in_1) \mid_{j_1} in) \mid_{j_2} in_2$$
$$out_3 = in_3 \mid_{j_2} (in \mid_{j_1} (in_4 \mid_{j_2} in_3))$$
$$out_4 = (in \mid_{j_1} (in_4 \mid_{j_2} in_3)) \mid_{j_2} in_4$$

j_1	j_2	out_1	out_2	out_3	out_4
D	D	in_1	in_2	in_3	in
D	E	in_1	in_2	in	in_4
E	D	in_1	in	in_3	in_4
E	E	in	in_2	in_3	in_4

The expansion of two n-deep de-multiplexor trees into a single $(n+1)$-deep de-multiplexor is shown in Figure 4-8.

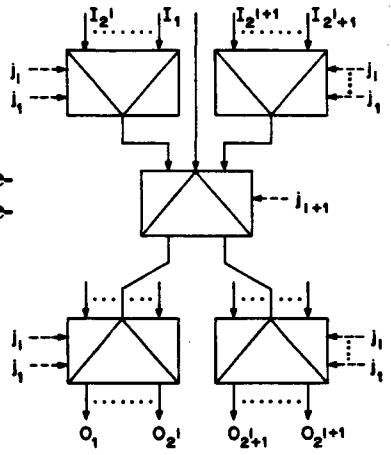


FIG. 4-8.

The imaged interconnection of two n-deep de-multiplexors and multiplexors, receiving the same control signals, forms a path-selector for routing an incoming signal into one of 2^n paths and collecting the result back into a single outgoing signal. By placing other elements in different paths of such selectors, certain useful subsystems can be specified. One useful application of path-selectors is in the specification of memory systems. By placing any of the different kinds of registers discussed in the paths of the path-selectors, different types of memory systems can be specified. In Figure 4-9 we have shown the specification of a simple two-word read-write memory. Several points are worth mentioning about this memory:

1- First, its functional operation is fully specified by the following set of data-path equations and the corresponding control table. Such detailed and realistic capture of a memory sub-system function, using the operations of very simple primitives (in fact for memory function definitions

only *selectors* and *delay* elements are used) is to our knowledge new.

$$out = A \mid_j B$$
$$A = \Delta(C)$$
$$B = \Delta(D)$$
$$C = A \mid_i E$$
$$D = B \mid_i F$$
$$E = A \mid_j in$$
$$F = in \mid_j B$$

	i	j
read 1	D	D
read 2	D	E
write 1	E	D
write 2	E	E

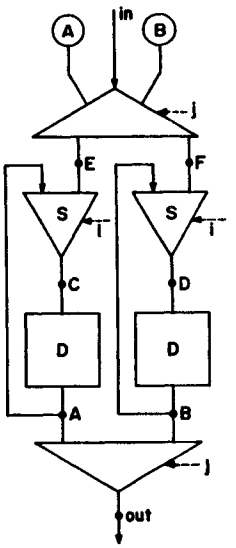


FIG. 4-9.

2- In practice the existence of a separate control line for reading and writing of every location is unrealistic, and the user must also specify the coded address of the addressed location along with the specification of the "read" or "write" operations. This converts our scheme to that of current practice.

3- The command to load a value into a register and the path-selection address (in a path-selector mechanism) now have the same semantic interpretation. They are both the settings of some *selector's* control-input in the data path.

4.3. Three State and Bus operations

Traditionally, one of the problems facing most specification models has been in the area of specifying the Bus and three-State operations of digital systems. Here we represent a floating output, the so-called third-stated, by a *selector* in which the selector's output has been connected to one of its inputs, usually the one normally selected. The output of such a *selector* will confirm the value placed by the other outputs connected to the same *net* under normal conditions. On the other hand, when its control-input is "enabled", its output will be that of its other input. In Figure 4-10 we have shown the data-path of three *rw−registers* placed on a single Bus structure.

$$A = D \mid_{i_1} BUS$$

$$B = E \mid_{i_2} BUS$$

$$C = F \mid_{i_3} BUS$$

$$BUS = BUS \mid_{j_1} D$$

$$BUS = BUS \mid_{j_2} E$$

$$BUS = BUS \mid_{j_3} F$$

$$D = \Delta(A)$$

$$E = \Delta(B)$$

$$F = \Delta(C)$$

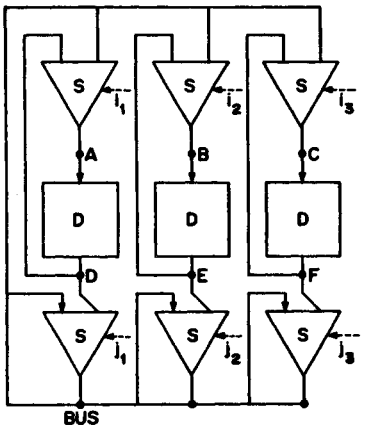


FIG. 4-10.

5. FORMAL DEFINITION

A register-transfer-level model of a module consists of a data-path and a control-unit. The data-path has a set of data-inputs, data-outputs, path-selector-inputs, and status-outputs. The data-inputs and data-outputs communicate with the environment and/or the other modules of the design. The path-selector-inputs and the status-outputs communicate with the control-unit.

The control-unit has a set of control-lines, status-inputs, and action-outputs. The path-selector-inputs and status-outputs of the data-path are connected to the action-outputs and status-inputs of the control-unit. The schematic representation of a module is shown in Figure 5-1.

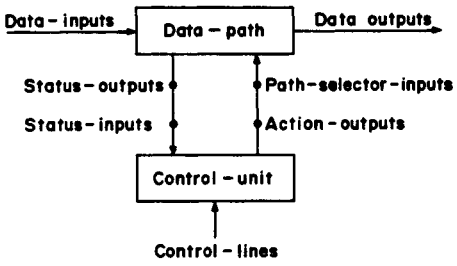


FIG. 5-1.

5.1. The Data-Path

A data-path is a bipartite directed graph $G = (V,E)$ with partitioned subsets X and Y, such that for all $v \in V$, $v \in X$ if v is a *primitive*, and $v \in Y$ if it is a *net*. Only the *data* terminals of a *selector* participate in the formation of the data-path.

We assign a value to every *net* equal to the value of its incoming edges, which should not contradict each other. The *net* value is passed to its adjacent nodes (which must be *primitives*) through its outgoing edges. The set of all control-inputs of the *selector* type vertices, Q, forms the path-selector-inputs of the data-path. If Q is empty, i.e. no *selectors* are present in the data-path, then the data-path has a fixed structure with nothing to control.

A *port* is a *net* with one incoming or one outgoing edge which enters or leaves the data-path. Some output *ports* with logical values are called *probes* and are used to feed back the data-path status to the control-unit. The set of all *probes* in the data-path, R, forms its status-outputs. If R is empty, i.e. the data-path is not *probed*, the action of the control-unit will be independent of the status of the data-path. The graph representation of the *epro−register* data-path, discussed earlier, is shown in Figure 5-2. This example has one data-input, one data-output, two path-selector-inputs, and zero status-outputs. Obviously the action of control on an *epro−register* is independent of its status.

5.1.1. Data-Path Characteristic Equations

We write one equation for every incoming edge of every *net*, relating the value of the *net* to the values of the *nets* feeding the preceding *primitives*. The bipartite property of the data-path graph guarantees that only three kinds of equations, corresponding to the three types of the *primitives*, need to be written.

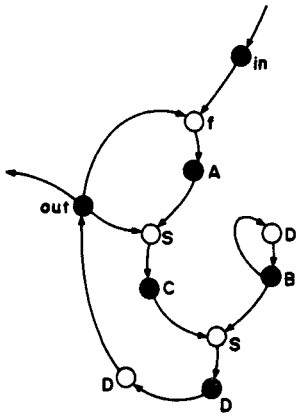


FIG. 5-2.

For every edge coming from a *delay* element in the data-path we write:

$$net_i = \Delta(net_j)$$

where net_i and net_j are the reference names of the input and output *nets*.

For every edge coming from a *combinational* element in the data-path we write:

$$net_i = f(net_{j+1}, net_{j+2}, \ldots, net_{j+n})$$

where net_i is the reference name of the output *net*, net_{j+k} for $1<k<n$ are the reference names of the n input *nets*, and f is a function describing the operation of the *combinational*.

Finally, for every edge coming from a *selector* in the data-path we write:

$$net_i = net_j \,|_l\, net_k$$

where net_i is the reference name of the output *net*, net_j and net_k are the reference names of the input *nets*, and $|_l$ is the selection operator, controlled by the l control-input, assigning either the value of net_j or net_k to net_i.

When the incoming (or outgoing) edge of a *net* is connected to the environment, we write:

$$net_i = in_j$$

(or)

$$out_j = net_i$$

where net_i is the reference name of the corresponding *net*, and in_j (out_j) is the value of the *jth* input (output) stream to the data-path.

The set of simultaneous equations written for every incoming edge of every *net* in the data-path are the characteristic equations of that data-path. The number of such equations, N, is shown by the following equation:

$$N = \sum_{v \in Y} d_i(v)$$

where Y is the partitioned subset containing the *nets*, and $d_i(v)$ is the indegree

of the *ith* v.

5.2. The Control Unit

The control-unit is a device with three sets of terminals: the set of control-lines, C, the set of status-inputs, S, and the non-empty set of action-lines, A. The control-lines, if present, are under the control of other modules or the environment. There is a one-to-one correspondence between the elements of Q and R, and those of A and S respectively.

We assume that the truth values of the *probes* are assigned to the corresponding status-inputs of the control-unit through the R to S correspondence, while the control-unit's actions affect the data-path's path-selector-inputs through the A to Q correspondence.

We are proposing a synchronous model with a global clock, **c**, oscillating between the T and F values with a period equal to the delay value associated with the *delay* primitives. All system activities are synchronized with the occurence of the T period of the global clock and should settle to a stable condition within that clock period.

A control-unit's action depends on the current *enabling* state of the control-lines and the state of its status-inputs. For simplicity we will assume that only one control-line is *enabled* at a time and later demonstrate that this restriction can be relaxed.

We associate the unique tables **C** and **A** with every control-line, assuming a one-to-one correspondence between the columns of **C** and $L \in 2^S$, and the columns of **A** and A, such that $c_{ij} \in \{T,F,X\}$ and $a_{ik} \in \{E,D,X\}$ for $1 \leq i \leq m$, $0 \leq j \leq |L|$, and $1 \leq k \leq |A|$, where m is the number of rows in **A** and **C**. According to this definition an empty L may have a maximum table entry size of one. We will assume that such a table always has a single T entry, known as the "unconditional" table.

We define the *ith* row of **C** to be satisfied *iff*, the corresponding control-line is enabled, the value of the global clock is T, and the truth values of the elements in L match the corresponding table entries of the *ith* line of **C**. The X entries in the table match both the T and F values of the corresponding element in L. The *ith* row of **A** is "activated" *iff* the *ith* row of **C** is "satisfied". Certain *probe* conditions may "satisfy" more than one row of the table, which is used to execute parallel activities within the module.

When the *ith* row of **A** is "activated", the corresponding elements of A are *enabled* or *disabled* for the respective E or D table entries, which in turn affects control-input settings in the data-path. The action of X type activations on the *selector* setting may depend on the technology. When more than one row is "activated", "strong" entries (D and E) in the same column should not contradict each other; the X entries may be overruled by stronger E and D entries.

This definition is "complete" if every *true* period of the global clock **c** is accompanied by the "enabling" of at least one of the control-lines. If this is not guaranteed then the state of all *selectors'* control-inputs should be specified for the periods of "inactivity". We assume the existence of an "implicit" control-

line which is "enabled" whenever all "explicit" control-lines remain "disabled" during the *true* period of the global clock. The "implicit" control-line is treated, and must act, like all other "explicit" control-lines.

5.2.1. Extending the Control Unit

The table-based specification of the control-unit is a simple and powerful scheme for designing familiar types of control, and is related to horizontal microprograms. In the rest of this section we will look at a sequence of conventions which progressively simplify the task of design specification and improve the expressive power of the model.

The first convention combines the tables associated with each control-line, by side-concatenating them. This is always possible, because the table pair associated with every control-line have an equal number of rows. The table entries are the same as those in the original tables. The following is an example of this convention:

c	s_1	s_2	...	s_n	a_1	a_2	...	a_p
	c_{11}	...	...	c_{1n}	a_{11}	...	...	a_{1p}
	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...
	c_{m1}	...	...	c_{mn}	a_{m1}	...	...	a_{mp}

The second convention combines the control-tables of all (or several) control-lines into a single table. This is achieved by first extending the tables into equal width and then concatenating them columnwise.

Tables are extended to the same width by adding columns to their condition parts. The extended condition parts must have identical columns which are the union of all columns of the individual tables prior to this extension. Every entry in the extended table which corresponds to a column not present in the original tables, receives an X value. To specify the section corresponding to each control-line, in the concatenated form, the control-line label is placed next to the corresponding rows of the table. This convention assumes that by "enabling" each control-line, the corresponding rows of the table are the only candidates for being "satisfied".

The third convention removes the labeling of the control-table rows by treating the control-lines like those of the status-inputs. This is possible by adding one extra condition column for every set of identical control-line labels to be removed. Suppose that the rows i to $i+k$ of the concatenated table have the same row labels and we have added the j*th* column for the removal of these labels. The new table, to be called the "unified" table, receives an F entry everywhere in the j*th* column, except at positions: $c_{i,j}, c_{i+1,j}, \ldots, c_{i+k,j}$ which receive the T value. The logic controlling the "satisfaction" of the model remains unchanged.

Under this convention there is no need to allocate a separate column to the specification of the "implicit" control-line, and extra logic to detect the

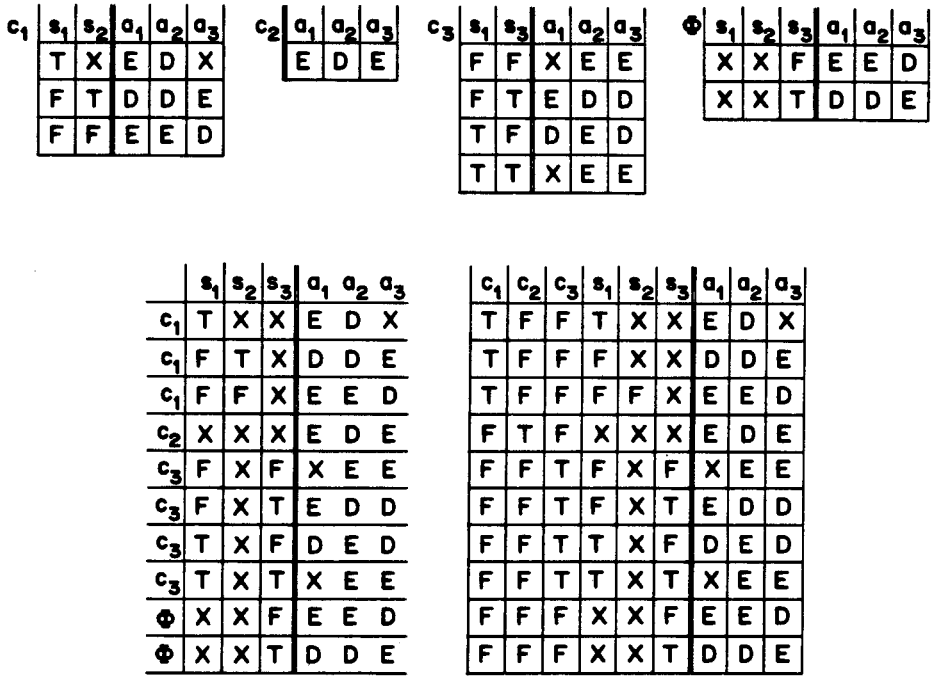
occurence of the"implicit" control-line condition. This is looked after by entering *F* values in every position associated with the "explicit" control-line columns and the "implicit" rows.

The concept of a "unified" control-table is also used to specify the action of the encoded control-lines. Under the control-line encoding convention more than one control-line may be "enabled" at any time and the pattern of the "enablings" signifies the action to be taken by the data-path.

To specify the action of an encoded set of control-lines, one row is assigned to every unique encoding pattern, with entries to guarantee the "satisfaction" of that row *iff* the correct pattern of the control-line "enablings" occurs. It is not difficult to see that the original scheme of "enabling" one control-line at a time is a special case of this more generalized scheme.

The last convention to be discussed deals with the specification of the control hierarchies. In the next section, while discussing the application of the proposed model to the specification of a hierarchical example, we will present the necessary conventions.

The proposed specification improvements enable us to efficiently merge the control-tables, associated with different control-lines, into an integrated control scheme, unifying all table pairs into a single table. One example of this sequence of conventions, starting with three simple control-line specifications, and progressively combining them into the "unified" table representation is shown in Figure 5-3.



c_1	s_1	s_2	a_1	a_2	a_3
	T	X	E	D	X
	F	T	D	D	E
	F	F	E	E	D

c_2	a_1	a_2	a_3
	E	D	E

c_3	s_1	s_3	a_1	a_2	a_3
	F	F	X	E	E
	F	T	E	D	D
	T	F	D	E	D
	T	T	X	E	E

Φ	s_1	s_2	s_3	a_1	a_2	a_3
	X	X	F	E	E	D
	X	X	T	D	D	E

	s_1	s_2	s_3	a_1	a_2	a_3
c_1	T	X	X	E	D	X
c_1	F	T	X	D	D	E
c_1	F	F	X	E	E	D
c_2	X	X	X	E	D	E
c_3	F	X	F	X	E	E
c_3	F	X	T	E	D	D
c_3	T	X	F	D	E	D
c_3	T	X	T	X	E	E
Φ	X	X	F	E	E	D
Φ	X	X	T	D	D	E

c_1	c_2	c_3	s_1	s_2	s_3	a_1	a_2	a_3
T	F	F	T	X	X	E	D	X
T	F	F	F	X	X	D	D	E
T	F	F	F	F	X	E	E	D
F	T	F	X	X	X	E	D	E
F	F	T	F	X	F	X	E	E
F	F	T	F	X	T	E	D	D
F	F	T	T	X	F	D	E	D
F	F	T	T	X	T	X	E	E
F	F	F	X	X	F	E	E	D
F	F	F	X	X	T	D	D	E

FIG. 5-3.

5.2.2. Example

In the following example, pairs of consecutive numbers are read from a single infinite-input sequence and outputted in sorted pairs. Input and output values are advanced by a global clock pulse and value pairs are separated by an extra clock used for the calculation period. A typical input sequence will therefore look like: $i_1 i_2 \# i_3 i_4 \# i_5 i_6 \# \ldots\ldots$, where # represents the pair separator as well as the space holder for the third period of the clock activity.

The graphical specification of the module's data-path is shown in Figure 5-4. In order to illustrate the technique, we have chosen a rather simple design, ignoring some important considerations required in real designs.

Part of the data-path is a two-bit counter which loops through the sequence of values FF, FT, and TF, never generating the TT output. As a result we have omitted this possibility from the control-table. The *f* and *g* *combinationals* used in the design of the counter have no signals along the *representation* axis and therefore are fully specifiable by their *data* input-output relationship shown below:

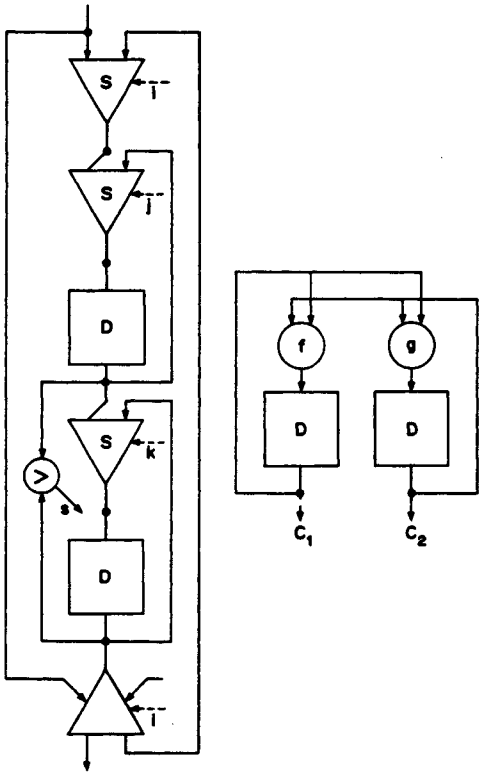


FIG. 5-4.

c_1	c_2	f	g
F	F	F	T
F	T	T	F
T	F	F	F

The following table shows the function of the control unit. The first two entries, "satisfied" during the first and second cycles of the counter operation, read the consecutive pair values from the input stream. The "activations" of the third and the fourth entries are mutually exclusive, and depend on the relative size of the last two values read, leading to the exchange of the values or their holding during the third counter value.

clock	c_1	c_2	s	i	j	k
	F	F	X	D	D	D
	F	T	X	D	D	D
	T	F	T	E	D	D
	T	F	F	E	E	E

This design has no control-lines and the only external signal to "activate" the table is the clock pulse. Therefore, we have associated this table with the enabling of the clock pulse.

6. EXAMPLES OF APPLICATIONS

We now discuss the degree to which our proposal satisfies the objectives set earlier in this paper. These discussions are informal and it is not our intention to make any claim beyond the intuitive level. We will show that the proposed notation has been developed with attention to the objectives and has the potential of application to many of the areas of intended application.

It is easy to see, without the need for supporting evidence, that the model meets the requirements of some of the objectives. For example, techniques of using tables to show the flow of control, and algebraic equations to show the flow of data, have efficient internal representations and are easy to process. This helps to ease the development of simulation packages to support the model.

It is also not difficult to see that the use of only three types of primitive elements, each with simple semantics, will help the cause of non-restrictive specification.

On the other hand, whether the model meets the expectations of other areas of application is not always easy to see and needs some explanation. In the remainder of this section, we will try to demonstrate, through examples, that the proposed model has the potential of meeting the objectives of a number of other applications.

6.1. Application to Derivability

We consider a limited class of synthesis activities, where the data-paths of the modules, interconnected at their inputs and the outputs, form a larger data-path, and the control function is defined through a hierarchy of definitions. This form of module synthesis is not uncommon and covers a reasonable number of design activities. In Figure 6-1 we have shown a schematic representation of the synthesis of two modules into a new module in this simple form.

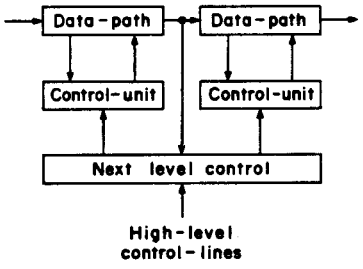


FIG. 6-1.

Other expansion methods, which mix the functions of control and data are also possible. One example of this generalized form of synthesis is the design of programmable architectures. For example, in a microprocessor, which is probably a good example of mixed synthesis, the data-path of its control part is used as the control-unit of its data-path.

We have chosen the design of a hardware sorter as our example of hierarchical design. The hardware sorter accepts n numbers, and outputs them in sorted form, under the control of a number of control-lines.

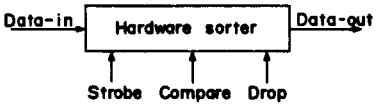


FIG. 6-2-a.

This design will output the sorted list immediately after the entry of the last input value. Such automatons are known to operate in real time. Our design reads the unsorted numbers, one at a time, at its input and outputs the sorted list at its output. Three control-lines control the operations of the sorter unit, Figure 6-2-a. The *strobe* control-line strobes (reads) the next number into the unit. This should be followed by the enabling of the *compare* control-line, which compares the last input value with the smallest value currently in the unit and presents the smallest new value at the unit's output. To create room for the entry of the next number, the *drop* control-line is enabled, dropping the current smallest value from the unit. The repetitive enabling of the *strobe*, *compare*, and *drop* control-lines, in a loop form, sorts any list of n values, where n is the internal capacity of the sorting unit. To compensate for the initial values in the sort unit and the values strobed after completion of the last entry, each list must be appended by a sufficient number of very small and very large values at both ends.

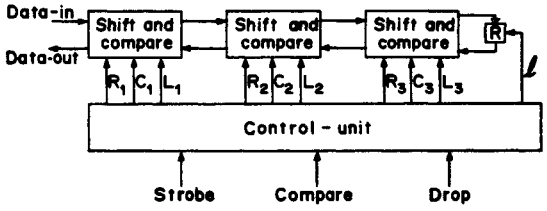


FIG. 6-2-b.

We realize the sort unit using a linear array of "shift and compare" units and a single register, all connected as in Figure 6-2-b. Input numbers are presented to the left "shift and compare" unit and the sorted numbers are returned by the same unit. The function of the top level control-unit, in which the input control-lines to the "sort-unit" are related to the control-lines of the "shift and compare" and the "register" units, is shown below.

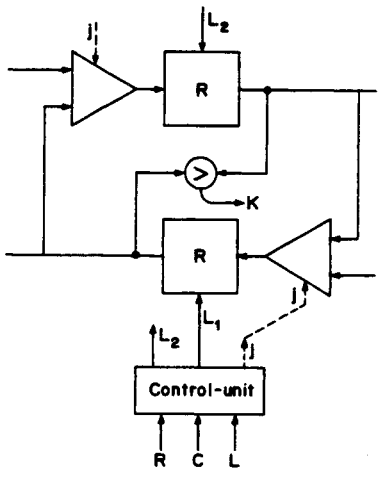


FIG. 6-2-c.

	R_1	C_1	L_1	R_2	C_2	L_2	R_3	C_3	L_3	ℓ
strobe	E	D	D	E	D	D	E	D	D	E
compare	D	E	D	D	E	D	D	E	D	D
drop	D	D	E	D	D	E	D	D	E	D

Next in the hierarchy we have to realize the "shift and compare" units. The internal design of one such unit is shown in Figure 6-2-c. The data-path is made of two "registers", two *selectors*, and a "comparator". The two registers hold two numbers of the "sort-unit". The L and R control-lines "load" the numbers present at the left and the right inputs of the "shift and compare" unit into the *registers*. The C control-line compares the two values and exchanges their position if the the one on top is the smaller of the two. This leaves the smallest value in the lower register after every enabling of the C control-line. It is easy to show, by inductive methods, that when these elements are configured in an array, and closed at the right by a register, the result of applying a continuous sequence of "strobe ", "compare", and "drop" control-lines is that the value in every lower register is always the smallest of all those to its right. The functional specification of the control-unit, relating the control-line inputs of the "shift and compare" unit to the those controlling the "register" and *selector* operations, is shown below.

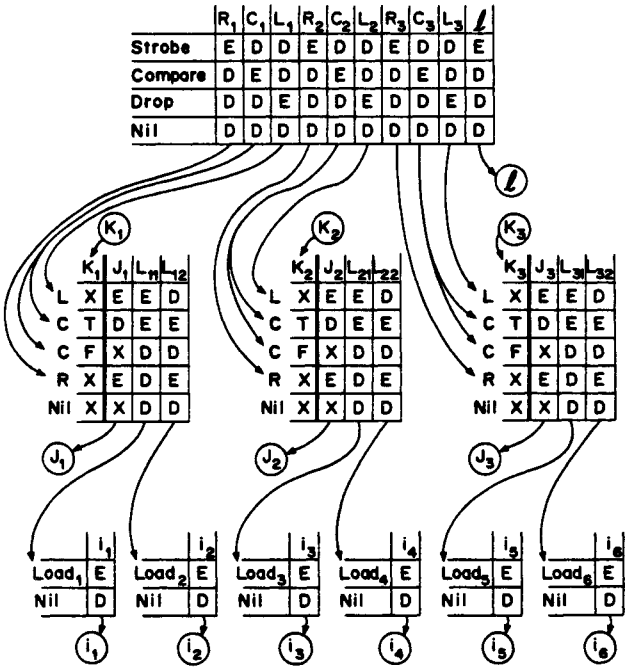


FIG. 6-3.

	k	j	L_1	L_2
L	X	E	E	D
C	T	D	E	E
C	F	X	D	D
R	X	E	D	E
Nil	X	X	D	D

The lowest level in the hierarchy, which relates all the units to the primitives of the proposed model, is the design of the "registers". This was included in the discussion of *rw−register*, in section 4-1. In Figure 6-3 we have shown a graphical representation of the total hierarchy of the "sort-unit" control functions.

6.2. Application to Mechanical Proof techniques

The ability to prove assertions about a design is a powerful technique in the development of a number of design tools, the most obvious being the development of the design verification tools. Less obvious of these applications is its use in "helping with usage", and "component retrieval". Instead of presenting a formal framework for the development of mechanical proof activities, which we are currently considering as a separate activity, we present here an informal scheme of reasoning about an example which demonstrates the suitability of the model to mechanical deduction.

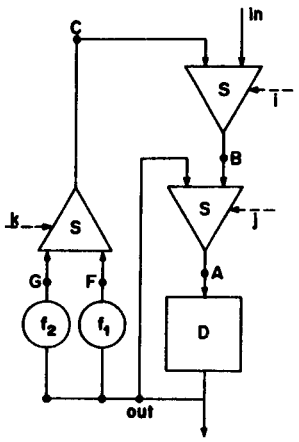


FIG. 6-4.

Let us consider the data-path of Figure 6-4 and find out whether it can compute the $f_1(f_2(f_1(x)))$ expression through successive enabling of some sequence of the control-lines. We are also interested in finding the sequence, should one exist. Following are the set of equations describing the data-path and the table specifying the effect of the control-lines on the data-path.

To search for the answer we have to go through a number of steps which lead to the sequence, should one exist.

1- First we cut the network at all delay elements. This provides us with a number of sub-graphs with new input and output

$$A = out \mid_j B$$
$$B = C \mid_i in$$
$$C = F \mid_k G$$
$$F = f_1(out)$$
$$G = f_2(out)$$
$$out = \Delta(A)$$

	i	j	k
LOAD	E	E	X
F1	D	E	D
F2	D	E	E
Nil	X	D	X

points. Figure 6-5 shows the data-path of Figure 6-4 after the cutting at the delay elements.

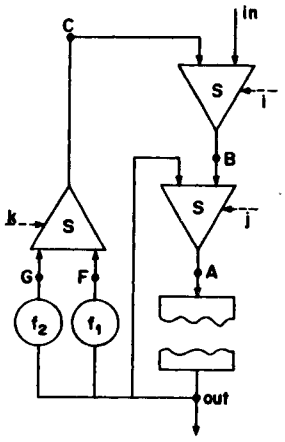


FIG. 6-5.

2- We write an expression for every output point in terms of the main and the new input points. This relates the main output net and the delay element input net values to the main input net and the delay element output net values.

$$A = out \mid_j B$$
$$B = C \mid_i in$$
$$C = f_1(out) \mid_k f_2(out)$$

3- We re-write the expressions obtained at step 2 in a form describing the output point values for every "enabling" of the control-lines.

$$\begin{array}{rl} LOAD & A = in \\ F_1 & A = f_1(out) \\ F_2 & A = f_2(out) \\ Nil & out = \Delta(A) \end{array}$$

4- In this step we apply substitution techniques by matching the expression to be computed with the equations derived in step 3. The substitution of

any equation will imply the enabling of the corresponding control signal. If and when the sequence terminates, the reverse enumeration of the enabled control-lines is the sequence of steps to be executed for the computation of the corresponding expression. Following is the list of substitutions and their corresponding control signals.

$f_1(out) = f_1(f_2(f_1(x)))$
implies the need for
$out = f_2(f_1(x))$ prior to the enabling of F1.

$f_2(out) = f_2(f_1(x))$
implies the need for
$out = f_1(x)$ prior to the enabling of F2.

$f_1(out) = f_1(x)$
implies the need for
$out = x$ prior to the enabling of F1.

$in = x$
implies the need for
$in = x$ prior to the enabling of *LOAD*.

We find out that the computation can be implemented by "LOADING" the x from the input and enabling "F1", "F2", and "F1" in sequence.

6.3. Design Transformations

As the computer aids to design mature, design tasks traditionally in the domain of the human designer, become new candidates for computerized aid. Design optimization is a good example of this kind of activity which has received much attention in the recent years [3,15,16,17].

The capability of transforming a given specification of a design to the specification of another, functionally equivalent design, is a necessary operation of any optimizing program. It is through the application of these transformations, guided by some criteria of optimum search, that optimization is mechanized.

In this section, we will demonstrate, through an example, that the proposed notation provides the necessary mechanisms for transforming designs among functionally equivalent implementations. No suggestion of universally optimum result is implied by the example. Our discussion will be limited to demonstrating the notation's adaptability to the kind of operations needed for the writing of more sophisticated optimization algorithms.

In this example we propose a simple algorithm, operating on SDC-based specifications, transforming a highly parallel, single step computation into a highly serial single ALU equivalent network. To better describe the steps of the algorithm we apply the procedure to an optimization problem proposed in [6]. We start by describing the problem.

Suppose we are given the initial design

shown in Figure 6-6-a. This data-path represents a highly parallel circuit which computes two output values as a function of two input values at every beat of the input clock. The input values are presented with the beat of the clock. Lacking any *delay* element, the output will be available during the same clock pulse. For large circuits of this kind the clock period should be long enough to allow for the propagation of values along the longest path of the circuit. Such data-paths must not have any closed paths. We assign the value "zero" to all nodes of this data-path, indicating that they all operate at the same beat of the clock, (i.e. the beat "zero"), on different steps of the same computation.

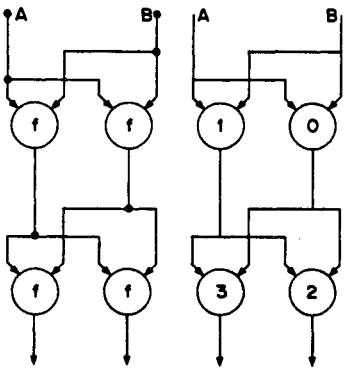


FIG. 6-6-a.

FIG. 6-6-b.

We are interested in transforming this circuit into an equivalent circuit in which a single ALU is multiplexed between the functions performed by the individual *combinational* elements. The steps to be taken in transforming such parallel computation networks into the fully sequential equivalent are as follows:

1- Assign an integer delay value i, $1 \leq i \leq n$ and n is the number of *combinational* elements in the circuit, to every node of the graph. The re-timed value of every successor node should exceed that of all its predecessors. The result of this re-timing, applied to Figure 6-6-a, is shown in Figure 6-6-b.

2- Add m *delay* elements to the data-path to guarantee the correct arrival of related tokens of data at every *combinational* element. Figure 6-6-c shows the network after the introduction of delays. Ignoring the pipelining possibility, each delay element will be used once during the lifetime of a computation. The period of usage of every delay element is shown next to its symbol. Terminate this step by assigning a unique name to every *net* of the data-path.

3- Draw the usage chart of the m *delay*

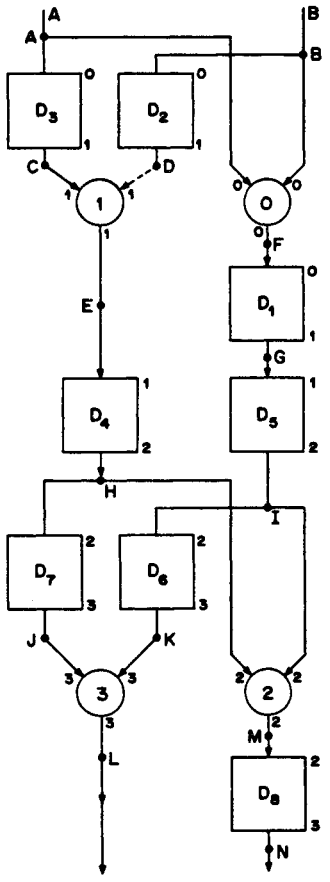


FIG. 6-6-c.

elements. Every *delay* element appears once in the life-time of every computation, Figure 6-6-d.

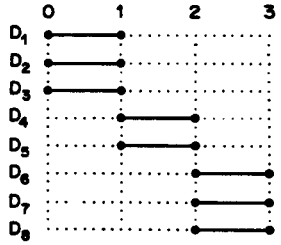


FIG. 6-6-d.

4- Draw a new usage chart, using only k delay elements, where k is the maximum number of *delay* elements needed simultaneously. The new *delay* elements are multiplexed among those used singly.

Allocate the m singly used delay elements among the k multiplexed elements, trying to assign those used in series to the same member, Figure 6-6-e.

5- Design a new circuit by selecting a single ALU and k *delay* elements. Label every input and output of the multiplexed elements with the names of all *nets* they are being multiplexed among.

Precede every input multiplexed between $j \geq 2$ places with a j input selector. Label the selector inputs with the multiplexed *net* names. The result of this element selection and labeling procedure is shown in Figure 6-6-f.

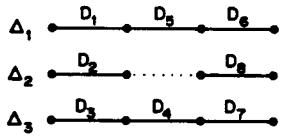


FIG. 6-6-e.

6- Complete *net* connections with identical names. Select a unique name for multiple named *nets*. Simplify selectors by grouping several inputs of the same name with a single input. The result of this step applied to our example is shown in Figures 6-6-g and 6-6-h. Realizing that two of the selector-delay combinations, with a tight feed-back loop around them, are in fact regular register, transforms the design to that of Figure 6-7-i which is almost identical to the one reported in [6] as the most economical realization of the proposed circuit. In our scheme we have realized that one of the three registers, as proposed in [6] is in fact a simple delay of one unit which in most VLSI implementations has a simpler realization than that of a static or dynamic register.

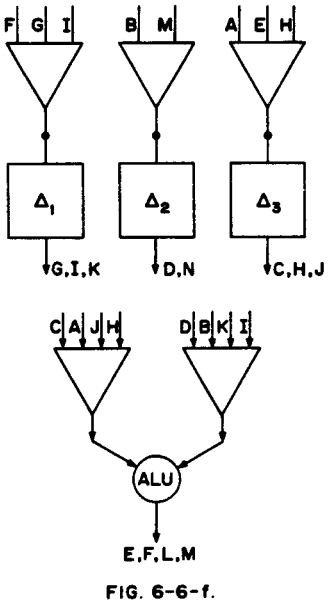


FIG. 6-6-f.

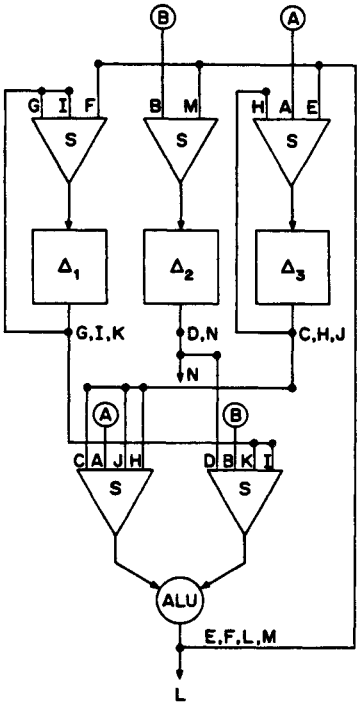


FIG. 6-6-g.

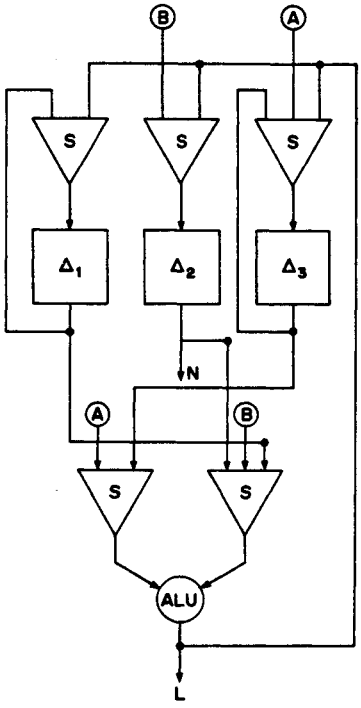


FIG. 6-6-h.

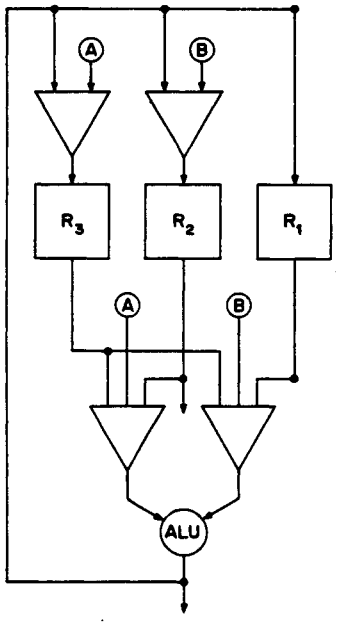


FIG. 6-6-i.

7. Acknowledgements

The research described in this report was supported by the National Science and Engineering Research Council of Canada grant numbers G1140 and A5515.

8. REFERENCES

[1] Anceau, F., "CAPRI: A design Methodology and a Silicon Compiler for VLSI Specified by Algorithms", Third Caltech Conference on Very large Scale Integration, Ed. R. Bryant, california Institute of technology, Computer Science press, 1983, pp. 15-31.

[2] Barbacci, Mario R., Syntax and Semantics of CHDLs, *Computer Hardware Description Languages and Their Applications,* Eds: M. Breuer, and R. W. Hartenstein, North Holland Publishing Co., Amsterdam, 1981, pp.23-36.

[3] Darringer, J., W. Joyner, "A New Look at logic Synthesis", Proceedings of the 17th. Design Automation Conference, IEEE Computer Society, June 1980, pp 543-549.

[4] Estrin, Gerald, "A Methodology for Design of Digital Systems- Supported by SARA at the age of one", AFIPS Conference Proceedings, Vol. 47, National Computer conference, 1978, pp. 313-324.

[5] Gordon, M. J. C. (1981), Register Transfer Systems and Their Behavior, in *Computer Hardware Description Languages and Their Applications,* Eds: M. Breuer, and R. W. Hartenstein, North Holland Publishing Co., Amsterdam, 1981, pp.23-36.

[6] Hafer, L. J., A. C. Parker (1983), "A Formal Method for Specification, Analysis, and design of Register-Transfer Level Digital Design", IEEE Transactions on Computer Aided Design, Vol. CAD-2, No. 1, January 1983, pp. 4-18.

[7] Hartenstein, R. W. (1982), Basics of Structured Design Methodologies: Data-Path and Finite State Machines, in *Design Mathodologies in VLSI Circuits,* Eds: P. G. Jespers, C. H. Sequin, F. Van de Wiele, Sijthoff and Noordhoff Publishing, 1982, Rocjville, Maryland, USA, pp. 73-107.

[8] Hitchcock III, Charles Y., "Automated Synthesis of data-paths", Research Report No. CMUCAD-83-4, Centre for Computer Aided Design, Dept. of Electrical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213.

[9] Johannsen, D., "Bristle Blocks: A Silicon Compiler", Proceedings, 16th. Design Automation conference, July, 1979.

[10] Johnsson, Lennard, Danny Cohen, A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks, *VLSI Systems and Computation,* Eds: H. T. Kung, Bob Sproul, and Guy Steele, Computer Science Press, 1981, pp. 213-225.

[11] Mavaddat, F., "High Level Approaches to VLSI Design", Digest of papers, 1983 Canadian VLSI Conference, Waterloo, Ontario, Oct. 1983, pp. 5-11.

[12] Losleben, P. (1980), Computer Aided Design of VLSI, in *Very Large Scale Integration: Fundamentals and Applications,* Ed. D. F. Barbe, Springer Verlag, Berlin, 1980, pp 89-127.

[13] Shrobe, Howard E., "The Data-Path Generator", Proceedings, 1982 Conference on Advanced Research in VLSI, M.I.T., 1982, pp. 175-181.

[14] Siskind, J. M., J. R. Southard, K. W. Crouch, "Generating Custom High performance VLSI Designs from Succinct Algorithms", Proceedings, 1982 Conference on Advanced Research in VLSI, M.I.T., 1982, pp. 28-39.

[15] Snow, Edward A., Daniel P. Siewiorek, Donald E. Thomas, "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Design tradeoffs", Proceedings of the 15th. Design Automation Conference, IEEE Computer Society, June 1978, pp. 220-226.

[16] Thomas, D. E. (1981), "The Automatic Synthesis of Digital Systems", Proceedings of the IEEE, Vol. 69, No. 10, October 1981, pp. 1200-1211.

[17] Thomas, D. E., C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, and R A. Walker (1983), "Methods for Automatic Data-Path Synthesis", Computer, December 1983, pp. 59-70.

[18] Tseng, Chia-Jeng, Daniel P. Siewiorek, "The Modeling and Synthesis of Bus Systems", DRC-18-42-82, Digital research center, Carnegie-Mellon University, Pittsburgh, PA 15213.

[19] vanCleemput, W. M., "Hierarchical Design for VLSI: Problems and Advantages", Caltech Conference on VLSI, Proceedings of Caltech Conference on Very Large Scale Integration, Caltech computer Science Dept., Califonia Institute of technology, January 1981, pp. 259-274.

[20] Zimmermann, G., "The MIMOLA design system: A Computer Aided Digital processor Design Method", Proceedings of the 16th. Design Automation Conference, IEEE Computer Society, June 1979, pp. 53-58.