Multilevel Simulator

Erik P. DeBenedictis

10 May 1979

Masters Project Report

Department of Electrical Engineering
Carnegie-Mellon University

Table of Contents

1. Introduction

Until recently computer aided design of digital systems has involved various programs, each of which solves its own particular problem. When these programs were written they were intended to be optimal at finding solutions to one person's interpretation of a particular design task, without concern for compatibility with other programs, or expansion of capabilities as design needs change. When a particular user designs using these programs he will typically operate in cycles. The cycles involve manually making input to a program, running the program, and analyzing the output in preparation for the next cycle. Out of a space of all possible design tasks these programs can be represented as points, with the user connecting these points, and the input and output states, by his own intelligence. If a design system could be developed whose functions are both compatible with each others' inputs and outputs, eliminating the need for translating by hand, and intersect the space of design functions in exactly the way each user wants, computer aided design would be more efficient. An ideal computer aided design system would be one that completely fills a portion of the design task space, rather than being just a point. What is presented here is a discussion of how these considerations can be applied to define a system with these desirable properties. Such a system solves the compatibility problem by serving as a database for information about the design in its various forms. The system is also organized in such a way that the user can, and is encouraged to, define functions to soive his own problems.

To give some idea of why this is useful consider some functions that cannot now be performed, but that an ideal system should be able to perform: There are essentially two types of simulators, gate level[8, 2], and behavioral level[7, 1]. Previously this meant that there were two programs, one for each type of simulation. All simulations had to be done with devices described entirely at one level. Unfortunately the user's problem often cannot be solved directly in this way. The user will usually want to know how one part of the design will respond in an

environment with the other parts. The user should be able to simulate a part of the design at some level with the rest of the design at a more abstract level. This capability is not in simple simulators. The user instead has to abstract by hand the behavior into the input format of the simulator and interpret the results. An ideal system would allow the different simulators to be run simultaneously on a design partially defined to each one. Consider now the documentation of a design. Say the user wants a schematic drawing of his design. Since this information has been used by perhaps the simulator and the wire-list generator there should be no reason that the user should have to provide this information again. Using separate design programs, however there would be two different descriptions for the simulator and the wire-list generator, neither of which can be used to produce a schematic. In fact it is difficult to verify that the simulator's description is the same as the wire-lister's. For an ideal system, where the descriptions are all compatible, and organized by their structural features, this task is trivial. As a final example, it should be pointed out that there are some programs in existence that do several of the tasks outlined above, like gate level simulation, wire-lists, and schematics[3, 5]. These programs are generally very large and complex, they probably took a team of programmers years to develop. The documentation for these systems describes only how to use those features supplied by the factory. If the user wants an additional feature he may either rewrite the program himself, or ask the factory for the additional feature. An important aspect of an ideal system is that it would be designed with expansion and changes in mind. There would be a formalism the user could follow to make small or major changes at even the most basic levels.

Our approach is to make a system, called the interconnect system, that serves as a binder for all the various design functions. Each of the design functions would be written to run under the control of the interconnect system. The interconnect system would provide information about the design from a database, would provide system procedures to the design functions for manipulating the database, and would provide utility procedures. By having one interconnection system to run all the design functions we solve the compatibility problem. In fact, there is no fundamental reason why several design functions cannot be performed simultaneously in the

same program. This supports a solution to the multilevel simulation problem mentioned. Further, providing a description of the interface between the interconnect system and the design functions allows the user to define his own design functions, somewhat solving the expansion problem.

This paper discusses the design, implementation, and results of such a system. We do this by discussing the interconnection system first, and the issues involved in making it support as many different design functions as possible. To show that we have been at least partially successful we discuss a multilevel simulator that was designed and implemented under the proposed interconnection system. Test runs made on actual implementations of the interconnection system and the simulator did describe and simulate designs as they were predicted to do, the results of which are discussed.

2. Interconnect System

The interconnect system has been presented as a totally new concept. While in some respects it is new, in many respects it draws upon concepts from existing, but totally unrelated systems. The only really new idea in the interconnect system is to draw together these particular ideas into a single CAD system. To begin this discussion of interconnect systems we should mention these systems, and their attributes that are used in the interconnect system. A familiarity with these systems will convey much of the philosophy of the interconnect system.

As a start, almost all CAD programs take some input which describes an interconnection. Circuit simulators use connections of transistors and resistors, etc., wire-listers use connections of integrated circuits, display programs use connections of graphical modules, and so on[2, 8]. This system combines many of the interconnection forms of other CAD programs. Since conceptually all interconnection forms are very similar, the combination, although more complex, is different only in the details. Retrospective analysis of the implementation of the interconnect system allows us to propose a feature derived from the macro facilities of conventional CAD programs, but significantly more powerful.

Secondly this system will need to use its own interconnection structure to connect the proper elements for a particular task, and return the output after the task has been performed. This is just the function of a database manager. A portion of this system is a database manager. The database contains the interconnection information and the primitive elements of the interconnection[4].

Finally, some of the attributes of this system related to its capacity for expansion are drawn from the LISP system. LISP has two noteworthy qualities in this context, it allows the user to add lines of code to his program while running, and the user can define parts of programs over a very wide range of levels. The system is structured so the user can almost always add to a program at both the high and low ends. These qualities are why the largest programs in existence can be made and maintained in LISP. It is hoped that by copying some of these characteristics of

LISP this system will also be expandable. LISP has a serious drawback concerning efficiency, however. To overcome this drawback this system is designed with some of the philosophy of the programming language BLISS in mind. This language provides the user with many high level features, yet is very careful not to restrict the user with respect to what is not well supported. We incorporate these ideas into this system by providing system procedures of great power, but also allow the user to bypass most of these features if they become a hindrance.

The rest of this chapter will discuss the interconnection system by building it from of the various unrelated systems from which it borrows ideas. The outline will be along the lines of the last paragraph, it will start with those attributes most applicable to CAD and will progress to supporting systems. First there will be a discussion of the form of the data structure used to represent general interconnection. This is followed by a section on the issues concerned with managing a database for the interconnection information, and how this can be used to gain additional features. Finally a short discussion of a supporting software system is given. Although the supporting systems may be totally unrelated to CAD, they are crucial to the success of this system, and were developed especially for it. We do, however refrain from discussion here of the actual design functions. The following chapter is devoted to simulation, and what additional capabilities are possible when a simulator is implemented with this interconnection system.

2.1 Interconnection Structure

The most application specific part of the interconnection system is how information about a design is represented. Naturally some of this is totally specific to an application, but as much information as possible should be kept in a common form. As shown previously the information required by any particular function can be abstracted as an interconnection of more primitive application specific data. By primative we mean information about the design after all the interconnection information is removed. The simulation behavior of a register transfer element is an example. The behavior is the same regardless of how the element is connected.

We can now develop most of the concepts of the interconnection structure. In providing a system which combines the different functions we run into the multi-level problem. This problem is that while all CAD functions use interconnections of primitives, a primitive of one function may be equivalent to a whole interconnection of primitives of another function. A typical example of this is circuit level and gate level representation of designs: each gate is an entire connection of circuit elements. The solution to this is normally accomplished by a macro facility. We propose for this system a feature that includes the capabilities if a macro facility. We achive greater power in our implementation by allowing a macro part to be on every element.

2.1.1 The Element

The fundamental data structure in the interconnect system is called the element. Each of the application required primitives is associated with an element data structure. The association is realized by a pointer in the element that points to the primitive to be connected. A pointer, a fundamental concept in programming systems, is a word in storage with the address of a data structure. Interconnection is represented by pointers in the elements, pointing to other elements. Connections are not necessarily as simple as pointers, a connection normally involves two pointers and has other data associated with it. Elements also have labels, to uniquely identify them, and names, to tell what kind they are. Pointers to inferior and superior elements, and parameters, are present, but are more obscure. All of this will be discussed in turn.

All previous design programs have interconnection elements that may each have several independent connections. For example, a transistor will have four connections, a two input gate three. We formalize this concept by giving each element a number of ports, and allow the ports to be connected together. A port is an association of pointers and other data on an element that can be identified. A connection therefore is a pointer to an element and some data identifying the selected port. Port connections must be somewhat bi-directional, i.e. both elements

must be able to determine what other elements they are connected to. This is for many reasons. Consider simulation, where electrical signals may travel across connections in both directions. It is also very desirable to have bi-directional pointer sets for storage allocation purposes.

As an example of a connection consider an adder and latch from an integrated circuit design. Each will be shaped like a rectangle whose tops and bottoms will be studded with input and output connections. Through the sides will come power and control, like clock and carry signals. Within are transistors. The example is to describe the connection of the latch to the adder. Visually this would be accomplished by joining the two rectangles along a common side. Since on an integrated circuit only two solids may meet at a line we can use a connection that is only for connecting two elements together. The simplest way to do this is to take the two ports corresponding to the matching sides and direct their connections at each other. This most basic connection is called a direct connection. Another example of when a direct connection is used is when a functional simulation element connects to one of its subdevices. These examples have the property that they make sense when two elements connect, but not when three or more do. This is just as well because three or more pointers cannot each point at all the others. Figure 2-1 depicts a direct connection.

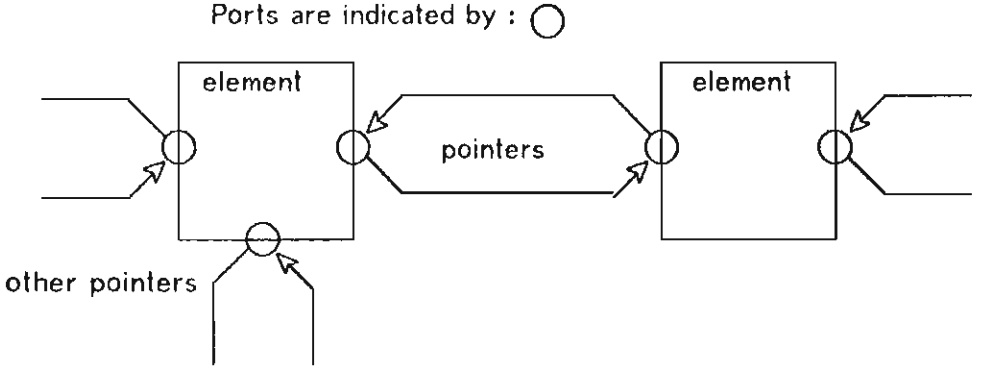

Figure 2-1: Example of Pointer Structure in a Direct Connection.

2.1.2 Wires

There are many cases where more than two elements need to connect. The best example of this is connection by wire. Consider wire wrapping computer backplanes, where groups of pins, typically an output and a number of inputs, are connected together. The number will be variable in that an arbitrary number of pins can be wired together. Connection by wire is done by making the wire an element itself. One way to implement such a connection would be to put however many ports are necessary on the wire, giving each port a direct connection to the selected port on one of the other elements. Another implementation is to have the selected ports on the other elements point to the wire. A second pointer can then be added to each element and the wire, and these pointers connected into a ring. Now, in both cases, the requirement that connections be bi-directional is satisfied, since each element can find all of the devices connected to it by following pointers. In the first case the connections are the simple, direct, ones, but there are a variable number of ports on the wire. In the second case the pointers are more complex, but there is exactly one connection to a wire. In an actual system only one method will be used for implementing wires, but we present the details of both methods to make clear what the properties of this type of connection are. Figure 2-2 shows an example of a wire connection.

2.1.3 Interpretation of Interconnect

We have seen a way of representing interconnection. Since the discussion refered repeatedly to pointers, the user might think these are details of implementation, and that there is some written way of representing interconnection that is the definition of interconnection. This is common in other design programs, but is not done here. Data structures in a computer connected by pointers form a simple and extremely powerful concept, but trying to make a language to describe such potentially random interconnections is cumbersome. We therefore define our interconnection structure to be the internal computer representation. There is also a written form of the interconnection structure, that can be thought of as an

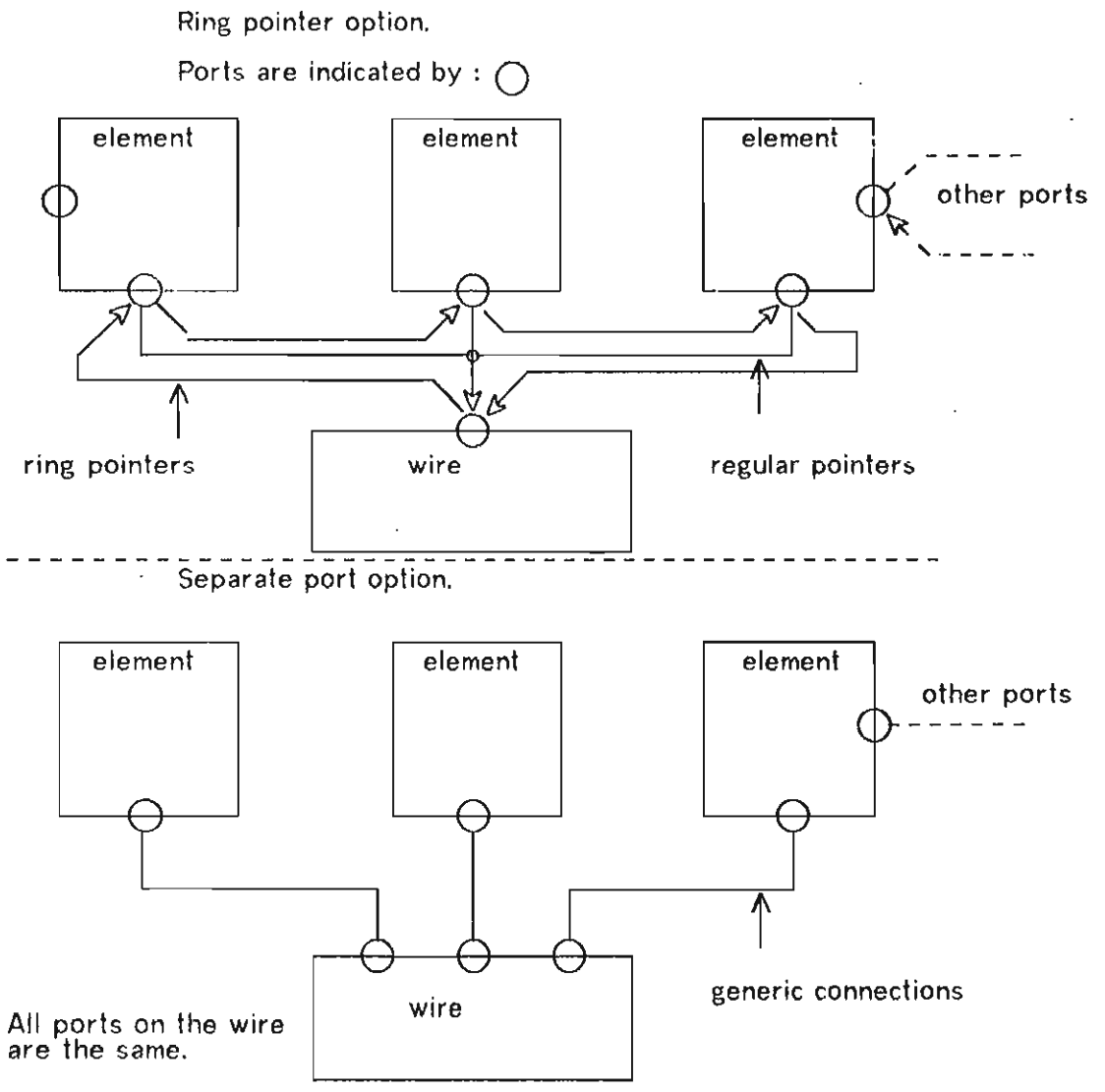

Figure 2-2: Example of Pointer Structure in a Wire Connection.

afterthought. Such a language is necessary if for no other reason than to allow the linked core structures to be stored in files and regenerated later. Usually the user will visualize an interconnection graphically, using shapes for the elements and lines to represent interconnections. The shapes and lines map directly to the internal representation, and not well to a written representation. Even in a graphical representation some of the features of a written language, if present, would be used, however. For example, designs will often assign words to various wires and

elements as a mnemonic for remembering what they are. A written representation is not only a necessary evil, as previously mentioned, but may be a source of useful concepts to aid our understanding of an actual interconnection. We therefore develop a written representation, from which we will later take what is useful from, but will seldom use it in its entirety.

2.1.4 Written Representation

The first step in developing a written representation is to find a way of describing interconnections. To do this we need a way of uniquely identifying each element, and the ports on the element. The first is solved by associating with each element a label, or a word unique to the element in a particular context. A direct connection, in addition, needs a way of identifying a particular port on a particular element. The simplest way to do this is to number the ports, and describe a port by both its element's label and the port number. This is adequate, although a system can be made nicer by allowing mnemonic words instead of numbers to describe the ports. Since a wire has only one type of connection, it is in one sense simpler. If we know that a connection is of type wire there should be no need to specify which port. We unify this concept by allowing one port in the element to be a special, or default port. A wire connection could be made by specifying the label of the wire, omitting the port. The default port would be used. This is equivalent to saying that connection by wire is several elements connecting to a wire directly, rather than to any port.

Now that we can identify the ports involved in an interconnection we can show how the connections of an element are described. At this level in the language we view connections from the elements. The language will show how ports on a particular element connect to other ports in the design. This The connection information for each element will be a list associating each of its ports with a port on other elements. means each connection will appear twice, once from each end. There is an opposing view, however where the description is ordered by connection, and the elements associated with a connection are listed. Making wires like other

elements means we can both look at one element and see the wires it connects to, as well as look at a wire and see what elements it connects to. We therefore effectively support both views of connections.

The next level of the language involves representing a design as a number of connected primitives. The design is the entire specification for some system being constructed, or perhaps a portion of a larger system. For now our design will be a closed system, all connections are between two ports both in the same design. Later, when macros are introduced, we show subdesigns, each like a complete design itself, incorporated into a larger design. These subdesigns may not be closed, in a sense connections outside a particular design will be possible. We have shown already how to represent the connections of each particular element, and we must now further identify each element, and show how they are connected into a design. We have introduced labels to identify elements uniquely, but we should have a way of identifying elements of the same type. For instance, we have refered to elements of type wire being used for wire connections, but as yet have no way of telling a wire from some other element. This problem is solved by having another word associated with each element, called its name. Examples of names could be "wire", "NandGate", and "CPU". The second problem is solved by allowing individual element's connection descriptions to follow each other, separated by lines or special characters. A design will be represented be a series of lines, each specifying a label, name and having a list of connections for one element.

2.1.5 Examples of Interconnections

We can now give an example of a simple interconnection. Consider two elements connected by a wire. The elements may be any primitive, circuits oriented readers may think of transistors, the elements could also be gates, or a processor and memory module. The elements may have other ports. We represent an unreferenced port by the symbol "-". Figure 2-3 shows a wire connection, abbreviating the pointer structure to where only the port connections are shown. In practice the pointer structure below the connection level will largely be transparent.

Figure 2-4 shows an example of the written representation of this interconnection. Similarly figure 2-5 shows the direct connection of our previous example of a register and adder on an integrated circuit. Figure 2-6 is the language representation.

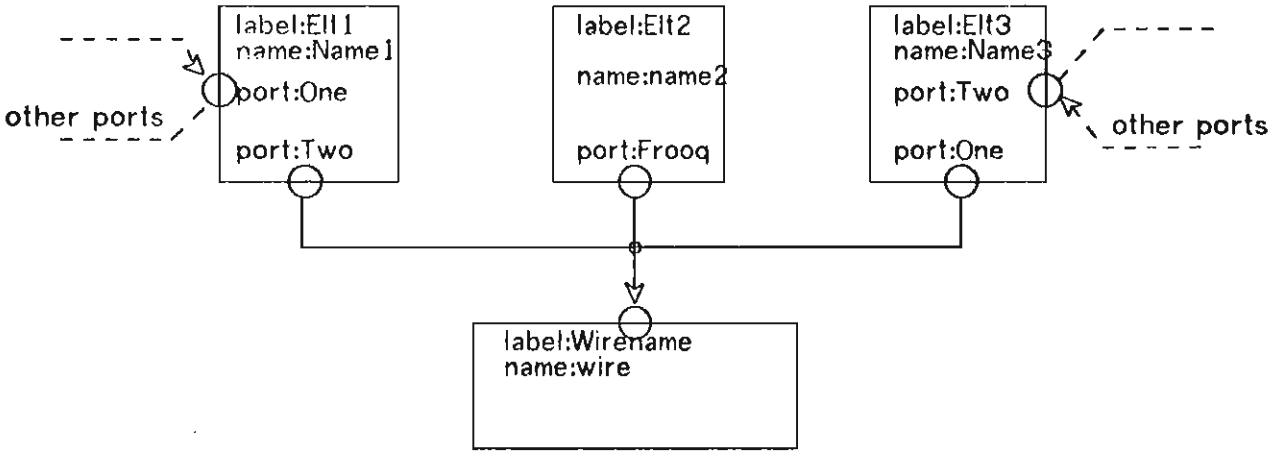

Figure 2-3: Core Structure Using Wires.

Numbered Ports:

Elt1:Name1	- Wirename ;	Ports are in positional order.
Elt2:Name2	Wirename ;	The wire is automatically
Elt3:Name3	Wirename - ;	generated and need not appear in the written description.

Named Ports:

Elt1:Name1	One=- Two=Wirename ;
Elt2:Name2	Frooq=Wirename ;
Elt3:Name3	One=Wirename Two=- ;
Wirename:wire	-=Elt1.Two -=Elt2.Frooq -=Elt2.One ;

Figure 2-4: Written Representation of Figure 2-3

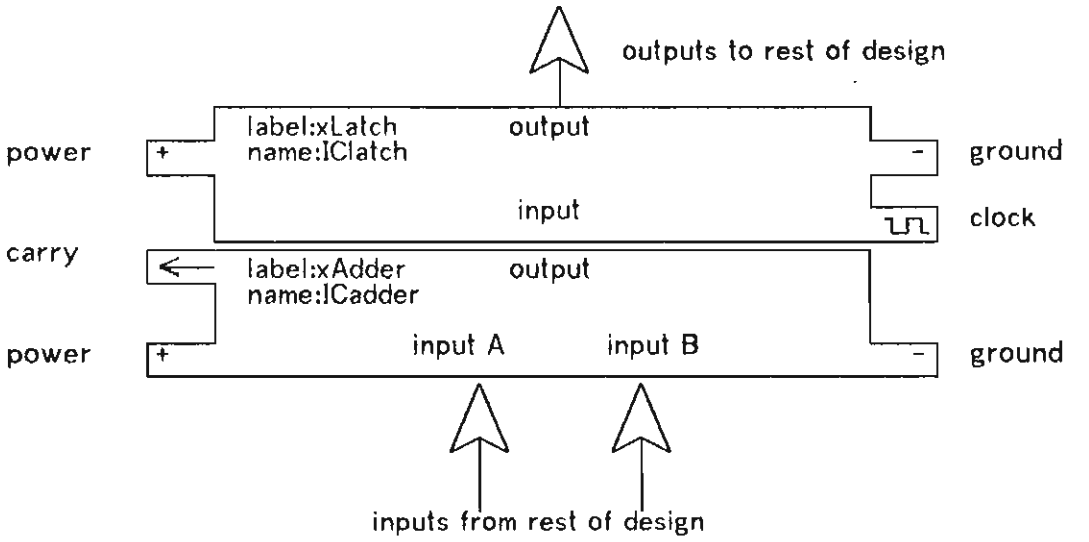

Figure 2-5: Core Structure Using Direct Connections.

```
xLatch:IClatch      input=xAdder.output output=- clock=- power=- ground=- ;
xAdder:ICadder      InputA=- InputB=- output=xLatch.input carry=- power=-
                    ground=- ;
```

Figure 2-6: Written Representation of Figure 2-5

2.1.6 Macros

The macro concept leads to a simple extension of what we have previously been calling the design. The proposed facility, that has the same function as a conventional macro facility, will be developed first as a conventional macro facility. It will then be shown to be more powerful due to the manner in which the whole facility can be incorporated into this system. We previously viewed the entire design as being a single interconnection of elements. In some sense this is an adequate way to view any design. For example a supercomputer will be composed of a few hundred thousand transistors and resistors. It is straightforward to model this as a few hundred thousand connected elements. A feature of most real designs is that it is possible to choose subsets of the elements such that the number of

interconnections to elements out of the set is small compared with the number inside it. The supercomputer example was probably made of integrated circuits, each having sixteen interconnections through its pins and hundreds within itself. Being able to divide a design down this way makes it significantly more understandable. The macro is a way of dividing a design down in this way.

Initially we can view the macro as being a dotted line surrounding a group of elements. The dotted line is routed so that the fewest number of connections pass through it, most will be entirely on one side or the other. The macro is strictly the dotted line only, and the elements within are the macro's inferior elements. If such a division were made in most designs it would be found that many of the dotted lines enclosed groups of elements connected in identical ways, the only difference being where the whole groups are situated in the design. This leads to giving each of these dotted lines a name, which identifies its contents. It would be most desirable to be able to have only one definition of each similar macro, rather than having to specify each instance. With the interconnection structure so far developed this is not possible, as a particular port within the dotted line would be connected to a different port in each instance of the macro. We solve this problem by actually putting ports on the macro. The elements inside can connect to their enclosing macro as though the port were inside the dotted line, whereas elements outside the macro connect as though they were outside. Now the interconnections within similar macros are identical. The part of the design outside the macro can connect to the macro through its ports, just as though it were any other element. In fact, the dotted line enclosing elements is an element itself. We now generalize elements to have inferior elements. Our notion of a dotted line is replaced with the solid line notation of elements, where there can be other elements inside.

Previously when we had developed only the concept of the design, how the individual elements connected together was not too important. There was only one design and every element belonged to it. We now need to formalize a way of associating elements into groups, both to implement macros and the main design. The simplest way to make groups of elements is to give each element a pointer and

have each element point to the next member of its group, in a ring. If the group is a macro then the element which actually implements the macro, i.e. the line around the group, needs to have a pointer to the group, or at least a first element. Similarly there needs to be a pointer to point to the first element of the whole design. Every element is given a pointer to inferior elements, from which it can find the rest by following the ring. There is similarly a global inferior element pointer, which points to an element, as though that element were an inferior element in some more global element. This concept is extended one step further to say that there is no global design, but there is a top level element. That is, there is a name for the entire design, and there may be ports at the outermost level, like the ports visible from the inside of a macro.

Combining the basic elements and macros allows us to make an extension to the familiar concept of the macro. We allow an element to have both inferior elements, for which it serves as an enclosure, and primitive characteristics of its own. The most direct application of this is to functional descriptive languages. A typical description will have a main part and some number of subparts. The subparts may further have subparts. The representation of such a description will be an element with the characteristics of the main part, with inferior elements, each with the characteristics of the subparts. Figure 2-7 demonstrates some of the pointer structure within an element with the macro facility.

2.1.7 Written Representation of Macros

We now need to extend the written representation to accommodate the macro facility. We first need to consider how to represent connections from inside an element to the ports on the macro. We do this naming ports outside the immediate enclosure by the name of the macro and the port name, or the name of the macro and the label of one of its inferior elements and that element's port name. Notice of the use of both names and labels. This gives us considerably more capability than needed just to connect to the ports of the immediately enclosing superior element. In fact this gives us access to all ports at a lower level of macro nesting, in a

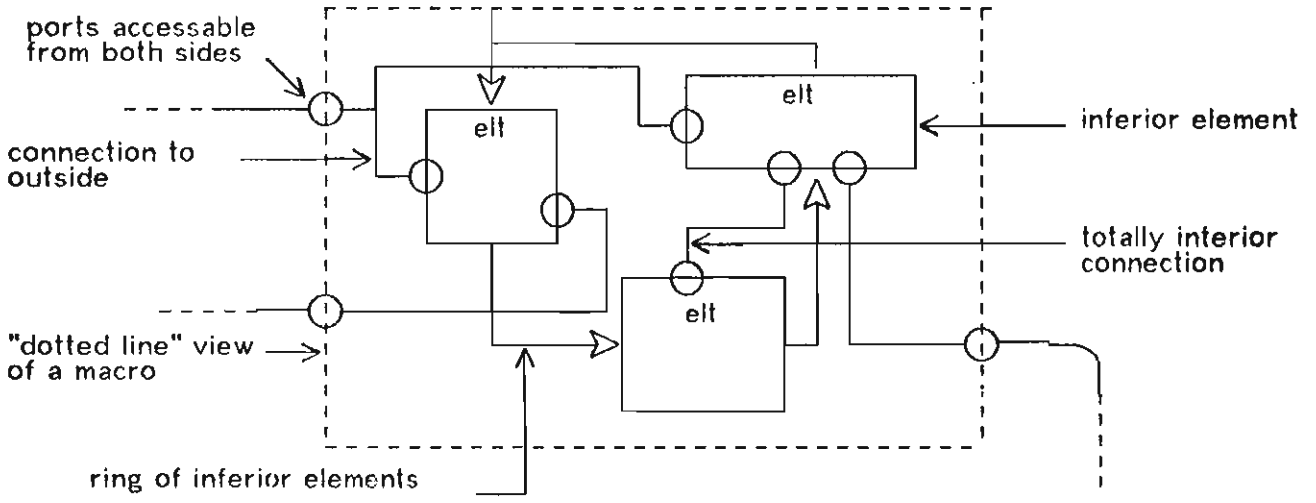

Figure 2-7: Example of Inferior Element Pointer Structure.

manner similar to variable accessing rules in ALGOL like programming languages. To show how this is used, a port of the immediate macro is accessed as though the macro were in the present group with its name being its label. If there were a nesting of several macros within macros it would be like each of those macros were in the present group, with names being their labels. If it were desired to connect directly from within a macro to a port outside, it could be done by replacing the port name of one of these virtual elements with the label and port name of one of its subelements.

We can now give some rules for naming and finding ports. These rules are with respect to the element they are being named from. The name is like a map of how to go from one element to some port. Firstly the ports of the reference element are scanned to see if their port name matches the element's name, if so then that port is named. Otherwise the labels of the other elements at the same level are scanned. If there is a match then the second part of the name identifies the port on that element. Otherwise the names of the chain of elements representing outer macro levels are scanned. If there is a match then the two following parts of the name are the label and port name of one of its inferior elements. Not all ports can

be named from every element. This means that some connections, like from an element to an outer macro level, will be named only once, rather than twice as are normal connections. The way in which whole macros are manipulated by the system is a subject if the next section.

2.1.8 Parameters

We now digress and discuss a feature that has been passed over. It is sometimes desirable to have elements that are the same in many respects but have different parameters. For example a propagation delay might be a parameter for a gate, an initialization file name a parameter for a memory, etc. To provide this we allow elements to have parameters. Parameters may be numbers, or identifiers, or whatever the user wants to implement. Parameters are grouped into parameter sets, each of which has a set name. Each element can have an arbitrary number of parameter sets. Parameters have no real conceptual significance, but they make the system easier to use.

2.2 Database Facilities

A sufficient and convenient technique for storing all the information about our interconnections is a database organized by the names of elements. Under each name are entries for primitive descriptions and descriptions in terms of inferior elements. For a little more flexibility these entries are grouped together under option names, and there may be several options for each element name. Since the system treats the whole design as though it were an element, the whole design will be stored under its name. A main design's entry would likely have no primitive description. Its inferior element description would contain only the immediately inferior elements. If the inferior elements had inferior elements themselves they would have an entry in the database with a non-empty inferior element part. At some level there would be entries for devices with no inferior element part but a primitives part.

Consider the assembly of an interconnect structure from the database. The user

will provide a name for the top level element. The database is then called to provide primitives to the system, and to setup the inferior elements of the top element. At this point there are elements which are partially represented, i.e. they are connected to elements on the same level but have no description. The user will then instruct the system to build upon these incomplete elements by getting their descriptions from the database. The user will be supplying options to the system so the proper definition of each element is used. If the design is something realizable then eventually all the elements will be complete. It is possible, however to make recursive definitions of elements that cannot be built. At any stage the user can stop the building process and do editing of the structure or invoke design functions.

It is also possible to return the structure to the database without losing any information. This is possible because we defined the interconnection structure as being the internal representation, rather than the written one. By definition, therefore we can get all the information known about an element from the interconnection structure. When applied to macros this becomes a significant extension to the common macro concept. Here macros are defined in terms of their instance. We run in to an interesting issue here due to the existence of multiple copies, and hence multiple definitions of an element. If there are several instances of the same element in a structure, and one of them is changed, then whether the change becomes permanent depends on the order they are returned to the database. This is an unacceptable situation. To find a solution to this we must consider the different ways in which a change might be made to the interconnection structure. The first case is that there is one instance of an element and the user wants to change it. There is no problem here as long as the database is updated properly. A second case is when a user wants to change one instance of an element while leaving the others the same. To accomplish this the user needs to rename the one element so there is no conflict of definitions. The database then needs to have a copy of this new, but similar element. A third case is if the user wants to change all instances of an element in the structure. This could be accomplished by finding all instances of that element and changing them simultaneously. A better way might be to rename, perhaps automatically, one

instance, change it, then instruct the system to rename back the element and fix the other instances. Any system should have all three capabilities. Figure 2-8 shows an example of the database.

```
Z80
Functional option:
        Simulation description:
                {Relocatable simulation code.}
        Graphic description:
                {Nothing, since this is a simulation entry.}
        Inferior element description:
                {Part1:Z80IF            Z80.IFport ;
                Part2:Z80RD             Z80.RDport ;
                Part3:Z80WR             Z80.WRport ;}
Physical option:
        Simulation description:
                {Dummy code to echo simulation messages through
                the element.}
        Graphic description:
                {Nothing again.}
        Inferior element description:
                {Gate1:Nandgate         A B C ;
                Gate2:Norgate           D E F ;
                ... and so on ...}
```

The first option is for a functional description. The description has a main part and three subdevices corresponding to instruction fetching, reading and writing. The second option is a pure macro description, describing the Z80 as an interconnection of gates.

Figure 2-8: Example of a database entry for a Z80 processor.

2.2.1 Interconnection Editor

We have been discussing changes to the interconnection structure without telling how they might come about. One of the functions of the interconnection system is to provide an interactive editor for the interconnection structure. A design task will typically start with an empty database. The user will build the system by adding

and deleting subelements from the various elements, including the top level element. These changes will be done by the editor. Naturally, the more powerful and easy to use the editor is, the more useful the system will be. There are, however very few guidelines which need to be presented for the editor. Essentially the only requirement for the editor is that it not produce interconnections which are either illegal, or cannot be stored in the written form. One additional function of the editor is that it must be able to manipulate the primitive descriptions, it must at least be able to enter them into the database.

2.3 Software Support System

At present no single programming system has sufficient capabilities to support the interconnection system. The key reason for this is that the primitive descriptions for simulation will be executable machine code, and the system should be able to incorporate this code into itself automatically. The ability to define code while running partitions programming systems, the ones that can do this are often worthless otherwise and the ones that can't are worthless for that reason. What is suggested is that a software system with the necessary capabilities be written especially for the interconnection system.

Such a software system will be built around a storage allocator. The unusual feature of the storage allocation system is that a code data structure will be supported. Other, more mundane, data types will also be supported. The code data type will be associated with the relocatable files of the computing system, that is a relocatable file can be read to make a code data record. The relocation information will be permanently retained with the record in case storage compaction would occur, or the record should be written out, or edited. By the nature of addresses in code, code records can only be moved by the inefficient process of garbage collection. To be more efficient in the storage allocation of better conditioned data structures, ring type pointers will be supported. This will allow incremental storage allocation of the data structures which are commonly used. In the process of further discussing these issues we will formalize the concept of a data record, and

describe its three types, code, relocation record, and other.

2.3.1 Data Record

The storage allocation system will do its allocation in a single contiguous core vector. This vector will be divided into records. The records will form a doubly linked list, an action requiring two words, the first and last, of each record. Records are allocated by finding an empty record which is long enough and marking it as non-empty. It is split into two parts if it is too long, the second part remaining empty. Records are de-allocated by marking a record as empty, possibly joining it with empty records at each end. In this way reasonably efficient storage allocation will occur in most cases. It is possible, however that a great many empty records would accumulate which are too short to be used. In this case a method of storage compaction is desirable. Since a garbage collector will be present for other reasons, we can use it to move all used records into a contiguous part of the core vector, leaving a single large empty record.

2.3.2 Code Records and Relocation Records

The contents of a code record will be executable machine code and data areas. Such a record may contain instructions which have an address within the record, or within another record of the proper type. The relocation record will identify which words of a code record are addresses and which are not. Most machines have addresses in different forms, such as displacements and absolute addresses, left and right halves. This information will also be in the relocation record for each address. All of this is contained in relocatable files.

In addition, relocatable files contain information as to entry points and symbols. These are easily implemented within the already developed structure. The start of a code record could contain a list of symbol names and address words, the symbol names being unrelocatable, and the addresses being relocatable. When the entire record is moved the symbols will remain valid.

The reader should notice that retaining relocation information makes it possible to add or delete words or instructions from inside a code record.

2.3.3 Other Records and Storage Allocation Routines

The records called other records are generalizations of the code record. They may contain addresses of types not found in instructions, and they do not necessarily require a relocation record. It is necessary that there be a way of identifying which words are addresses, what they actually point to, and changing them so that they are valid after a storage shuffle. The way this is implemented is for a word in each record, say the second since the first is already used, to point to an address in a code record that is the entry point of the marking coroutine for that record. It may be desirable for there to be additional data, so the third word will point to a record of unrelocatable data. The code and relocation records are now special cases, a code record would have the entry for the system's code marking coroutine in its second word and its third word would contain the address of the relocation record. Groups of identical data structures would have the same addresses in their second and third words. That is, they would share the same marking routine and relocation record. If a set of data structures were similar in gross features but different in details, a common marking routine could be written and each different data structure would have a different relocation record. This is now as efficient with storage space as common programming language systems, but significantly more flexible.

2.3.4 Moving Records

There are two types of records with respect to the amount of effort required in moving them. One is typified by the code record. Such a record may be refered to by pointers located in any other record. The only way to find all of these pointers is to examine the entire storage pool. Another type of record can be made with only ring pointers pointing to it and from it. This means that every pointer to it can be found by following only a few pointers contained in the particular record. If a record

of the first type needs to be moved, an entire garbage collection must be performed, whereas the second can be moved by simple incremental techniques. Both types of records are supported. This is because it is realized that records that require frequent moving can be made with just ring pointers with a little effort, whereas records known to be stationary can be allowed to take the simpler form. Figure 2-9 depicts the core structure.

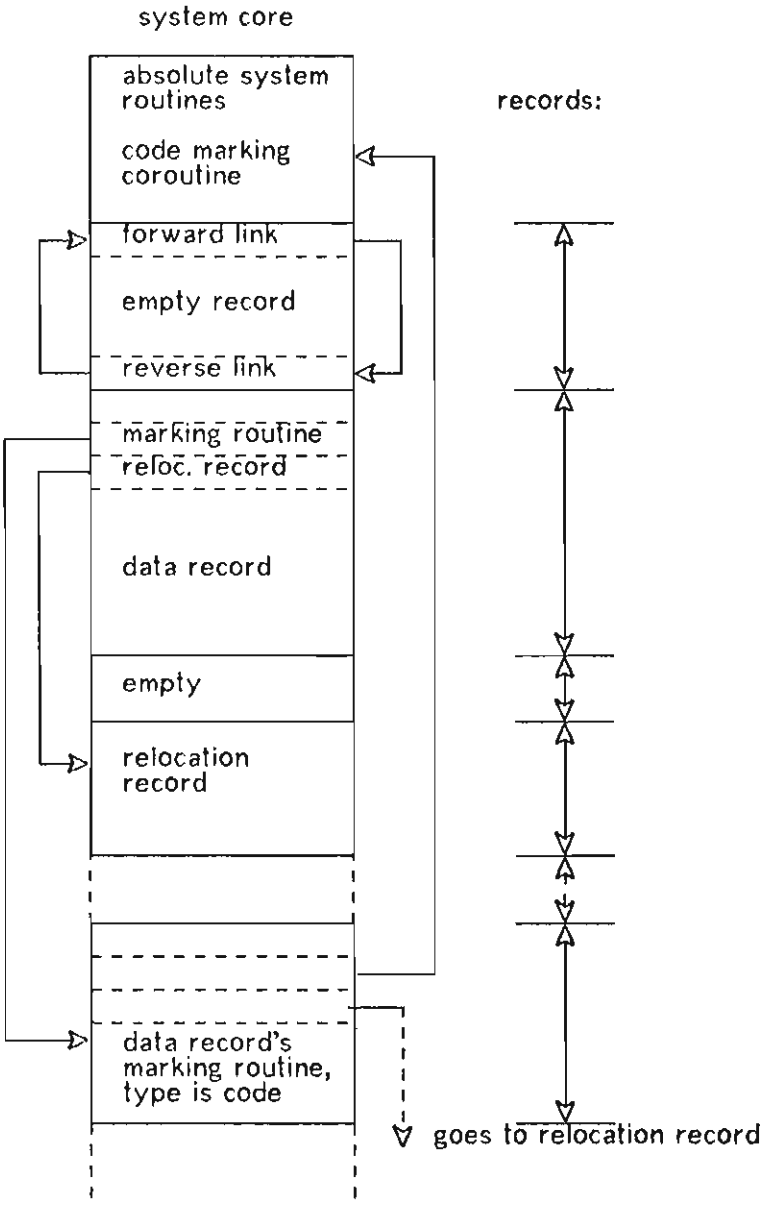

Figure 2-9: Example of record structure.

2.3.5 Other Features

Since relocation information and symbols are retained with all the code records, the user can have the system display the machine code in almost assembly language. This can be facilitated by including in the relocation record information such as whether a non-relocatable word is an instruction or data, and what type of data, integer, floating point, etc. It was previously pointed out that words could be inserted and deleted in the middle of a code record by appropriate action of the garbage collector, therefore a complete language system could be made. The easiest language system would be assembler language, but by incorporating more source information into the records, other languages could be implemented.

This implies that the system will do interactive input and output with the user, and perhaps access secondary storage. The philosophy here is that the most global system should have total control over whatever it needs, and no control over what can be performed more locally. It would be disastrous if user programs tied up I/O to the extent that the system could not communicate an error condition to the user. Therefore I/O should be a global system function. In fact, we go further to suggest that the system manage all the user dialog, that is prompt for commands, implement a command dispatcher for system and user defined commands, process interrupts from the terminal, etc.

3. Functional Simulation

In the past digital discrete event simulators have been of two types with respect to how the simulation elements are viewed. One type, typically called logic simulators, looks mainly at the outside of the elements, where it is connected to other elements. The elements generally are supplied by the factory, and are simple primitives. The user can simulate complex devices by representing them as interconnections of many primitives. The other type of simulators, called functional simulators, view the simulation elements only from the inside, where its internal manipulations are described. They allow the user to simulate complex devices by segmenting them into subprocesses and procedures, which may themselves be very long. These two types of simulators have been mutually exclusive. This chapter presents a way of merging these two types of simulators.

Elements in this simulator are viewed from both the outside and inside. In fact the simulator has three fronts, or angles from which it can be viewed. One of these is the manner in which the simulation is controlled and monitored, and is a front of all simulators. The others are the external, interconnection and internal, functional views of the elements. The last two are present, one each, in the original types of simulators. Large devices can now be simulated by both making elements of high complexity, and connecting as many together as desired.

This system has an analogy in the operating system of a timesharing computer. Such an operating system is intended to be used in two entirely different ways. One is the human interaction from the terminal, the other is the support of user programs. Such an operating system will typically have two manuals, one for user commands, and one for operating system procedures. This simulation system will similarly have an interactive front, with commands activated by humans, and an internal front, where user defined computer code interacts with system procedures. The internal part is even an extension of the internal front of the operating system analogy, here the user can define his own system procedures. This is because it is recognized that some higher level applications will find sufficient and convenient the provided set of system procedures, whereas other applications will need some

esoteric user defined procedures in addition. At one level the system supports discrete event simulation of elements that are generally like digital computer elements. The user may define the way particular elements interact and how they work. At a lower level the user could perhaps define a circuit simulator, and simulate continuous systems together with discrete systems. At a higher level system procedures are provided that are like those required by high level functional description languages.

The lowest level of this system is a discrete event simulation scheduler. This is a piece of code that control is passed to when simulation is invoked. It will start and stop schedulable processes in accordance with simulation time. These processes include the descriptions of simulation elements, an interactive simulation control process, and whatever processes the user cares to define.

Above this level there is a formalism for the creating, delivering, and receiving of messages, the external events of simulation. This formalism allows a computer, capable of doing only do one thing at a time, to handle messages, which fundamentally involve more than one element at a time. There are some fundamental forms of messages which commonly appear in simulation, and these are supported as part of the system. The user may define his own message types and delivery mechanisms, and, with regards to compatibility, implement them. There is also a formalism for checking message compatibility.

The highest level involves the organization of the functional descriptions that describe the simulation elements. The purpose of functional description languages is just this, but here we include the philosophy of how to operate in an environment with other elements. Using these guidelines functional descriptions of elements normally appearing in logic simulators and elements normally described functionally have been implemented. The user is not required to follow these guidelines, he may implement arbitrary message types.

This chapter is concerned strictly with the issues directly relevant to simulation, and refrains from discussion of supporting systems. This simulation system requires

that there be an interconnect system to generate, in core, a number of data structures to represent the simulation elements, interconnected by pointers. In simulation all that is needed is to know what connects to what, whereas a great deal more information may be desired for non-simulation purposes. The interconnect system will also need a facility for incorporating selected pieces of code, representing the simulation models, into a runnable program. It is believed that the material in this chapter can be applied independently of the details of any interconnection system, or in situations where a system like the interconnection system previously described is not present.

The issues of this simulator can be logically addressed by examining the features of the logic and functional simulators from which it was developed. The overwhelming features of logic simulators are the simulation elements, and their connections to other elements. These are the first issues discussed. This part is quite simple in logic simulators, and very rich in this simulator. In fact, the details and side issues of the basic simulation elements extend through the rest of this chapter. From logic simulators the issue of inter-element communication can be addressed. The message types of logic simulators are expanded upon in this portion of the discussion. Simulation elements from functional description languages follow fairly closely the view of simulation elements so far developed, with some additions. These additions concern mainly the presence of subdevices in a simulation model. These are just software techniques for making the descriptions more structured, but they require support from the simulation system. We then explore the issues of actually making functional descriptions by following examples of increasing capability. Finally some side issues, previously set aside, are discussed.

3.1 The Element

The interconnection system provides its own element data structure for each simulation model. The parts of this data structure that are relevant to simulation are the ports and their connections to other ports on other elements, and the pointer to the instance of the simulation model. A simulation model has an internal state part

and a description part. The internal state part consists of explicitly defined variables, and implicitly, or system defined, variables. The explicitly defined variables are the flags, registers, or arrays needed to conveniently describe the function of the device. The implicit variables are the simulation time for the next activation, address of the next machine instruction to be executed, the stack, etc. The internal variables are not strictly internal, they can be accessed interactively by the user, or by the element's subdevices.

The description is a sequence of machine instructions that implement the function of the element. A particular sequence of machine instructions need not be unique to a particular element, but may be shared by several elements of the same type. During simulation an element's internal state will change, and it will send a sequence of messages out over its ports. The sequence of messages, each of which exists only for a discrete value of simulation time, forms the external state of the device. Since this is a discrete event simulator it is necessary that every change in internal or external state of some element be caused by either a pre-arranged delay elapsing in, or a message being received on one of the ports of that element, or one of its subdevices. By this definition the external state and some of the internal state can be made an arbitrary casual function of all the inputs to a device. Figure 3-1 depicts the structure of an element with a simulation model.

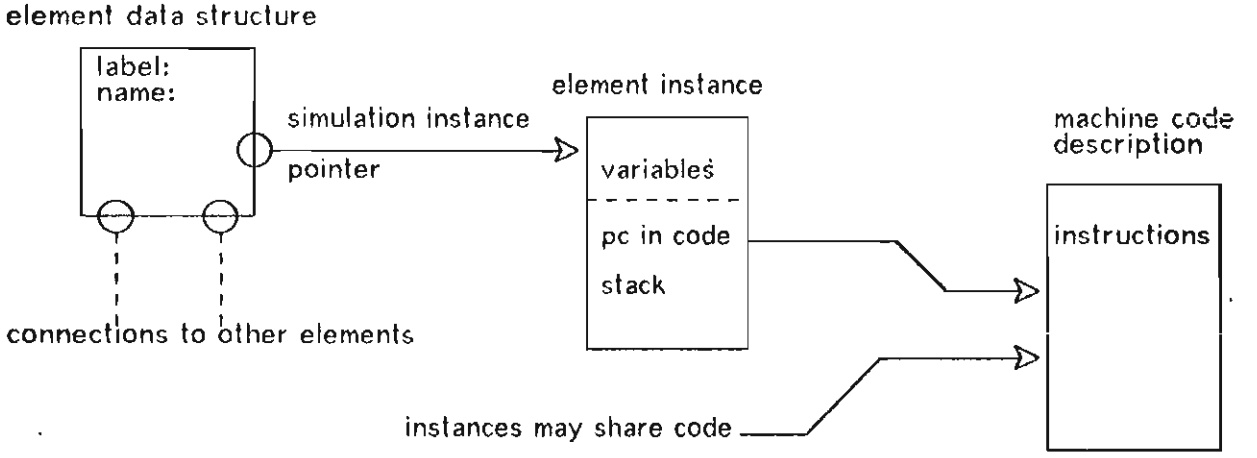

Figure 3-1: Pointer structure of an element with a simulation model.

3.1.1 Types of Elements

There are several different classes of elements. One of these represents the regular simulation elements. Regular elements may be system defined primitives, such as logic gates and arithmetic elements, or may be user defined, such as functional descriptions of new processors. Another class is those elements the system has special interest in. In this class are various types of wires. In this simulator the wire connecting several elements is itself an element. Each of the ports on the several other elements connects to the wire. The wire therefore has a simulation model, that usually is just to transparently echo the messages from one port to all the others. Similarly, the internal state of a wire is the value on the wire. As was mentioned previously, wires are specially known to the simulation system, i.e. they can be automatically generated, and the simulation implementation is optimized due to their simple and well known behavior.

3.2 Message System

A two phased approach is used to implement the message delivery system. The two phased approach is used simply so that messages can be delivered in contiguous simulation time. The phases are implemented in the description code by the process part and the input chaining part. The process part is responsible for generating all the messages. The input chaining part receives all the messages. The process part, a schedulable coroutine containing the bulk of the functional description, has control over simulation time. All changes in external state are due to this part. The input chaining routine has the characteristics of a procedure, although it may be implemented as a coroutine. This routine serves no other purpose than to receive all the messages from the other elements and relay them, with some processing, to the process part. This part is restricted in capabilities, all that it may do is manipulate internal state and arrange the delay for its process. It may not alter simulation time.

This is sufficient to implement an arbitrary behavior. It was stated that every

change in state is caused by either a delay elapsing, or a message being received. Changes due to delay elapsing are simply done by the process part, as it is activated at the proper simulation time. Since delay time is part of internal state it may be set by either part. Changes caused by messages being received are implemented by the input chaining part directly, if the change is internal. External changes are implemented by the input chaining routine setting a zero time delay for the process part, and the process part will then implement the external change at the same real time.

3.2.1 Scenario of Operation

The general scenario for the operation of the two phased system is as follows: When a message is delivered the input chaining routine is activated. The internal state will be changed as necessary to reflect that the message was received. The input chaining part then examines the internal state to determine the earliest future time that another change in state could occur. Then a delay for the process part is set. When the process part is activated it examines the internal state and makes the appropriate changes to the internal and external state. The process part then determines whether another delay is needed, and sets its delay accordingly.

3.2.2 Third Phase of Message System

In many cases an element will have no use for the messages delivered to certain of its ports. To improve efficiency in these cases a third phase exists. Its function is to mask any messages to some ports. This phase consists of an array of flags, one corresponding to each port. Whenever a message comes to the element the proper flag is tested, and if set the input chaining routine is not activated. The total picture of message delivery can now be summarized in a slightly different way. When a message is sent to a port processing occurs at up to three levels, depending on the predicted complexity of the response. If the response is simply to ignore the message then processing stops after checking the mask flag. Some responses require activation of the chaining routine, and responses actually

requiring change in external state eventually activate both the chaining routine and the process part. A desirable result of this is high efficiency, the simplest and quickest action possible is taken for each message.

A message is delivered by a system procedure activating the input chaining routine in an element. Information can be passed in any convenient way. Since message delivery is not recursive the information can be left in globally accessible locations. A message delivery system is implemented by code which follows pointers from an element's port to connected ports, managing the additional information, and activating the input chaining routine. There are several message types predefined by the simulator, and the user may implement others.

3.2.3 Implemented Message Forms

The several types of message forms implemented by the system are believed to be the most common. Two of these are associated with the concept of connection by wire. The natural function of a wire is to transparently distribute any message from any connection to all other connections, although more complex schemes may be implemented. The wire has an internal state, which is the value on the wire, of which two types are recognized. One may have a value of only one or zero, i.e. a bit, the other uses an integer variable as its value. The integer is intended to represent a group of bit type wires, in number up to the word size of the computer. The third message form does not utilize wire connection, but applies only when two ports are connected directly to each other. Messages of this type are merely flags, and have no additional information. There are system procedures that can be called by a process part to generate these messages. Figure 3-2 illustrates the various message types.

Consider now some examples of message types the user may want to implement. The wire method of connection is limited here to two types, bits and integers. Larger numbers of bits may need to be implemented for some applications, necessitating two integers, or possibly an array to hold the value. Interconnections to the backplane of a computer have some characteristics of wires, but the

Type	Information	Comments
Bit	true/false	The basic bit type wire.
Integer	machine's integer	Group of several bits, a bus.
Direct	none	A flag that something is happening at that time.

Figure 3-2: Summary of message types

information transfered is many bits in irregular groups. This could be implemented by a special wire type for a particular backplane configuration. An application has been suggested where a tree type priority structure needs to be simulated. Such a structure can be viewed as an extension of a wire interconnection. A user could implement a delivery procedure that would deliver a message presented to one part of the structure only to selected other parts.

3.3 Subdevices

In functional simulation it is common for an element to have subdevices, and for these subdevices to have access to the internal and external state of the main element. A subdevice here means a description which can be executing during overlapping simulation time with its main element. A subdevice is like a separate element, except that it can manipulate the state of its main element. When subdevices are implemented they are as separate elements connected by special, direct, connections. The pointers of the direct connection can be followed to give access to the state of the main element. Accessing the internal state of other elements can be desirable, but side effects may occur if not approached with caution. For example, if the value on a wire is arbitrarily changed without calling the system wire change procedure then the elements connected to the wire will not receive a message informing them of the change, and the simulation could proceed incorrectly. The rules for accessing are extended to allow the subdevices of an element, of subdevices of subdevices, etc. to read or alter the internal state, or send a message from one of the ports of a particular element. If this rule is

followed functional subdevices get the desired access whereas other devices are secure merely by not having subdevices.

3.4 Organization of Descriptions

The primitives outlined above leave the user much freedom in how he will implement functional descriptions. In the case of large formal functional descriptions compiled from a functional description language, the user will be well advised to use a readable, expandable, and well structured method for implementation. It is realized, however that the user must have the ability to generate unstructured elements which may be especially efficient or flexible. For this reason the system does not restrict the user in defining arbitrary elements. There are guidelines for how to organize functional descriptions in a reliable way, however. In presenting these guidelines we will start with descriptions of the simplest form and demonstrate how more advanced features can be added.

3.4.1 Minimum Implementation

The simplest form of a functional description is just a computer program. This would have no procedures, access to external states, or time dependencies. Such a description would describe a machine that simply transforms its internal state. The internal state can be monitored and would serve as the output. The implementation of this is trivial, the computer program simply becomes the process part of the description. This part is activated automatically at zero simulation time, and runs to completion. Completion here refers to a call to the simulation scheduler to indicate that it may proceed with the next schedulable process.

3.4.2 Basic Time Dependency

Time dependency can be implemented by including calls to the simulation scheduler that reschedule the process part after a delay. Consider, for example an element that changes its internal state periodically by, say, incrementing a register. This could be implemented by writing a program, as before, but including in it a loop

that calls the delay procedure and increments the register. The device could be monitored during simulation and its register would be observed to increment.

Calling the delay procedure within the element can be viewed as adding to the simulation time variable, independently of the rest of the simulation.

3.4.3 External State

The ability to manipulate external state is fundamental to any useful simulation. Consider first an element connected only by wires. The element may change the value on the wire by calling the system wire change procedure. It will specify the port the wire connects to, and the new contents. For example an oscillator is made by making a loop which delays and outputs alternating values. Input from the wires can be implemented by making calls to the system wire read procedures. These procedures return the value on a wire given the port it connects to. The oscillator could also be implemented by making a loop that delays and outputs the complement of the input from the wire.

3.4.4 Use of the Chaining Routine

Previous examples have not made use of messages received from other elements. To do so requires that the input chaining routines be used. In functional description languages the primitives that would do this wait until a particular value appears on a wire. A trivial implementation of this is for the process part to enable the mask flag only on the port of interest. The desired value for the wire would be left in a convenient place in the internal state. The input chaining routine would then be activated only when the selected port receives a message. The routine would be programmed to test the value on the wire. It would activate the process part, with zero delay, if the value was the same as the given value. The process part would then set all the mask flags to prevent future interference. All this could be implemented as a procedure called in the process part. A programmer should immediately see that this can be extended to provide arbitrary programmed response to inputs.

While programmed responses are sufficient in general to produce arbitrary behavior, they may not be convenient. An example of this is an element having one input of more immediate interest than the others. An asynchronous reset input to an element is illustrative. Such an input should produce an immediate response regardless of whether the process is waiting on input or not. The reset function could be implemented directly in the input chaining routine. The reset port would, of course, never be masked, and the routine would do a special reset function whenever that particular port received the reset message. This function could include restarting the process part. Another example is a logic element where the processing of programmed responses to inputs would be a large part of the description. In such a case it may be desirable for the input chaining routine to actually do processing. These methods should be used sparingly, as they make descriptions less structured. They may also lead to problems. If the element is connected to itself, care must be taken that the chaining routine doesn't interfere with the process part.

3.4.5 Subdevices

Some functional languages[1, 7] allow the user to create subdevices. The special, direct, connection allows messages to be sent to control the subdevices, and for the subdevices to access their main element. This system provides procedures to send an informationless messages over these connections. The input chaining routines must be set up to handle this type of message, however. By convention a message from a main element to a subdevice will start the subdevice, and the reverse message will be readable from a programmed wait. Internal access is provided by a system procedure that returns a pointer the element directly connected to a particular port. An important attribute of subdevices is whether they are running. Subdevices usually carry out some sequence actions for a time after they are started, but eventually terminate. They should only be started when not running. Through a system procedure a main element can determine if a subdevice is running, and by convention when a subdevice changes from running to not running

it sends a message to its superior element. There is a system procedure, using these, which implements a programmed wait until a subdevice has finished running.

Experience has indicated that while most elements can be described in any of the ways above, one way is often vastly superior. Some elements are much more easily implemented with a special reset function, whereas others may be satisfied with using programmed input response. Some elements are implemented in a structured fashion with subdevices but are difficult to implement without. And then some devices may require esoteric, user defined techniques.

3.5 Interaction

Since the user has much flexibility in the implementation of simulation models, a single method of interacting with them is inadequate. Instead, the user provides each simulation model with a part to do the interaction for that element. This allows an arbitrary interpretation of interaction. The interactive part of a simulation model is a function of the input chaining routine. It is activated by a message being received over a special port, called the interactive port. This port is usually unconnected by the interconnection system, and hence would receive no messages during simulation. These are instead activated by commands entered interactively by the user. The simulation system will fake a message by activating the input chaining routine with the interactive port specified. The interactive part may then do output, or even interact with the user.

Wires have a slightly different interpretation of their interactive part. The interactive port on a wire is actually the port that other devices connect to. Whenever the value on the wire changes this port receives the message. Since the system wire change procedures do all the necessary processing, the input chaining part on wires is not necessary for simulation. It usually notifies the user, by printing a line on the terminal, that the value on the wire has changed. This is the trace feature. The mask flag has an effect on the interactive port. In the case of a wire, if the port is masked then the interactive routine is not called. There is an interactive command for setting and resetting this flag, which is the user's control

of tracing. Non-wire devices may use this flag to indicate the user's desire for tracing, and if set when selected internal changes occur, inform the user of the changes. If the internal state of a class of elements is similar, as would be of those from a common functional language, a system procedure could be written that allows general interactive examination and altering of internal variables.

The simulation system's main command interpreter is also a schedulable process to the simulation scheduler. This process is the one that runs initially. When simulation elements are setup they are also scheduled. To start the simulation the user issues a command that instructs the main process to increment simulation time, scheduling itself for some later time. Simulation will occur until that time is reached the main process is continued. Commands can then be issued to examine the simulation while in progress. Naturally, the main process can repeatedly start the simulation, as desired by the user. If a condition occurs in an element which should stop the simulation, that element can continue the main process immediately.

3.6 Setup

The interconnection system provides an interconnection of its element data structures, in which initially the only information about how an element will simulate is given by names associated with the elements. These names match with names of the descriptions. At some point it is necessary to read this name and follow the setup instructions for the named description. The setup instructions are a part of the functional description written by the user, and hence there is a great deal of flexibility. For example, a memory is a common simulation element. In general, however the user will want to specify the initial contents of the memory. Instead of requiring an entirely different description for memories with different initial contents, the setup code reads a file containing the initial contents. Another function performed during setup is compatibility checking. An interconnection system would allow elements to be connected that might try to use connections in different ways, say connecting a device expecting a bit type wire to one expecting an integer wire. There are system procedures that check connections, which should be called in the

setup phase.

The parameter feature of the interconnection system is also used during the setup phase. The parameter feature allows small amounts of information, called parameters, to be associated with each element. The parameters are grouped into parameter sets, each of which has a set name. Generally there will be more sets present than are of interest to any particular function, hence only those with the appropriate set names will be used. The setup phase may call system routines which fetch parameters from specified sets. These are used, for example to set propagation delay time, or in the case of the memory discussed above, specify a file name for its initial contents.

4. Implementation of the Interconnect System

An interconnection system was implemented along the lines developed in chapter 2. Not all of the features described were implemented, but the resulting system was able to demonstrate the usefulness of interconnect systems. Specifically, PDP-10 SIMULA[6] was chosen as the implementation language. Since SIMULA does not have a facility for making code data structures during run time and executing them, not all of the features of the software support system are present. The important capabilities of the software system are available, but in a less convenient form. There is a preprocessor for SIMULA code that can add and delete modules from the program, but requires that the compiler be run each time. As mentioned previously, there was no macro facility implemented. The macro facility was defined by examining the working implementation of the interconnect system and observing that such a facility could be easily implemented and would be very useful. In this system there is a single group of elements, all at the same level, that form the design. Interestingly the absence of a macro facility has minimal effect upon the demonstration of design functions, since macros are intentionally transparent to them.

This chapter will follow the reverse of the order used earlier to develop the interconnection system. We will discuss first the fundamental implementation issues, such as the preprocessor, and how the code modules are organized. Utility functions of the interconnect system are then discussed in the order of increasing applicability to CAD. First such features as the I/O handling and command dispatcher are mentioned. This leads to a discussion of the database manager. The interconnection editor is discussed, and the particular selection of commands is described. The most application specific portion of the interconnect system is the actual structure of the interconnect elements, which is discussed last.

4.1 P sorter

One of the features of the interconnect system is its modularity with respect to the design functions. This means ideally that there should be separate files with the code for the interconnect system and each design function. When the system is run only the code modules actually needed will be in core. To implement this we have a program, called P. P reads files containing the SIMULA code for the interconnect system and the selected portions of the design functions. This program will sort these files into a compilable SIMULA program. The program can also supervise the compilation and loading of the complete program. The P program divides a SIMULA program into a number of parts. The input files contain control lines that specify which part code lines belong to. The lines are then sorted into the selected part in the order they are read. In practice parts often have a beginning section, with BEGIN statements, declarations, etc. and an ending section with ENDs, etc. To support this we allow the control file, which is discussed in detail later, to have these beginning and ending sections. We will describe the input format to the P program to give the reader a feeling for the source format before discussing the actual parts of the program.

The P program reads source files that have three kinds of lines. The first of these line types specifies which part the source lines following should be directed to. The second type is a source line, which is sorted into the selected part. The third type is a comment line. Comments are only valuable in the original source file, and not in the sorted file that is input to the compiler. We allow comment lines, therefore, that are discarded by the preprocessor. The formats of these lines is shown in figure 4-1.

Format of Line	Type	Comments
##PART## name	control line	The following lines go into part name.
line without #	source line	The line is transfered to the proper part.
#rest of line	comment line	Line discarded.

Figure 4-1: Input Formats to the P Program.

There is a control file that specifies all of the parts for a run, the beginning contents of each part, the files for source input, and the ending contents of each part. The execution of the P program is simple. It sets up files for the appropriate parts after reading the first line of the control file. It then takes source lines from the control file to initialize the parts. Input is then taken from each of the files listed in the control file, to do the bulk of the sorting. The control file becomes the input again to put the ends on each of the parts. The control file format is pictured in figure 4-2.

File is called SECT.CMD

partname1,partname2...,partnameN;	*Part names for the run.*
...lines...	*Input lines to start the parts.*
/*	*To end input lines.*
file1,file2,file3...,fileM;	*Source files. No extension.*
...lines...	*Input lines to end the parts.*
/*	*To end input lines.*

Figure 4-2: Control File Format.

The parts used to make the entire SIMULA program are listed in figure 4-3. Most of these correspond to the syntactic sections of a SIMULA program.

All of the code, including the main interconnect program, is structured as allowed by the preprocessing. Code is written with, for example, a class declaration and initialization contiguous in the source file. There must, however be a formalism for including code modules and all references to it in a module, to allow modules to be selectively added and deleted. To do this we have a special CLASS in the main program, corresponding to part SETUP, containing the generating code for the modules that can be added. This CLASS, used as a coroutine, compares a global variable with the name of each added code module, and when a match occurs it generates an instance of the module's CLASS. The format for modules that can be

freely included is as follows: There can be an arbitrary amount of code going into the various parts, but it must not be refered to by any code outside the module. There are then a few lines of code, which go into the SETUP part, and serve as the reference of the rest of the program to that module.

1. CLASS. This has the CLASSes of the main program. Each entry in this part is a complete CLASS declaration, hence there are no order restrictions in this part. This part has a beginning section, that includes initial BEGIN statements, prefix CLASSes, etc.

2. SETUP. This is the CLASS used as a coroutine to generate instances of other CLASSes that may be included in the program. Each CLASS contributes a part of the SIMULA case statement that compares a global variable with its name, and generates an instance if there is a match. This class must have a few lines at the beginning and ending besides the big case statement. These are in the control file so that this part is order independent.

3. PROC. This part has globally accessible procedures. It is like the CLASS part.

4. DECL. This part has global variable declarations.

5. INIT. This part becomes the first executable statements of the main program. These are intended to be order independent and are the first statements executed.

6. MAIN. This part has the main program. The main program usually comes from one contiguous source file. This is order dependent, and cannot generally be added to.

Figure 4-3: Parts of a SIMULA program

The sorting program performs other functions. On the machine that the programs were developed on it uses the TMPDSK facility to invoke the SIMULA compiler, delete temporary files, loads the program, and start execution. These features are very installation dependent and will almost certainly require modification to run on other machines.

4.2 Utilities

The interconnect system has a series of procedures which do the various functions that need to be performed centrally. The interconnect system itself uses these system procedures, and they are also available to user programs. These procedures generally perform the interface between the program and some resource, like I/O or the monitor. The functions are input/output control, including a command dispatcher for the interactive terminal, error and debugging control, and manipulation of the program's core.

The I/O facilities extend somewhat over the normal I/O available to a SIMULA program. The system maintains several input and output channels, including the terminal. It defines a selected input and selected output channel that other parts of the program should use. By changing the selected channel from terminal to a disk file it is possible to effectively log the terminal output, or accept terminal commands from a disk file.

The system uses this I/O facility to implement a command dispatcher. This is a large loop which prompts for a command, and interprets it in a giant case statement. A feature is that the input is first run through a unique substring routine to determine if the command is an abbreviation of a longer command. The system gets the actual commands at initialization time from a file. The user can add commands by inserting his command into the file, and adding another branch to the case statement.

Some of these commands are interpreted by the system. We include a description of these here.

- TRACE. This toggles the trace flag on the input parser. This is useful for locating errors on input.

- -. Minus is not really a command, as there is no prompt after its execution. It sets an invert flag so that the next command will have its normal function reversed. For example, the PROBE command will start tracing, but will stop tracing if preceded be a minus, e.g. -PROBE.

- FREEZE. This command causes the entire program to be readied for a save to a disk file. It closes terminal files, releases the high segment, saves the accumulators, etc. This can be used, for example, to save a large simulation while in progress.

- SIMDDT. For serious debugging the user may want to use DDT. This command calls the SIMULA DDT. The user can then examine storage, set breakpoints, etc., then continue execution.

- EXIT. Exits to the monitor. A lengthy version of ↑C.

4.2.1 Variable Names and Parameter Passing

To facilitate the structured expansion of the program a formalism has been developed for the naming of global variables and routines. The object of this is to assure the user that a section of code included in the main program will not attempt to use the same identifiers as another section of code. We hope to prevent compilation errors, or worse, forming incorrect programs. The formalism developed also aids in the readability of the code.

All sections of the main program are divided into modules. Each module may contain one or more coroutines, procedures, or parts of the main program body. Each of the modules is, if necessary, given a name of exactly three characters. Some modules, however will not have any global variables, etc. and hence will not need a name. Within these modules the user is free to define any local variables he likes. Global variable declarations, however have to follow a particular format with respect to the names. Every global variable, of this type, is formed from seven characters: a three character local name, the symbol "@", and the module name. If the user chooses a unique module name he is assured that there will be no variable clashes.

There is also a formalism for making coroutines. In SIMULA a coroutine is a CLASS which makes use of a Detach statement. When a CLASS executes the Detach its state is saved in the CLASS instance and execution is continued from where the CLASS was generated or Called. Each coroutine has a single global variable that is

initialized to an instance of the CLASS. Coroutine activations are then done by Calling the global variable. This formalism requires that there be statements in the main program in a number of places. There is the CLASS, global variable declaration, initialization, and the various calls. The user is free to choose the CLASS identifier of the coroutine. The other type of global variable name, which is the name of the global coroutine variable, is formed by the symbol "$" followed by the CLASS identifier. When reading the program and there is a Call to a variable prefixed by "$", it is a call to a coroutine.

We have described the two ways of naming global variables. Every variable declaration now in the main program is of one of these two types. The user is encouraged, in the interest of program structure, to adhere to this formalism. We further recommend that the characters "@" and "$" not be used elsewhere in the source code.

SIMULA does not support parameter passing to coroutines. It is therefore necessary to define a manner for doing this. Parameters are passed, and values returned, by leaving them in globally accessible locations when the Call or Detach is executed. The variables are normally set by an assignment statement. When reading a program and there are a number of assignment statements to variables of the same module followed by a coroutine call to that module, the user can expect that parameters are being passed.

An example of the appearance of a coroutine in the source code appears in figure 4-4.

4.3 Database

The implementation of the database is considerably simplified here by the absence of a macro feature. Here, where there is a single design, the problems of having a multilevel structure, multiple instances of the same element, etc. disappear. It is still necessary, however to be able to load a description from the database and return it after changes have been made. We implement this by having files

Generating the coroutine.

```
##PART## CLASS
CLASS example;
        BEGIN
        WHILE TRUE DO
                BEGIN
                DETACH;
                ...body of coroutine...
                END;
        END;
##PART## DECL
INTEGER DAT@XMP;
REF(example)$example;
##PART## INIT
$example:-NEW example;
```

Calling the coroutine.

```
DAT@XMP:-4;
CALL($example);
```

Figure 4-4: Example of a Coroutine

containing the descriptions of a particular design. The contents of the files is only the inferior element part of the description, since none of the others apply. The database manager, such as it is, reads these disk files into the interconnection structure and writes the entire design back out. The database commands are listed below:

- GET filename. Read the written representation of an interconnect structure from DSK:filename.IL. This command can actually edit the design. If it is used when a design already exists, if will join the two interconnection structures.

- DUMP filename. Write the written representation of the entire design to DSK:filename.IL.

4.4 Editor

A great deal of flexibility was left in chapter 2 in the implementation of the interconnection editor. We have chosen a few commands which are useful in many editing tasks. There are a couple commands for examining the existing interconnect structure, and a very flexible command for adding to the structure. These commands and their functions are listed below:

- ALL. This displays for the user the entire written representation of the interconnect structure.

- ELEMENTS. All of the elements are listed, giving their label and name. This is different from a written representation as the connections are not shown, but elements such as wires, not otherwise shown, appear.

- ADD line ;. This command adds the line to the structure. It is flexible in that the line may refer to an element that already exists. If the element exists then undefined parts of the element will be defined by the line. Using this the user can, for example, add parameters, connect ports, or even add more ports to an existing element.

4.5 Parameters

We digress slightly to discuss the parameter feature. Parameters are implemented just as described in chapter 2. The names of the parameter sets, and the parameters themselves are text. The details of the pointers are not important to the user beyond this point since there are system routines for manipulating parameters. Specifically there is a procedure, called FETCHPARM which will get parameters in an arbitrary way. This procedure is called specifying the name of the parameter set desired, and the format of the parameters in that set. The procedure will find the parameters, if they are present, and load them into the next available positions in an array. If the procedure is called several times the array continues to

fill up. If it is desired to have default values for the parameters, values can be loaded into the array before the procedure is called, and they will remain unchanged if the parameters don't exist. In this way numerous parameters from different parameter sets can be accessed, with defaults for the parameters in each set. After the parameters are obtained, and used, there is a procedure, called FINPARM, which will reset the array back to an empty state for the next parameter fetching session.

The format of the parameters within a set is supplied in a data word. The fields within this word specify the number of parameters expected in that set, and the type of each. Presently parameters may be integers, reals, text identifiers, or don't cares. Although the parameters are stored as text they are translated to the expected form when fetched. If the fetching routine finds the format of the actual set is different from what was specified, an error occurs.

4.6 Syntax of the Written Representation

We include here, for reference, some details about the written representation of the interconnect structure. A description of a design is composed of a number of lines separated by semicolons, each describing an element. The lines, although not restricted to fit on one line, often do. This format is used in the database files, and in output to the terminal. The format is illustrated in figure 4-5.

line ;	*Regular input line.*
line	
..... ;	*Long line.*
line ;	*Semicolon required after last line.*

Figure 4-5: Format of Database Files

Within each line there is a description of one element. This includes its label, name, parameters, and connections to other elements. The general form is shown in figure 4-6.

```
label:name/set1(parm1,parm2,parm3)/set2(parms) Lab1 Lab2 Lab3.5 - ;
```

Figure 4-6: Format of a Line

A discussion of each part is in order:

- LABEL. This part is delimited by the colon. The part is optional, but if omitted it is hard to refer to the element later.

- NAME. In some form this part always exists. The null name is acceptable, which is represented by the symbol "-". Wires always have null names.

- PARAMETERS. There may be an arbitrary number of parameter sets, including zero. There may be more then one set with the same name, but the interpretation of this is unknown. The setname must be an identifier, but the parameters may be identifiers, integers, or floating point numbers.

- CONNECTIONS. The number of ports on the element will be the number of entries in this part. Each part makes a connection to the port corresponding to its position, starting with port one. Each entry can be either empty, direct, or wire. Empty connections have a "-". Direct connections have the element's label and port number, separated by a period. Wire connections have the wire's label, connecting to port zero by default.

- SEMICOLON. The semicolon is not part of the line, but must be present as a delimiter.

4.7 Interconnection Structure

The implemented interconnection structure follows that described in chapter 2, using the option where wires have a single port. We show in figure 4-7 the actual SIMULA declarations for CLASS element. Figure 4-8 is a diagrammatic interpretation of the element. Some additional interpretation is in order, however. In chapter 2 we refered to an element's ports, and said that the ports connected together. The reader will not notice any obvious ports in CLASS element. The implementation is

slightly obscure. In chapter 2 we required that each port have two pointers, one to the wire it connects to, and the other to the next element in the ring of elements pointing to the same wire. We are also using integers to name the ports. This allows all ports to be implemented as arrays, each individual port being the entries in all the arrays with index equal to its port number. The pointer to the wire is in array TOP. The pointer to the next port is in the two arrays POR and LNK. These two arrays specify an element and the port number of the selected port in that element. We refer to the two variables, one REF(element), and the other INTEGER as a complex pointer. Both pieces of the complex pointer are required to make sense.

```
CLASS element(pts);
INTEGER pts;              The ports are numbered from zero to pts.
    BEGIN
    INTEGER ARRAY por[0:pts];
                          Port number part of the complex pointer.
    TEXT lab,nam;         These are the label and name of the element.
    REF(chain)chn;        Points the the instance of the description.
    REF(element)nex;      This is the chain of elements in the design.
    REF(element)ARRAY lnk[0:pts],top[0:pts];
                          lnk is part of a complex pointer, top points to wire.
    REF(set)hds,tls;      These point to the parameter sets.
    END;
```

Figure 4-7: CLASS Element.

4.8 Support of Design Functions

The interconnection system can be run independently of any design function. This would happen if it were only desired to edit a design. Design functions normally associate additional information with the element data structures, and there must be a way of generating instances of these. Since checking is performed when these instances are generated they should not be generated when the elements

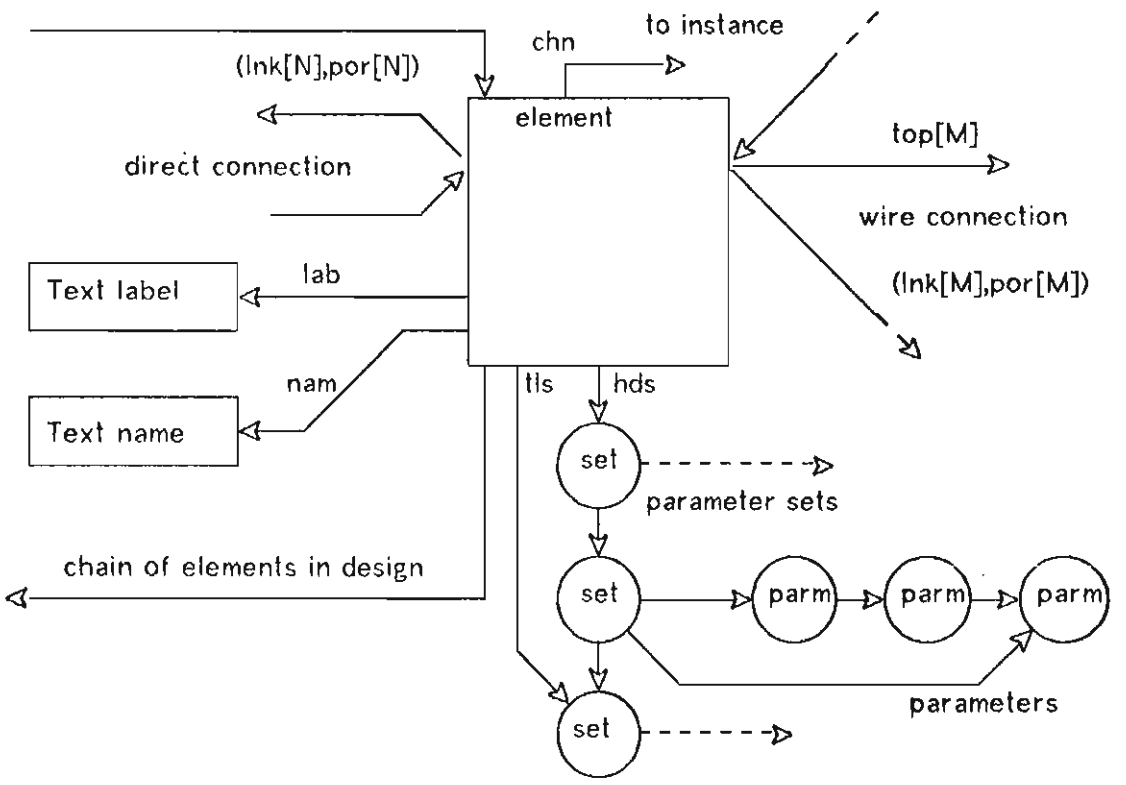

Figure 4-8: Diagram of CLASS Element.

themselves are created, as this would prevent correction of errors. We instead suggest that each design function have a command that sets up the models for its function. This simulator has such a command. This command examines each element in the design and calls the coroutine of the SETUP part. The simulation models included in the program have each inserted code in this CLASS which does the setup function for that type of element. The command is listed below:

- SIMULATE. This generates a simulation model for each element in the design that doesn't have one. This command causes checking of ports and parameters, hence may cause errors to be revealed.

5. Implementation of a Functional Simulator

A simulator was implemented that follows closely the outline presented in chapter 3. A general discrete event simulator was implemented, as well as a number of simulation elements ranging in complexity from a nand gate to a microprocessor. These simulation elements are in several files, which can be included in the system according to the formalism developed in the last chapter. In discussing this implementation we will follow essentially the same outline used in developing the theory of the simulator. The actual structure of the simulation element will be discussed first. This will be a simple extension of the element data structure described in the last chapter. Here we describe what the instance pointer called chn actually points to. Following this will be a description of how messages are delivered. There will be a brief description of how the system defined message types do message delivery. Users who will implement their own message types will find this particularly useful. We continue along this line with a description of the various system defined message routines, and list them and their function. The issues called side issues in chapter 3 become very important in this chapter. The first of these is the interactive facility, and this extends to cover the system commands for controlling the simulation. The final discussion is about the setup phase of the simulation, where the simulation models are invoked, and where various checking functions are performed.

5.1 Structure of Simulation Elements

We present here the structured approach to making simulation models. The description in chapter 3 of the simulation elements required that they have only a few properties, not including the property of structure. Since this implementation is in SIMULA, the user can make descriptions which are extremely complex and unstructured. Since it was shown that the structured approach is sufficient in general we will assume that all implementations will follow the proposed structure. Figure 5-1 shows the simple, but sufficient structure to implement a simulation model. Figure 5-2 shows the graphical interpretation of this.

```
CLASS chain;            General prefix class.
    BEGIN
    REF(Process)pro;    Pointer to the process part.
    INTEGER msk;        Masking flags for this element.
    END;
chain CLASS Cmodel(mom);
                        This is the chain part.
REF(element)mom;        mom is a pointer to the element data structure.
    BEGIN
    internal variables
    coroutine           Input chaining coroutine.
    END;
Process CLASS Pmodel(sis);
                        This is the process part.
REF(Cmodel)sis;         sis points to the sister chain part.
    INSPECT sis do      The chain part is the central part.
    Loop calling passivate and hold
```

Figure 5-1: SIMULA Declarations for a Simulation Model.

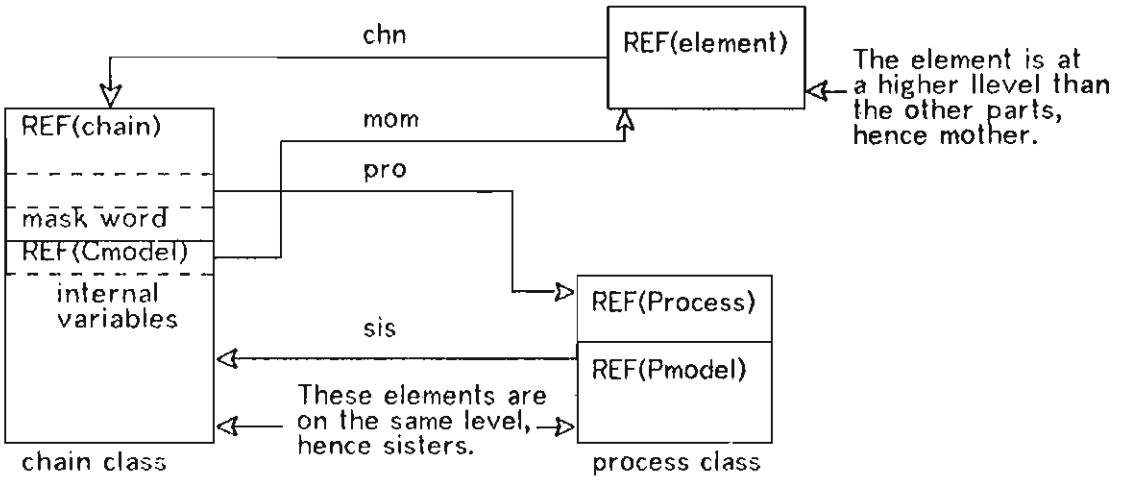

Figure 5-2: Illustration of Simulation Model.

5.2 Message System

Message delivery is implemented by the process part, the chain part, and system delivery procedures, just as proposed earlier. The input chaining routine is present in a simulation model as the chain CLASS. This CLASS serves as both a holder for the internal variables of the simulation model and a coroutine. The input chaining coroutine is activated by using the SIMULA procedure Call. A message delivery system is a set of procedures that examine the interconnect structure and activate the appropriate input chaining routines. The information content of the message is usually passed in a globally accessible variable, or it is implicit from the context of the message. Consider now a routine that delivers a message. There will be a part that is dependent on the message type. This part will just write the message value into the appropriate locations. There is also a connection dependent part. This part will activate the appropriate chaining routines, dependent upon the type of the connection. For example, if a message goes over a direct connection there is exactly one element on the other end of the connection, hence one activation. A wire connection would require a routine that follows the ring of pointers, activating every chaining routine until it gets back to the original port. These routines must test the appropriate flag in the mask word to determine if activation of the chaining routine should be suppressed. There are procedures that do these functions, but they are all called by other system procedures, hence they will not be further discussed.

The actual message delivery procedures used by the simulation models are less dependent upon the interconnection structure and more dependent upon the message type. The procedures will optimize somewhat their own use. For example, the wire change routines check to see if the wire's value is actually being changed, and if not they return immediately. There are an assortment of these procedures which are outlined below. With each delivery routine there is a corresponding routine which accepts the message. These procedures are twins, as both utilize the same method of transfering the information of the message.

- PBIT(dat,prt,elt). Puts boolean value dat on the wire connected to port prt of elt.

- GBIT(prt,elt). Returns the boolean value on the wire connected to port prt of elt.

- PINT(dat,prt,elt). Puts the integer value dat on the wire connected to port prt of elt.

- GINT(prt,elt). Returns the integer value on the wire connected to port prt of elt.

- NOTIFY(prt,elt). Sends an informationless message over the direct connection with one end at port prt of elt.

- RUNNING(prt,elt). Returns true if the simulation element connected directly to port prt of elt is in the run state.

- WAIT(prt,elt). Passivates the calling simulation model until the element connected directly to port prt of elt changes state. If the element is in a non-running state the procedure returns immediately.

These procedures assume that the connection the message is to be sent over is of the proper type. It is the responsibility of the setup phase to verify that the connections are of the proper types.

5.3 Interaction

Interaction with a particular simulation model is accomplished by the system faking a message to the interactive port on the element. This port has index zero in the arrays. The interaction routine will be within the chain class, and will be activated when the class is Called with port zero specified. This routine will usually call the system message procedure to describe its state, although the full I/O facilities are available.

The system will generate the call to port zero by the user issuing several commands from the terminal. One of these will do the call immediately, the other will merely alter the masking flag corresponding to port zero. After this the port will be activated normally when messages arrive to that port. These commands, and their

variations, are listed below:

- WHAT identifier. This does an activation of port zero on the element whose label is identifier. The setting of the mask flag is not checked.

- PROBE identifier. (Positive version) This turns off the masking flag corresponding to port zero on the element whose label is identifier. If the element is a wire this turns on tracing for that wire.

- -PROBE identifier. (Negative version) This reverses the effect of the PROBE command.

To start the simulation it is necessary for the main program to make a call to the simulation scheduler to increment simulation time. The commands that do this simply make a call on the SIMULA procedure Hold. The two variations of this command are listed below:

- GO time. This causes simulation to proceed for the time increment time before terminal entry is requested again.

- UNTIL time. This causes simulation to proceed until simulation time is equal to time before terminal entry is requested again.

5.4 Setup

Since simulation models can be freely included in the compiled SIMULA program their code modules must follow the structure outlined in the preceding chapter. This means that the chain and process parts may not be referenced from any other module. It also means there will be a few lines of code added to the SETUP part of the main program that will generate the instances of the simulation model. This code is in the form of a branch of a case statement. Within the branch these lines do various checking functions fetch parameters, and generate the proper simulation model. They must make the actual calls to the parameter fetching routines, as the parameters may be necessary to set up the rest of the model. It is also necessary to check the ports of every element to verify that they are connected to things of the right type. There is a system procedure which does this. It accepts a data

word with information about the number of ports and the required type of each. The procedure examines what is connected to each port and verifies that it is correct. In the case of a wire connection it is possible that the wire has no simulation model. In this case the proper wire model is automatically generated. If it is found that an incorrect wire model is present, or the connection is of the wrong type then an error condition occurs. This action happens during the setup phase, but the code that does this is usually present in the beginning of the input chaining class. This is to prevent two elements from being generated at the same time, as would happen when a wire is being automatically generated. Appendix I outlines the general structure for a simulation model.

6. Results

To demonstrate that the features discussed in this report are valuable, we described and simulated several versions of a microprocessor system. In each case a functionally described microprocessor was in a system with a memory and input device. In the different versions the input device was described in fundamentally different ways. In one version the input device contained a shift register made of flip flops. In a lower level version the flip flops were replaced with their equivalent combinations of nand gates. At a higher level the entire input device was replaced by a couple of functional equivalents. The microprocessor portion of the simulation is in some respects similar to a description in a functional description language. The microprocessor has a main part and three functional subdevices, like processes of a functional descriptive language. However the microprocessor differs from most functional languages in that it has connections outside of the module. These connections are the address, data buses, and control lines. Connected to these buses are elements of wildly varying types. There are nand gates connected to the control lines to make the proper interface to the other parts of the system. There are also high level devices connected to the buses, like the memory or functionally described input port.

Due to the diversity of the elements in the various simulations we have demonstrated the incorporation of different types of simulators into a single program. The program acts as a logic simulator when it simulates the individual nand gates that form the flip flops. When the program simulates the microprocessor it is acting as a functional simulator. In addition the functions of both simulators are performed simultaneously on the same design. Since functional simulation is fundamentally faster than logic simulation, a fact commonly believed and verified here, we achieve a performance that is greater than could be obtained by either a logic or functional simulator alone. A functional simulator simply cannot simulate to the detail achieved here, and a logic simulator would have required that the entire description be at the gate level, including the microprocessor. Due to the complexity of the microprocessor, simulating it as gates would be an order of magnitude slower than the slowest simulation demonstrated.

We complete this report by describing how these microprocessor simulations exercised the capabilities of the interconnection system and functional simulator, and discuss the implications of this.

6.1 Representation of the Microprocessor System

Each of the designs simulated a Z80 microprocessor connected to a memory and line receiver input port. The memory contained a program for the Z80. The program gets characters from the receiver and transfers them to memory. The first of the simulations used a receiver described as a number of flip flops. This description is used as the reference because the flip flop shift register is the most concise way of describing the logic structure of the input device. Briefly, there were eight flip flops organized as a shift register and driven by a 200kHz clock. An input bit stream was read into the first flip flop of the register. After a one was transfered into the eighth bit, indicating a start bit, the entire contents of the register is shifted into a latch and the register cleared. The receiver has seven data bits and an eighth bit to indicate when new data is received. This eighth bit toggles every time a character is received, providing a cheap but adequate way to synchronize with input. The simulation at a lower level replaced the flip flops in the shift register by their nand gate equivalents, resulting in over fifty gates. The higher level simulations replaced the input device with functional equivalents of different types.

Figures 6-1, 6-2, 6-3, and 6-4 show diagrams of the microprocessor part and the different receiver parts. There were actually two versions of the functional receiver, which will be discussed later. The reader is refered to the listings of the program runs in appendix 1 for the written representation of these interconnections.

There are several files containing the descriptions. The microprocessor and memory are the same in each case, and are in one file. The input device is different in each case, and hence a different file is used in each simulation. The complete interconnections were made by reading two files consecutively, causing them to be properly concatenated.

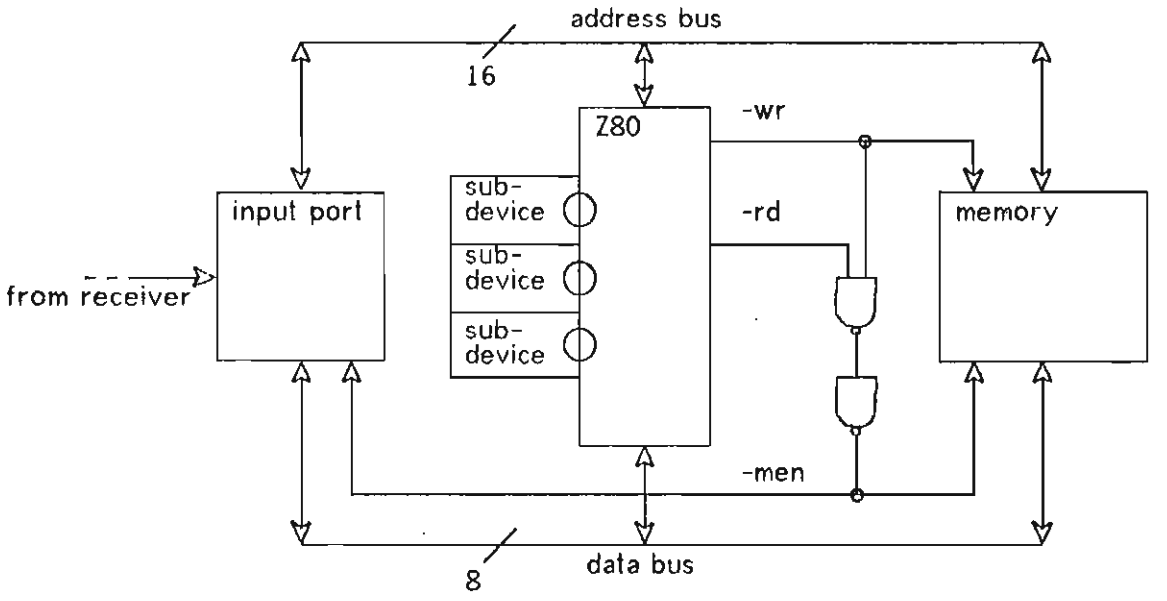

Figure 6-1: Microprocessor Part of Test Device

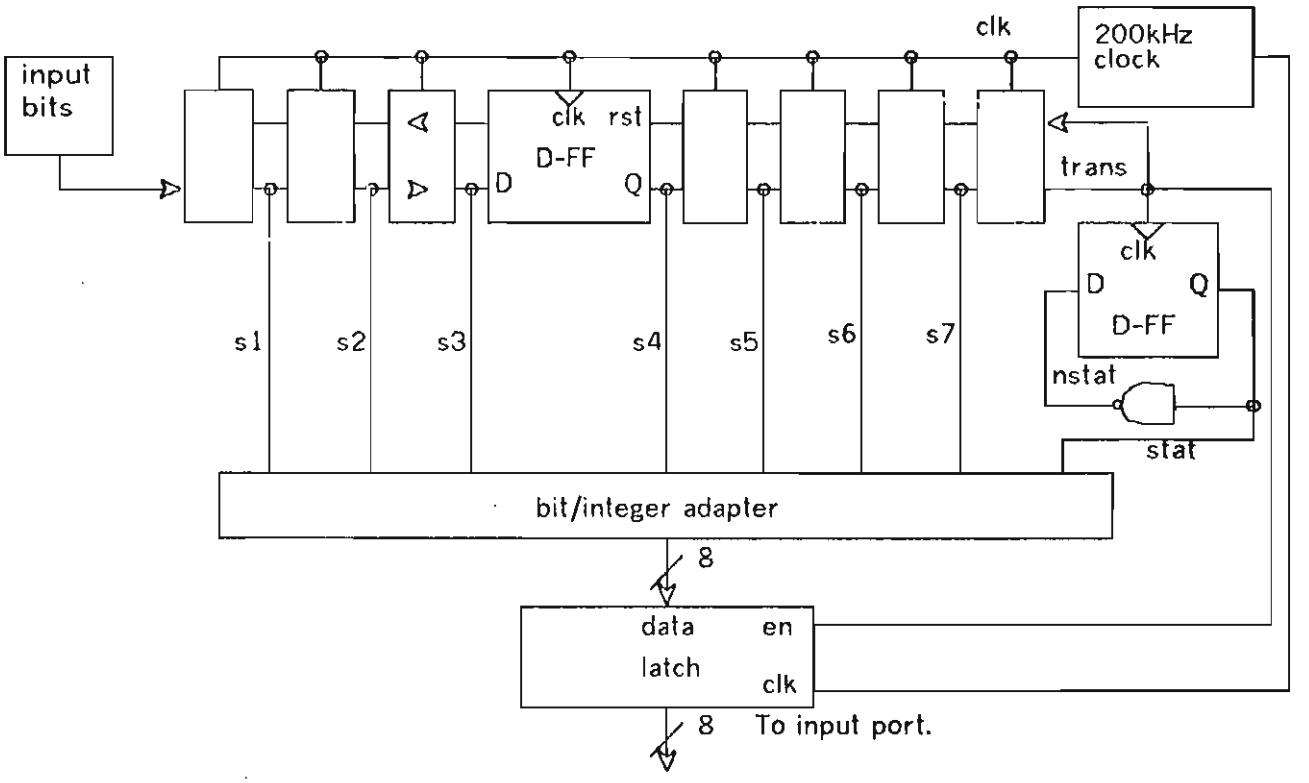

Figure 6-2: Input Part With Flip Flops

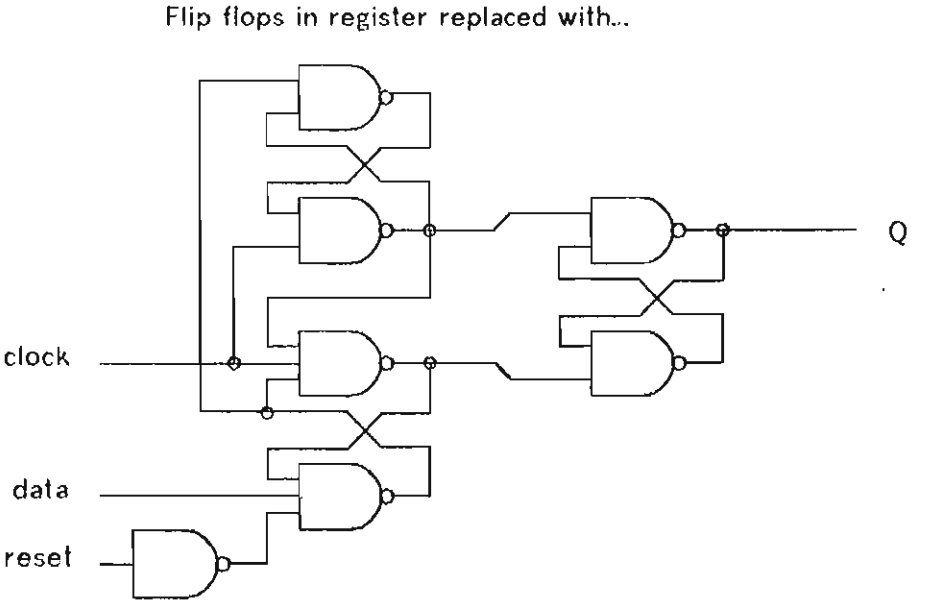

Figure 6-3: Input Part With Gates

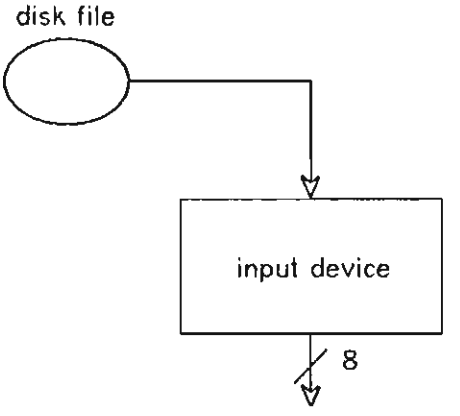

Figure 6-4: Functional Input Part

Some issues need to be resolved. In particular, how do we interface an input port with eight one bit outputs to a data bus with one eight bit input. The method used is to make an element that converts bits to buses and vice versa. In this case we use an element with nine ports, eight one bit ports, and an eight bit one. The element

simulates by changing the bus output whenever any bit input changes, and changing the bit outputs when the bus input changes. This element does not correspond to any physical part of the design, except perhaps a connecter, and is put into the description for compatibility reasons.

6.2 Simulation Models

The simulation models for the interconnected elements covered a wide range of complexities. The description of the Z80 is in the flavor of a functional description. The description involved a great deal of manipulation of internal state. The actual code for the Z80, most of which was internal state changes is eleven pages of code. At the other extreme is the nand gates, which have essentially no internal state, and whose description is about a half page. We will discuss briefly examples of elements implemented and mention the significant points about their operation.

The simplest element used, in terms of complexity of the physical device, was the nand gate. The simulation model was also among the simplest. The behavior of such a physical element can be fairly complex. The naive behavior is that the output becomes the nand function of its inputs within a time, called the propagation delay, after the inputs change. Complex behavior occurs when rapidly changing inputs are accounted for. To obtain any reasonable speed from the simulator it is necessary to restrict the accuracy of the simulation at some point. We arbitrarily chose to stop at the naive behavior. We described the output as changing to the nand function of the inputs after the inputs are stable for a set time. The nand gate has no internal variables. It implemented its function by reading the values from the input wires and writing them to the output.

A more complicated physical device is the D-type flip flop. Although it is physically more complex the functional description is no more lengthy than the nand gate's. The idealized behavior of this device, equivalent to the implemented behavior, is that the input is sampled at the rising edge of the clock and becomes the output after a propagation delay.

We now take a big jump and describe the memory element. This device has some of the more advanced features in it. For example, the memory will take parameters to specify, among other things, a filename for its initial contents. The memory also makes use of the interactive feature. The WHAT command can be used on a memory to enter an interactive mode. The contents of the memory can then be displayed on command.

The most complex device was the Z80 microprocessor. The functional description of this device was sectioned into a main part and subdevices. The subdevices do instruction fetching, read cycles, and write cycles. The Z80 also implements the interactive feature. There is a selected portion of its internal state, consisting of the contents of the general registers, that can be displayed by the WHAT command. The PROBE command will cause the display of the same internal state after each instruction is executed.

It should be emphasized that the Z80's functional description is, although written in SIMULA, a copy of the Z80's description in a functional description language. The description is formed of assignment and control statements organized into blocks and procedures. The syntax is somewhat different from that of actual functional description languages, but the fundamental concepts are very similar. Figure 6-5 is an example of a portion of the Z80's functional description which should illustrate this.

Some of the advantages and disadvantages of simulations with the different models should be immediately apparent to the reader. The simulations with flip flops of gates will yield an extremely accurate picture of how the real device will function. Simulations with functional elements may be of less accuracy, but will certainly be more efficient in terms of simulation time. It was in fact observed that the simulation time decreased as the design became more functionally described. The results of this is shown in figure 6-6.

The simulations with a functional input port deserve additional discussion. One of

```
INTEGER PROCEDURE FETCH1(RGS);      This procedure does a general operand fetch.
                                    The register field is analyzed to determine if
                                    it specifies a general register or memory.
INTEGER RGS;
    IF RGS = 6 THEN                 6 specifies memory reference. The contents
        BEGIN                       of memory addressed by an index register is
        WAIT(17,MOM);               used.
        WAIT(18,MOM);
        WAIT(19,MOM);               These three statements wait until the three
                                    memory cycling subprocesses are completed.
        PINT(REG[2],2,MOM);         This outputs the index register to the address
                                    lines.;
        NOTIFY(18,MOM);             This starts the read cycle subprocess.
        WAIT(18,MOM);               This waits until the read cycle has accesses
                                    memory. The read cycle is not finished.
        FETCH1:=GINT(3,MOM);        The contents of the data lines are returned.
        END
    ELSE
    IF RGS = 7 THEN                 7 specifies the accumulator.
        FETCH1:=SUBYTE(REG[3],15,8)
    ELSE
    IF NOT BIT(RGS,0) THEN
        FETCH1:=SUBYTE(REG[SUBYTE(RGS,2,1)],15,8)
                                    Even registers are in the left half.
    ELSE
        FETCH1:=SUBYTE(REG[SUBYTE(REG,2,1)],7,0);
                                    Odd registers are in the right half.
```

Figure 6-5: Portion of the Z80 Functional Description

Simulation	CPU time
Nand gate receiver.	103.5
Flip flop receiver.	44.5
Functional receiver.	31.7
Functional input port.	13.7

Figure 6-6: Comparative Timings For Test Simulations

these simulations was achieved by observing the output of the receiver as a function of time and mocking it with a functional description. This made the simulation time for the input part very small compared to the other parts. The total simulation time was limited because the microprocessor part was still required to simulate at its slow rate for the full simulation time. In fact the microprocessor was spending most of its time in a loop waiting for the input to change. This waste of simulation time could be partially removed. In many cases the information derived from the simulation is only whether the system worked, not how many times it executed a loop. We can improve on time efficiency in these cases by making a super functional receiver. The new receiver outputs the same values as the first, but it outputs a new value every time it is accessed. As the reader will note, this resulted in a significant increase in the simulation time efficiency.

7. Conclusions

The original intent of this project was merely to show that simulations could be made more efficient by combining structural and functional descriptions; i.e. to verify the factor of nine improvement in run time observed in the last chapter. In order to do this, however some fundamentally different issues had to be addressed and solved first. In particular, it was determined early on that the quality of a simulation system would be largely dependent on the manner in which interconnection information is handled. This led to the definition and later implementation of the interconnect system. After the interconnect system, as it now exists, was implemented a number of significant improvements became apparent. These improvements led to the discussion of interconnect systems presented in chapter 2.

With the interconnect system implemented it was possible to develop the simulation system orthogonally from the issues of the representation of interconnection. We succeeded in implementing the joint logic and functional simulator by essentially developing a functional description language capable of describing both elements normally functionally described and logic elements. This unified approach to functional descriptions, which was not anticipated in the beginning, is much more powerful than was necessary to implement the simulator.

We have presented a way in which a wide variety of design functions can be combined into a large, but well structured and flexible system. We have demonstrated that there is some advantage to this by combining logic and functional simulators to give a simulator with more capability than is present in both when separate. This is not a complete test of the system. The real test will come when many very different design functions are added to a system such as this. This demonstration should show that such an effort may be indeed justified.

I. Test Runs

We include here the listings of the actual test runs on a PDP-10. The system is intended to be interactive, but the runs were done under the batch system. This causes the appearance of the log output to be considerably different from what the user would see at a terminal.

There have been columns removed from the left margin, and some lines too long have been shortened.

I.1 Flip Flop Receiver

```
BATCON version 12(1041) running WORKS2 sequence 1661 in stream 1 for DEBEN
EDICTIS
Input from DSKB0:WORKS2.CTL[55453,113213]
Output to  DSKB0:WORKS2.LOG[55453,113213]
Job parameters
                Time:00:05:00 Unique:YES      Restart:NO

.LOGIN 55453/113213
JOB 13 CMU10D 7.V2/DEC 5.06B TTY122
Other jobs same PPN
1932   08-Apr-79      Sun

..TYPE TEST.ASM
        PSECT
        LD      HL,BUFFER
        LD      C,$00
LOOP:   LD      A,@$1000
        LD      B,A
        XOR     C
        AND     $80
        JR      Z,LOOP
        LD      A,B
        LD      C,A
        AND     $7F
        LD      @HL,A
        INC     HL
        JR      LOOP
BUFFER: BLOCK 3

..TYPE MIC.IL
IN:     INPUT/SIZE(4096,4096)       DATL MEN AB DB;
MICRO: Z80     PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:   Z80IF   MICRO.17;
FSUB2: Z80RD  MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:    MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:   NAND2  RD WR MME;
TTL2:   NAND2  MME MME MEN;
```

```
..TYPE REC.IL
R1:     DFF     INPUT CLK TRANS S1;
R2:     DFF     S1 CLK TRANS S2;
R3:     DFF     S2 CLK TRANS S3;
R4:     DFF     S3 CLK TRANS S4;
R5:     DFF     S4 CLK TRANS S5;
R6:     DFF     S5 CLK TRANS S6;
R7:     DFF     S6 CLK TRANS S7;
R8:     DFF     S7 CLK TRANS TRANS;
STATFF: DFF     NSTAT TRANS NIL STAT;
        NAND2   STAT STAT NSTAT;
CLOCK: CLOCK/TIME(0.000005)         CLK;
        FUNCTION/TIME(0.000001)/INI(FUN)    INPUT;
        ADAPT   DATA S1 S2 S3 S4 S5 S6 S7 STAT;
LAT:    LATCH   CLK TRANS DATA DATL;

..TYPE MEM.INI
:10000000211600OE003A001047A9E68028F7784F25
:06001000E67F772318EFE4
:00000001FF

..TYPE FUN.INI
32.5    1
37.5    1
42.5    0
47.5    0
52.5    1
57.5    0
62.5    0
67.5    0
72.5    0
77.5    1
82.5    1
87.5    0
92.5    0
97.5    1
102.5   0
107.5   0
112.5   1
117.5   0
122.5   1
127.5   0
132.5   1
```

```
137.5  0
142.5  0
147.5  0
152.5  0
157.5  1
162.5  0

..RUN J
**EXIT

2 Garbage collections executed during 167 ms

End of SIMULA program execution.
CPU time: -25:-34.|)    Elapsed time: 4:46.15

..START
**GET REC
**GET MIC
**ALL
R1:      DFF     INPUT CLK TRANS S1;
R2:      DFF     S1 CLK TRANS S2;
R3:      DFF     S2 CLK TRANS S3;
R4:      DFF     S3 CLK TRANS S4;
R5:      DFF     S4 CLK TRANS S5;
R6:      DFF     S5 CLK TRANS S6;
R7:      DFF     S6 CLK TRANS S7;
R8:      DFF     S7 CLK TRANS TRANS;
STATFF: DFF      NSTAT TRANS NIL STAT;
         NAND2   STAT STAT NSTAT;
CLOCK: CLOCK/TIME(0.000005)          CLK;
         FUNCTION/TIME(0.000001)/INI(FUN)    INPUT;
         ADAPT   DATA S1 S2 S3 S4 S5 S6 S7 STAT;
LAT:     LATCH   CLK TRANS DATA DATL;
IN:      INPUT/SIZE(4096,4096)          DATL MEN AB DB;
MICRO: Z80      PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:   Z80IF   MICRO.17;
FSUB2: Z80RD   MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:    MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:   NAND2   RD WR MME;
TTL2:   NAND2   MME MME MEN;
**SIMU
**WHAT MEM
```

```
-->  0.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
**PROBE MICRO
**PROBE DATL
**UN .0001
-->  2.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0003
-->  4.50us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
-->  7.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
-->  8.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
-->  9.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 11.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 13.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 16.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 17.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 18.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 20.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 22.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 25.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 26.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 27.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 29.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 30.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 34.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 35.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 36.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
```

```
--> 37.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 39.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 42.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 43.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 44.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 46.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 48.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 51.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 52.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 53.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 55.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 57.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 60.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 61.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 62.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 64.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 65.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 69.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 70.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 71.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 72.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 74.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 75.02us DATL   200.
--> 77.75us A:C8 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 78.75us A:C8 F:40 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 79.75us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 81.50us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 83.25us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000E
--> 84.25us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000F
--> 85.25us A:C8 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0010
--> 87.00us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0012
--> 88.75us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0013
--> 89.75us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0014
--> 91.50us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0005
--> 94.75us A:C8 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0008
--> 95.75us A:C8 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0009
--> 96.75us A:00 F:40 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000A
--> 98.50us A:00 F:40 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000C
--> 100.00us
**WHAT MEM
--> 100.00us MEMORY   0:   64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
```

```
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
0016 is 48
**-PR MICRO
**UN .0002
--> 120.02us DATL    73.
--> 165.02us DATL   161.
--> 200.00us
**WHAT MEM
--> 200.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
```

```
0015 is EF
0016 is 48
0017 is 49
0018 is 21
**EXIT

12 Garbage collections executed during 7669 ms

End of SIMULA program execution.
CPU time: 44.47 Elapsed time: 2:27.12

.KJOB DSKB0:WORKS2.LOG=/W/B/Z:4/VR:10/VS:1661/VL:200/VP:10/VD:P
Other jobs same PPN

%WLDAFR All files rejected by conditions DSKB0:WORKS2.LOG[55453,113213]
Job 13, User [X330ED0J]  Logged off TTY122   1935  8-Apr-79
Another job still logged in under [X330ED0J]
Runtime 49.02 sec; Kilocore sec: 1834
Total of 303 disk blocks read, 10 written
Connect time 0 hr, 2 min, 55 sec; Total charge: $4.11
```

I.2 Functional Receiver

```
BATCON version 12(1041) running WORKS4 sequence 1662 in stream 1 for DEBEN
EDICTIS
Input from DSKB1:WORKS3.CTL[55453,113213]
Output to  DSKB1:WORKS4.LOG[55453,113213]
Job parameters
                Time:00:05:00 Unique:YES      Restart:NO

.LOGIN 55453/113213
JOB 13 CMU10D 7.V2/DEC 5.06B TTY122
Other jobs same PPN
1935   08-Apr-79      Sun

..TYPE TEST.ASM
        PSECT
        LD      HL,BUFFER
        LD      C,$00
LOOP:   LD      A,@$1000
        LD      B,A
        XOR     C
        AND     $80
        JR      Z,LOOP
        LD      A,B
        LD      C,A
        AND     $7F
        LD      @HL,A
        INC     HL
        JR      LOOP
BUFFER: BLOCK 3

..TYPE MIC.IL
IN:     INPUT/SIZE(4096,4096)       DATL MEN AB DB;
MICRO: Z80     PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:   Z80IF   MICRO.17;
FSUB2: Z80RD  MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:    MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:   NAND2  RD WR MME;
TTL2:   NAND2  MME MME MEN;
```

```
..TYPE REC.IL
R1:     DFF     INPUT CLK TRANS S1;
R2:     DFF     S1 CLK TRANS S2;
R3:     DFF     S2 CLK TRANS S3;
R4:     DFF     S3 CLK TRANS S4;
R5:     DFF     S4 CLK TRANS S5;
R6:     DFF     S5 CLK TRANS S6;
R7:     DFF     S6 CLK TRANS S7;
R8:     DFF     S7 CLK TRANS TRANS;
STATFF: DFF     NSTAT TRANS NIL STAT;
        NAND2   STAT STAT NSTAT;
CLOCK: CLOCK/TIME(0.000005)         CLK;
        FUNCTION/TIME(0.000001)/INI(FUN)    INPUT;
        ADAPT   DATA S1 S2 S3 S4 S5 S6 S7 STAT;
LAT:    LATCH   CLK TRANS DATA DATL;

..TYPE MEM.INI
:10000000211600OE003A001047A9E68028F7784F25
:06001000E67F772318EFE4
:00000001FF

..TYPE FUNBUS.INI
 75.0   200
120.0   73
165.0   161

..RUN J
**EXIT

2 Garbage collections executed during 167 ms

End of SIMULA program execution.
CPU time: -25:-34.[]    Elapsed time: 7:58.88

..START
**ADD FUNBUS/TIME(0.000001)/INI(FUNBUS) DATL;
**GET MIC
**ALL
        FUNBUS/TIME(0.000001)/INI(FUNBUS) DATL;
IN:     INPUT/SIZE(4096,4096)       DATL MEN AB DB;
MICRO: Z80      PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:   Z80IF   MICRO.17;
```

```
FSUB2: Z80RD  MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:   MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:  NAND2  RD WR MME;
TTL2:  NAND2  MME MME MEN;
**SIMU
**WHAT MEM
-->  0.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
**PROBE MICRO
**PROBE DATL
**UN .0001
-->   2.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0003
-->   4.50us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
-->   7.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
-->   8.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
-->   9.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
-->  11.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
-->  13.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
-->  16.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
-->  17.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
-->  18.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
-->  20.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
-->  22.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
-->  25.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
```

```
--> 26.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 27.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 29.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 30.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 34.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 35.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 36.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 37.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 39.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 42.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 43.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 44.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 46.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 48.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 51.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 52.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 53.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 55.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 57.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 60.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 61.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 62.25us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 64.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 65.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 69.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 70.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 71.00us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 72.75us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 74.50us A:00 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 75.00us DATL   200.
--> 77.75us A:C8 F:40 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 78.75us A:C8 F:40 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 79.75us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 81.50us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 83.25us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000E
--> 84.25us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000F
--> 85.25us A:C8 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0010
--> 87.00us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0012
--> 88.75us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0013
--> 89.75us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0014
--> 91.50us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0005
--> 94.75us A:C8 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0008
--> 95.75us A:C8 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0009
--> 96.75us A:00 F:40 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000A
```

```
--> 98.50us A:00 F:40 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000C
--> 100.00us
**WHAT MEM
--> 100.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
0016 is 48
**-PR MICRO
**UN .0002
--> 120.00us DATL    73.
--> 165.00us DATL   161.
--> 200.00us
**WHAT MEM
--> 200.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
```

```
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
0016 is 48
0017 is 49
0018 is 21
**EXIT

12 Garbage collections executed during 5417 ms

End of SIMULA program execution.
CPU time: 31.67 Elapsed time: 1:32.67

.KJOB DSKB1:WORKS4.LOG=/W/B/Z:4/VR:10/VS:1662/VL:200/VP:10/VD:P
Other jobs same PPN

%WLDAFR All files rejected by conditions DSKB1:WORKS4.LOG[55453,113213]
Job 13, User [X330ED0J]  Logged off TTY122   1937  8-Apr-79
Another job still logged in under [X330ED0J]
Runtime 36.14 sec; Kilocore sec: 1246
Total of 542 disk blocks read, 12 written
Connect time 0 hr, 2 min, 10 sec; Total charge: $3.14
```

I.3 Nand Gate Receiver

```
BATCON version 12(1041) running WORKS5 sequence 1664 in stream 1 for DEBEN
EDICTIS
Input from DSKB0:WORKS5.CTL[55453,113213]
Output to  DSKB0:WORKS5.LOG[55453,113213]
Job parameters
                Time:00:05:00 Unique:YES      Restart:NO

.LOGIN 55453/113213
JOB 13 CMU10D 7.V2/DEC 5.06B TTY122
Other jobs same PPN
1941   08-Apr-79      Sun

..TYPE TEST.ASM
        PSECT
        LD      HL,BUFFER
        LD      C,$00
LOOP:   LD      A,@$1000
        LD      B,A
        XOR     C
        AND     $80
        JR      Z,LOOP
        LD      A,B
        LD      C,A
        AND     $7F
        LD      @HL,A
        INC     HL
        JR      LOOP
BUFFER: BLOCK 3

..TYPE MIC.IL
IN:     INPUT/SIZE(4096,4096)       DATL MEN AB DB;
MICRO: Z80    PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:   Z80IF  MICRO.17;
FSUB2: Z80RD  MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:    MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:   NAND2  RD WR MME;
TTL2:   NAND2  MME MME MEN;
```

```
..TYPE RECCC.IL
R1N1:  NAND2  R1I4 R1I2 R1I1;
R1N2:  NAND2  R1I1 CLK R1I2;
R1N3:  NAND3  R1I2 CLK R1I4 R1I3;
R1N4:  NAND3  R1I3 INPUT NC R1I4;
R1N5:  NAND2  R1I2 NUL1 S1;
R1N6:  NAND2  S1 R1I3 NUL1;
R2N1:  NAND2  R2I4 R2I2 R2I1;
R2N2:  NAND2  R2I1 CLK R2I2;
R2N3:  NAND3  R2I2 CLK R2I4 R2I3;
R2N4:  NAND3  R2I3 S1 NC R2I4;
R2N5:  NAND2  R2I2 NUL2 S2;
R2N6:  NAND2  S2 R2I3 NUL2;
R3N1:  NAND2  R3I4 R3I2 R3I1;
R3N2:  NAND2  R3I1 CLK R3I2;
R3N3:  NAND3  R3I2 CLK R3I4 R3I3;
R3N4:  NAND3  R3I3 S2 NC R3I4;
R3N5:  NAND2  R3I2 NUL3 S3;
R3N6:  NAND2  S3 R3I3 NUL3;
R4N1:  NAND2  R4I4 R4I2 R4I1;
R4N2:  NAND2  R4I1 CLK R4I2;
R4N3:  NAND3  R4I2 CLK R4I4 R4I3;
R4N4:  NAND3  R4I3 S3 NC R4I4;
R4N5:  NAND2  R4I2 NUL4 S4;
R4N6:  NAND2  S4 R4I3 NUL4;
R5N1:  NAND2  R5I4 R5I2 R5I1;
R5N2:  NAND2  R5I1 CLK R5I2;
R5N3:  NAND3  R5I2 CLK R5I4 R5I3;
R5N4:  NAND3  R5I3 S4 NC R5I4;
R5N5:  NAND2  R5I2 NUL5 S5;
R5N6:  NAND2  S5 R5I3 NUL5;
R6N1:  NAND2  R6I4 R6I2 R6I1;
R6N2:  NAND2  R6I1 CLK R6I2;
R6N3:  NAND3  R6I2 CLK R6I4 R6I3;
R6N4:  NAND3  R6I3 S5 NC R6I4;
R6N5:  NAND2  R6I2 NUL6 S6;
R6N6:  NAND2  S6 R6I3 NUL6;
R7N1:  NAND2  R7I4 R7I2 R7I1;
R7N2:  NAND2  R7I1 CLK R7I2;
R7N3:  NAND3  R7I2 CLK R7I4 R7I3;
R7N4:  NAND3  R7I3 S6 NC R7I4;
R7N5:  NAND2  R7I2 NUL7 S7;
R7N6:  NAND2  S7 R7I3 NUL7;
```

```
R8N1:   NAND2  R8I4 R8I2 R8I1;
R8N2:   NAND2  R8I1 CLK R8I2;
R8N3:   NAND3  R8I2 CLK R8I4 R8I3;
R8N4:   NAND3  R8I3 S7 NC R8I4;
R8N5:   NAND2  R8I2 NUL8 TRANS;
R8N6:   NAND2  TRANS R8I3 NUL8;
R8NV:   NAND2  TRANS TRANS NC;
STATFF: DFF    NSTAT TRANS NIL STAT;
        NAND2  STAT STAT NSTAT;
CLOCK: CLOCK/TIME(0.000005)        CLK;
        FUNCTION/TIME(0.000001)/INI(FUN)    INPUT;
        ADAPT  DATA S1 S2 S3 S4 S5 S6 S7 STAT;
LAT:    LATCH  CLK TRANS DATA DATL;

..TYPE MEM.INI
:10000000211600OE003A001047A9E68028F7784F25
:06001000E67F772318EFE4
:00000001FF

..TYPE FUN.INI
32.5    1
37.5    1
42.5    0
47.5    0
52.5    1
57.5    0
62.5    0
67.5    0
72.5    0
77.5    1
82.5    1
87.5    0
92.5    0
97.5    1
102.5   0
107.5   0
112.5   1
117.5   0
122.5   1
127.5   0
132.5   1
137.5   0
142.5   0
147.5   0
```

```
152.5  0
157.5  1
162.5  0

..RUN J
**EXIT

2 Garbage collections executed during 167 ms

End of SIMULA program execution.
CPU time: -25:-33.'   Elapsed time: 13:28.20

..START
**GET RECCC
**GET MIC
**ALL
R1N1:  NAND2  R1I4 R1I2 R1I1;
R1N2:  NAND2  R1I1 CLK R1I2;
R1N3:  NAND3  R1I2 CLK R1I4 R1I3;
R1N4:  NAND3  R1I3 INPUT NC R1I4;
R1N5:  NAND2  R1I2 NUL1 S1;
R1N6:  NAND2  S1 R1I3 NUL1;
R2N1:  NAND2  R2I4 R2I2 R2I1;
R2N2:  NAND2  R2I1 CLK R2I2;
R2N3:  NAND3  R2I2 CLK R2I4 R2I3;
R2N4:  NAND3  R2I3 S1 NC R2I4;
R2N5:  NAND2  R2I2 NUL2 S2;
R2N6:  NAND2  S2 R2I3 NUL2;
R3N1:  NAND2  R3I4 R3I2 R3I1;
R3N2:  NAND2  R3I1 CLK R3I2;
R3N3:  NAND3  R3I2 CLK R3I4 R3I3;
R3N4:  NAND3  R3I3 S2 NC R3I4;
R3N5:  NAND2  R3I2 NUL3 S3;
R3N6:  NAND2  S3 R3I3 NUL3;
R4N1:  NAND2  R4I4 R4I2 R4I1;
R4N2:  NAND2  R4I1 CLK R4I2;
R4N3:  NAND3  R4I2 CLK R4I4 R4I3;
R4N4:  NAND3  R4I3 S3 NC R4I4;
R4N5:  NAND2  R4I2 NUL4 S4;
R4N6:  NAND2  S4 R4I3 NUL4;
R5N1:  NAND2  R5I4 R5I2 R5I1;
R5N2:  NAND2  R5I1 CLK R5I2;
R5N3:  NAND3  R5I2 CLK R5I4 R5I3;
R5N4:  NAND3  R5I3 S4 NC R5I4;
```

```
R5N5:  NAND2  R5I2 NUL5 S5;
R5N6:  NAND2  S5 R5I3 NUL5;
R6N1:  NAND2  R6I4 R6I2 R6I1;
R6N2:  NAND2  R6I1 CLK R6I2;
R6N3:  NAND3  R6I2 CLK R6I4 R6I3;
R6N4:  NAND3  R6I3 S5 NC R6I4;
R6N5:  NAND2  R6I2 NUL6 S6;
R6N6:  NAND2  S6 R6I3 NUL6;
R7N1:  NAND2  R7I4 R7I2 R7I1;
R7N2:  NAND2  R7I1 CLK R7I2;
R7N3:  NAND3  R7I2 CLK R7I4 R7I3;
R7N4:  NAND3  R7I3 S6 NC R7I4;
R7N5:  NAND2  R7I2 NUL7 S7;
R7N6:  NAND2  S7 R7I3 NUL7;
R8N1:  NAND2  R8I4 R8I2 R8I1;
R8N2:  NAND2  R8I1 CLK R8I2;
R8N3:  NAND3  R8I2 CLK R8I4 R8I3;
R8N4:  NAND3  R8I3 S7 NC R8I4;
R8N5:  NAND2  R8I2 NUL8 TRANS;
R8N6:  NAND2  TRANS R8I3 NUL8;
R8NV:  NAND2  TRANS TRANS NC;
STATFF: DFF    NSTAT TRANS NIL STAT;
        NAND2  STAT STAT NSTAT;
CLOCK: CLOCK/TIME(0.000005)        CLK;
        FUNCTION/TIME(0.000001)/INI(FUN)   INPUT;
        ADAPT  DATA S1 S2 S3 S4 S5 S6 S7 STAT;
LAT:    LATCH  CLK TRANS DATA DATL;
IN:     INPUT/SIZE(4096,4096)        DATL MEN AB DB;
MICRO: Z80     PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:  Z80IF   MICRO.17;
FSUB2: Z80RD  MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:   MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:  NAND2  RD WR MME;
TTL2:  NAND2  MME MME MEN;
**SIMU
**WHAT MEM
-->  0.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
```

```
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
**PROBE MICRO
**PROBE DATL
**PR S1
**PR S2
**PR S3
**PR S4
**PR S5
**PR S6
**PR S7
**PR TRANS
**PR INPUT
**PR CLK
**UN .0001
-->  0.00us CLK TRUE.
-->  0.01us S1 TRUE.
-->  0.01us S2 TRUE.
-->  0.01us S3 TRUE.
-->  0.01us S4 TRUE.
-->  0.01us S5 TRUE.
-->  0.01us S6 TRUE.
-->  0.01us S7 TRUE.
-->  0.01us TRANS TRUE.
-->  2.50us CLK FALSE.
-->  2.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0003
-->  4.50us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
-->  5.00us CLK TRUE.
-->  5.02us DATL   127.
-->  5.03us TRANS FALSE.
-->  5.03us S7 FALSE.
```

```
-->  5.03us S6 FALSE.
-->  5.03us S5 FALSE.
-->  5.03us S4 FALSE.
-->  5.03us S3 FALSE.
-->  5.03us S2 FALSE.
-->  5.03us S1 FALSE.
-->  7.50us CLK FALSE.
-->  7.75us A:7F F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
-->  8.75us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
-->  9.75us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 10.00us CLK TRUE.
--> 11.50us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 12.50us CLK FALSE.
--> 13.25us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 15.00us CLK TRUE.
--> 16.50us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 17.50us CLK FALSE.
--> 17.50us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 18.50us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 20.00us CLK TRUE.
--> 20.25us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 22.00us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 22.50us CLK FALSE.
--> 25.00us CLK TRUE.
--> 25.25us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 26.25us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 27.25us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 27.50us CLK FALSE.
--> 29.00us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 30.00us CLK TRUE.
--> 30.75us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 32.50us INPUT TRUE.
--> 32.50us CLK FALSE.
--> 34.00us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 35.00us CLK TRUE.
--> 35.00us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 35.02us S1 TRUE.
--> 36.00us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 37.50us CLK FALSE.
--> 37.75us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 39.50us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 40.00us CLK TRUE.
--> 40.02us S2 TRUE.
--> 42.50us CLK FALSE.
```

```
--> 42.50us INPUT FALSE.
--> 42.75us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 43.75us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 44.75us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 45.00us CLK TRUE.
--> 45.02us S3 TRUE.
--> 45.03us S1 FALSE.
--> 46.50us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 47.50us CLK FALSE.
--> 48.25us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 50.00us CLK TRUE.
--> 50.02us S4 TRUE.
--> 50.03us S2 FALSE.
--> 51.50us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 52.50us CLK FALSE.
--> 52.50us INPUT TRUE.
--> 52.50us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 53.50us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 55.00us CLK TRUE.
--> 55.02us S5 TRUE.
--> 55.02us S1 TRUE.
--> 55.03us S3 FALSE.
--> 55.25us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 57.00us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 57.50us CLK FALSE.
--> 57.50us INPUT FALSE.
--> 60.00us CLK TRUE.
--> 60.02us S6 TRUE.
--> 60.02us S2 TRUE.
--> 60.03us S4 FALSE.
--> 60.03us S1 FALSE.
--> 60.25us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 61.25us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 62.25us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 62.50us CLK FALSE.
--> 64.00us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 65.00us CLK TRUE.
--> 65.02us S7 TRUE.
--> 65.02us S3 TRUE.
--> 65.03us S5 FALSE.
--> 65.03us S2 FALSE.
--> 65.75us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 67.50us CLK FALSE.
--> 69.00us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
```

```
--> 70.00us CLK TRUE.
--> 70.00us A:7F F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 70.02us TRANS TRUE.
--> 70.02us S4 TRUE.
--> 70.03us S6 FALSE.
--> 70.03us S3 FALSE.
--> 71.00us A:7F F:00 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 72.50us CLK FALSE.
--> 72.75us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 74.50us A:00 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
--> 75.00us CLK TRUE.
--> 75.02us DATL   200.
--> 75.03us TRANS FALSE.
--> 75.03us S7 FALSE.
--> 75.03us S4 FALSE.
--> 77.50us INPUT TRUE.
--> 77.50us CLK FALSE.
--> 77.75us A:C8 F:40 BC:7F00 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
--> 78.75us A:C8 F:40 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
--> 79.75us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
--> 80.00us CLK TRUE.
--> 80.02us S1 TRUE.
--> 81.50us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
--> 82.50us CLK FALSE.
--> 83.25us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000E
--> 84.25us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000F
--> 85.00us CLK TRUE.
--> 85.02us S2 TRUE.
--> 85.25us A:C8 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0010
--> 87.00us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0012
--> 87.50us CLK FALSE.
--> 87.50us INPUT FALSE.
--> 88.75us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0013
--> 89.75us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0014
--> 90.00us CLK TRUE.
--> 90.02us S3 TRUE.
--> 90.03us S1 FALSE.
--> 91.50us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0005
--> 92.50us CLK FALSE.
--> 94.75us A:C8 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0008
--> 95.00us CLK TRUE.
--> 95.02us S4 TRUE.
--> 95.03us S2 FALSE.
--> 95.75us A:C8 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0009
```

```
--> 96.75us A:00 F:40 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000A
--> 97.50us CLK FALSE.
--> 97.50us INPUT TRUE.
--> 98.50us A:00 F:40 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000C
--> 100.00us CLK TRUE.
--> 100.00us
**WHAT MEM
--> 100.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
0016 is 48
**-PR MICRO
**UN .0002
--> 100.02us S5 TRUE.
--> 100.02us S1 TRUE.
--> 100.03us S3 FALSE.
--> 102.50us CLK FALSE.
--> 102.50us INPUT FALSE.
--> 105.00us CLK TRUE.
--> 105.02us S6 TRUE.
--> 105.02us S2 TRUE.
--> 105.03us S4 FALSE.
--> 105.03us S1 FALSE.
--> 107.50us CLK FALSE.
--> 110.00us CLK TRUE.
--> 110.02us S7 TRUE.
```

--> 110.02us S3 TRUE.
--> 110.03us S5 FALSE.
--> 110.03us S2 FALSE.
--> 112.50us CLK FALSE.
--> 112.50us INPUT TRUE.
--> 115.00us CLK TRUE.
--> 115.02us TRANS TRUE.
--> 115.02us S4 TRUE.
--> 115.02us S1 TRUE.
--> 115.03us S6 FALSE.
--> 115.03us S3 FALSE.
--> 117.50us CLK FALSE.
--> 117.50us INPUT FALSE.
--> 120.00us CLK TRUE.
--> 120.02us DATL 73.
--> 120.03us TRANS FALSE.
--> 120.03us S7 FALSE.
--> 120.03us S4 FALSE.
--> 120.03us S1 FALSE.
--> 122.50us CLK FALSE.
--> 122.50us INPUT TRUE.
--> 125.00us CLK TRUE.
--> 125.02us S1 TRUE.
--> 127.50us INPUT FALSE.
--> 127.50us CLK FALSE.
--> 130.00us CLK TRUE.
--> 130.02us S2 TRUE.
--> 130.03us S1 FALSE.
--> 132.50us INPUT TRUE.
--> 132.50us CLK FALSE.
--> 135.00us CLK TRUE.
--> 135.02us S3 TRUE.
--> 135.02us S1 TRUE.
--> 135.03us S2 FALSE.
--> 137.50us INPUT FALSE.
--> 137.50us CLK FALSE.
--> 140.00us CLK TRUE.
--> 140.02us S4 TRUE.
--> 140.02us S2 TRUE.
--> 140.03us S3 FALSE.
--> 140.03us S1 FALSE.
--> 142.50us CLK FALSE.
--> 145.00us CLK TRUE.
--> 145.02us S5 TRUE.

```
--> 145.02us S3 TRUE.
--> 145.03us S4 FALSE.
--> 145.03us S2 FALSE.
--> 147.50us CLK FALSE.
--> 150.00us CLK TRUE.
--> 150.02us S6 TRUE.
--> 150.02us S4 TRUE.
--> 150.03us S5 FALSE.
--> 150.03us S3 FALSE.
--> 152.50us CLK FALSE.
--> 155.00us CLK TRUE.
--> 155.02us S7 TRUE.
--> 155.02us S5 TRUE.
--> 155.03us S6 FALSE.
--> 155.03us S4 FALSE.
--> 157.50us INPUT TRUE.
--> 157.50us CLK FALSE.
--> 160.00us CLK TRUE.
--> 160.02us TRANS TRUE.
--> 160.02us S6 TRUE.
--> 160.02us S1 TRUE.
--> 160.03us S7 FALSE.
--> 160.03us S5 FALSE.
--> 162.50us INPUT FALSE.
--> 162.50us CLK FALSE.
--> 165.00us CLK TRUE.
--> 165.02us DATL    161.
--> 165.03us TRANS FALSE.
--> 165.03us S6 FALSE.
--> 165.03us S1 FALSE.
--> 167.50us CLK FALSE.
--> 170.00us CLK TRUE.
--> 172.50us CLK FALSE.
--> 175.00us CLK TRUE.
--> 177.50us CLK FALSE.
--> 180.00us CLK TRUE.
--> 182.50us CLK FALSE.
--> 185.00us CLK TRUE.
--> 187.50us CLK FALSE.
--> 190.00us CLK TRUE.
--> 192.50us CLK FALSE.
--> 195.00us CLK TRUE.
--> 197.50us CLK FALSE.
--> 200.00us
```

```
**WHAT MEM
--> 200.00us MEMORY    0:   64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
0016 is 48
0017 is 49
0018 is 21
**EXIT

17 Garbage collections executed during 23599 ms

End of SIMULA program execution.
CPU time: 1:43.47       Elapsed time: 3:24.75

.KJOB DSKB0:WORKS5.LOG=/W/B/Z:4/VR:10/VS:1664/VL:200/VP:10/VD:P
Other jobs same PPN

%WLDAFR All files rejected by conditions DSKB0:WORKS5.LOG[55453,113213]
Job 13, User [X330ED0J]  Logged off TTY122   1945  8-Apr-79
Another job still logged in under [X330ED0J]
Runtime 1 Min, 49.90 sec; Kilocore sec: 5209
Total of 550 disk blocks read, 11 written
Connect time 0 hr, 4 min, 4 sec; Total charge: $11.17
```

I.4 Functional Input Port

```
BATCON version 12(1041) running WORKS6 sequence 1665 in stream 1 for DEBEN
EDICTIS
Input from DSKB1:WORKS6.CTL[55453,113213]
Output to  DSKB1:WORKS6.LOG[55453,113213]
Job parameters
                Time:00:05:00 Unique:YES     Restart:NO

.LOGIN 55453/113213
JOB 13 CMU10D 7.V2/DEC 5.06B TTY122
Other jobs same PPN
1945  08-Apr-79      Sun

..TYPE TEST.ASM
        PSECT
        LD      HL,BUFFER
        LD      C,$00
LOOP:   LD      A,@$1000
        LD      B,A
        XOR     C
        AND     $80
        JR      Z,LOOP
        LD      A,B
        LD      C,A
        AND     $7F
        LD      @HL,A
        INC     HL
        JR      LOOP
BUFFER: BLOCK 3

..TYPE MICC.IL
IN:     INDATA/SIZE(4096,4096)/INI(INDATA)  MEN AB DB;
MICRO: Z80     PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:   Z80IF   MICRO.17;
FSUB2: Z80RD  MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:    MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:   NAND2  RD WR MME;
TTL2:   NAND2  MME MME MEN;
```

```
..TYPE MEM.INI
:10000000211600E003A001047A9E68028F7784F25
:06001000E67F772318EFE4
:00000001FF

..TYPE INDATA.INI
200
73
161

..RUN J
**EXIT

2 Garbage collections executed during 167 ms

End of SIMULA program execution.
CPU time: -25:-35.)   Elapsed time: 17:26.30

..START
**GET MICC
**ALL
IN:      INDATA/SIZE(4096,4096)/INI(INDATA)  MEN AB DB;
MICRO: Z80     PHI AB DB MREQ IORQ RD WR RFSH M1 RESET INT NMI WAIT HALT
BUSRQ BUSAK FSUB. 1 FSUB2. 1 FSUB3. 1;
FSUB:   Z80IF   MICRO.17;
FSUB2: Z80RD  MICRO.18;
FSUB3: Z80WR MICRO.19;
MEM:    MEMORY/INI(MEM)/SIZE(0,64)  MEN WR DB AB;
TTL1:   NAND2  RD WR MME;
TTL2:   NAND2  MME MME MEN;
**SIMU
**WHAT MEM
-->   0.00us MEMORY    0:   64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
```

```
000C is 28
000D is F7
000E Is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
**PROBE MICRO
**UN .00006
-->   2.75us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0003
-->   4.50us A:00 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0005
-->   7.75us A:C8 F:00 BC:0000 DE:0000 HL:0016 IX:0000 SP:0000 PC:0008
-->   8.75us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:0009
-->   9.75us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000A
-->  11.50us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000C
-->  13.25us A:80 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000E
-->  14.25us A:C8 F:00 BC:C800 DE:0000 HL:0016 IX:0000 SP:0000 PC:000F
-->  15.25us A:C8 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0010
-->  17.00us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0012
-->  18.75us A:48 F:00 BC:C8C8 DE:0000 HL:0016 IX:0000 SP:0000 PC:0013
-->  19.75us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0014
-->  21.50us A:48 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0005
-->  24.75us A:49 F:00 BC:C8C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0008
-->  25.75us A:49 F:00 BC:49C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:0009
-->  26.75us A:81 F:00 BC:49C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000A
-->  28.50us A:80 F:00 BC:49C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000C
-->  30.25us A:80 F:00 BC:49C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000E
-->  31.25us A:49 F:00 BC:49C8 DE:0000 HL:0017 IX:0000 SP:0000 PC:000F
-->  32.25us A:49 F:00 BC:4949 DE:0000 HL:0017 IX:0000 SP:0000 PC:0010
-->  34.00us A:49 F:00 BC:4949 DE:0000 HL:0017 IX:0000 SP:0000 PC:0012
-->  35.75us A:49 F:00 BC:4949 DE:0000 HL:0017 IX:0000 SP:0000 PC:0013
-->  36.75us A:49 F:00 BC:4949 DE:0000 HL:0018 IX:0000 SP:0000 PC:0014
-->  38.50us A:49 F:00 BC:4949 DE:0000 HL:0018 IX:0000 SP:0000 PC:0005
-->  41.75us A:A1 F:00 BC:4949 DE:0000 HL:0018 IX:0000 SP:0000 PC:0008
-->  42.75us A:A1 F:00 BC:A149 DE:0000 HL:0018 IX:0000 SP:0000 PC:0009
-->  43.75us A:E8 F:00 BC:A149 DE:0000 HL:0018 IX:0000 SP:0000 PC:000A
-->  45.50us A:80 F:00 BC:A149 DE:0000 HL:0018 IX:0000 SP:0000 PC:000C
-->  47.25us A:80 F:00 BC:A149 DE:0000 HL:0018 IX:0000 SP:0000 PC:000E
-->  48.25us A:A1 F:00 BC:A149 DE:0000 HL:0018 IX:0000 SP:0000 PC:000F
-->  49.25us A:A1 F:00 BC:A1A1 DE:0000 HL:0018 IX:0000 SP:0000 PC:0010
-->  51.00us A:21 F:00 BC:A1A1 DE:0000 HL:0018 IX:0000 SP:0000 PC:0012
```

```
--> 52.75us A:21 F:00 BC:A1A1 DE:0000 HL:0018 IX:0000 SP:0000 PC:0013
--> 53.75us A:21 F:00 BC:A1A1 DE:0000 HL:0019 IX:0000 SP:0000 PC:0014
--> 55.50us A:21 F:00 BC:A1A1 DE:0000 HL:0019 IX:0000 SP:0000 PC:0005
--> 58.75us A:10 F:00 BC:A1A1 DE:0000 HL:0019 IX:0000 SP:0000 PC:0008
--> 59.75us A:10 F:00 BC:10A1 DE:0000 HL:0019 IX:0000 SP:0000 PC:0009
--> 60.00us
**WHAT MEM
--> 60.00us MEMORY   0:  64
OUTPUT CONTENTS? *Y
0000 is 21
0001 is 16
0003 is 0E
0005 is 3A
0007 is 10
0008 is 47
0009 is A9
000A is E6
000B is 80
000C is 28
000D is F7
000E is 78
000F is 4F
0010 is E6
0011 is 7F
0012 is 77
0013 is 23
0014 is 18
0015 is EF
0016 is 48
0017 is 49
0018 is 21
**EXIT

7 Garbage collections executed during 2216 ms

End of SIMULA program execution.
CPU time: 13.73 Elapsed time: 55.37

.KJOB DSKB1:WORKS6.LOG=/W/B/Z:4/VR:10/VS:1665/VL:200/VP:10/VD:P
Other jobs same PPN

%WLDAFR All files rejected by conditions DSKB1:WORKS6.LOG[55453,113213]
Job 13, User [X330ED0J]  Logged off TTY122   1946  8-Apr-79
Another job still logged in under [X330ED0J]
```

Runtime 17.64 sec; Kilocore sec: 522
Total of 336 disk blocks read, 11 written
Connect time 0 hr, 1 min, 27 sec; Total charge: $1.45

II. Example of a Functional Description

This appendix gives an example of a functional description. This particular example was chosen because it demonstrates many of the features allowed in a description.

The example is of a D-type flip flop. The flip flop has four ports, a data input, clock input, synchronous reset input, and output. The function of the element is to sample the input when the clock goes high and transfer the input to output after a propogation delay. If the reset input was high then zero is transfered to the output. The propogation delay is obtained from a parameter. The parameter set TIME is searched for a real parameter which is the propogation delay in seconds. If no parameter set exists, the default value of 10ns is used.

```
#
#       D TYPE FLIP FLOP.  PORTS ARE DATA,CLOCK,RESET(SYNCHRONOUS),
#       Q.
#
##PART## CLASS                          Chain class part.
CHAIN CLASS CDFF(MOM,ACCESS);
REF(ELEMENT)MOM;                       Reference to the element data structure.
REAL ACCESS;                           Delay time parameter.
        BEGIN
        BOOLEAN DATA;                  This saves the data during the delay.
        MSK:=8R777777777773;           These are the mask flags.
        PRO:-NEW PDFF(THIS CDFF);
        DETACH;                        Allow setup code to finish with parameters.
        DAT@CHK:=8R444404;             Octal data specifyiing ports.
        ELT@CHK:-MOM;
                CALL($CHECK);          Check the ports for compatibility.
        WHILE TRUE DO                  Infinite loop.
                BEGIN
                DETACH;
                IF PRT@DCH = 2 THEN
                        BEGIN          Ignore all but the clock.
                        IF GBIT(2,MOM) THEN
                                BEGIN  If the clock went to a one state.
                                IF PRO =/= CURRENT THEN
                                        REACTIVATE PRO AT TIME+ACCESS
                                ELSE
                                        WARNING("GLITCH!");
```

```
                        IF GBIT(3,MOM) THEN
                                    Check reset line and change the output.
                            DATA:=FALSE
                        ELSE
                            DATA:=GBIT(1,MOM);
                        END;
                    END;
                END;
        END;
##PART## CLASS                     Process part.
PROCESS CLASS PDFF(SIS);
REF(CDFF)SIS;                      Pointer to the corresponding chain CLASS.
        INSPECT SIS DO
            WHILE TRUE DO
                BEGIN
                PBIT(DATA,4,MOM);
                                    Do the output.
                PASSIVATE;
                END;
##PART## SETUP                     Simulation model initialization.
        IF $ELT.NAM = "DFF" THEN
            BEGIN
            ELT@PRM:-$ELT;         Fetch parameters.
            NAM@PRM:-COPY("TIME");
            REA@PRM[1]:=0.00000001;
            DAT@PRM:=8R501;
            CALL($FETCHPARM);      Get the parameters.
            $ELT.CHN:-NEW CDFF($ELT,REA@PRM[1]);
            CALL($FINPARM);        Finish the parameter fetching session.
            CALL($ELT.CHN);        Give control to the simulation element so it
            END                    can check its ports.
        ELSE
```

III. Summary of Commands

- ADD line ;. This command adds the line to the structure. It is flexible in that the line may refer to an element that already exists. If the element exists then undefined parts of the element will be defined by the line. Using this the user can, for example, add parameters, connect ports, or even add more ports to an existing element.

- ALL. This displays for the user the entire written representation of the interconnect structure.

- DUMP filename. Write the written representation of the entire design to DSK:filename.IL.

- ELEMENTS. All of the elements are listed, giving their label and name. This is different from a written representation as the connections are not shown, but elements such as wires, not otherwise shown, appear.

- EXIT. Exits to the monitor. A lengthy version of ↑C.

- FREEZE. This command causes the entire program to be readied for a save to a disk file. It closes terminal files, releases the high segment, saves the accumulators, etc. This can be used, for example, to save a large simulation while in progress.

- GET filename. Read the written representation of an interconnect structure from DSK:filename.IL. This command can actually edit the design. If it is used when a design already exists, if will join the two interconnection structures.

- GO time. This causes simulation to proceed for the time increment time before terminal entry is requested again.

- -. Minus is not really a command, as there is no prompt after its execution. It sets an invert flag so that the next command will have its normal function reversed. For example, the PROBE command will start tracing, but will stop tracing if preceded be a minus, e.g. -PROBE.

- PROBE identifier. (Positive version) This turns off the masking flag corresponding to port zero on the element whose label is identifier. If the element is a wire this turns on tracing for that wire.

- -PROBE identifier. (Negative version) This reverses the effect of the PROBE command.

- SIMDDT. For serious debugging the user may want to use DDT. This command calls the SIMULA DDT. The user can then examine storage, set breakpoints, etc., then continue execution.

- SIMULATE. This generates a simulation model for each element in the design that doesn't have one. This command causes checking of ports and parameters, hence may cause errors to be revealed.

- TRACE. This toggles the trace flag on the input parser. This is useful for locating errors on input.

- UNTIL time. This causes simulation to proceed until simulation time is equal to time before terminal entry is requested again.

- WHAT identifier. This does an activation of port zero on the element whose label is identifier. The setting of the mask flag is not checked.

References

[1] Barbacci, M., Barnes, G., Cattell, R., Siewiorek, D.

The ISPS Computer Description Language.
Technical Report, Computer Science Department, Carnegie-Mellon University, March 1978.

[2] Case, G., Stauffer, J.
SALOGS-IV A Program to Perform Logic Simulation and Fault Diagnosis.
In *Proceedings of the No. 15 Design Automation Conference*, pages 392-397. IEEE/ACM, June, 1978.

[3] Chen, R., Coffman, J.
Multi-Sim, A Dynamic Multi-Level Simulator.
In *Proceedings of the No. 15 Design Automation Conference*, pages 386-391. IEEE/ACM, June, 1978.

[4] Ciampi, P., Donovan, A., Nash, J.
Control and Integration of a CAD Database.
In *Proceedings of the No. 13 Design Automation Conference*, pages 285-289. IEEE/ACM, June, 1976.

[5] McWilliams, T., Widdoes, L.
SCALD: Structured Computer-Aided Logic Design.
In *Proceedings of the No. 15 Design Automation Conference*, pages

271-277. IEEE/ACM, June, 1978.

[6] Birtwistle, G., Enderin, L., Ohlin, M., Palme, J.

DECSYSTEM-10 SIMULA Language Handbook, parts I, II, and III.
Technical Report, Swedish National Defence Research Institute and the Norwegian Computing Center, 1975.

[7] Wallace, J., Parker, A.
SLIDE: An I/O Hardware Descriptive Language.
Technical Report, Electrical Engineering Department, Carnegie-Mellon University, April 1979.

[8] Wilcox, P., Rombeek, H.
F/Logic - An Interactive Fault and Logic Simulator for Digital Circuits.
In *Proceedings of the No. 13 Design Automation Conference*, pages 68-73. IEEE/ACM, June, 1976.