CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science Department

# Structure, Placement And Modelling

Technical Report # 4029

by

Richard Segal

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

February 1, 1981

**Abstract**

The nature of hierarchical design tools for VLSI implementation is explored in terms of the "Caltech Structured Design Philosophy" as interpreted by Rowson in his doctoral thesis [Rowson]. One obvious implication of this thesis is the desirability of tools for leaf and composition cell design. This thesis describes one such tool targeted at the composition cell design problem. It is intended to be used in the architectual phases of a design and allows structural (interface specification), physical (floor planing), and behavioral (simulation modelling) descriptions to be written down, integrated, and tested. One biproduct of this process is the generation of a comprehensive design document from which workbooks can be generated automatically.

The later sections describe a hierarchical simulator and how it fits into the step-wise refinement process of design. The most important considerations in the design of this simulator were ease of expression and the provision of enough generality to allow the specification of any VLSI structure. Simulation takes place in a time axis/delay environment and uses a system in which nodes may take one of four values or states. This allows a high level simulation in which physical devices are replaced by register transfer type operations. Data is altered and moved around using flow control mechanisms, logical and mathematical operations, and various means of specifying delay. Though not necessary or typical it is possible to model actual devices as ideal switches using these techniques. It is a multi-model simulation because simulation can occur at any level of design abstraction. Several examples are given including the modelling of the GR2 stack data microprocessor which was recently fabricated in NMOS.

**Table of Contents**

1) Structured VLSI Design
    1.1) Communication is the Key
    1.2) Regularity and Interconnection
    1.3) Verification
2) SPAM
3) Creating Structural Descriptions
    3.1) The Hierarchy of Representations
    3.2) Composition Rules
    3.3) Interface Specification and Floor Plan
    3.4) Automatic Documentation
4) Behavioral Modelling
    4.1) Operations
        4.1.1) Data Storage
        4.1.2) Data Transfer
        4.1.3) Node Specification
        4.1.4) Logical/Mathematical Functions
        4.1.5) Supplementary Features
    4.2) Scheduling and Communication
        4.2.1) Simulation Hierarchy
            4.2.1.1) Connections
            4.2.1.2) Wires and Included Buses
        4.2.2) Simulating Combinatorial Logic
        4.2.3) The Sequential Behavior of an Instruction Processor
5) Software Organization
6) Conclusions

Appendix A) The SPAM Notation
Appendix B) SPAM User Guide
B.1) Cell Specification
B.2) Behavioral Descriptions
B.3) Compiling Input
B.4) Running a Simulation (MAPS)
Appendix C) Examples
C.1) A Miniature OM Chip
C.2) The GR2
C.2.1) A Sequential Description
C.2.2) A Bigger Picture

## 1) Structured VLSI Design

As VLSI devices grow smaller and smaller, designs in VLSI can grow more and more complex. A theoretically possible 10 million transistor chip presents an enormous challenge to the designer. The so called "Caltech Structured Design Philosophy" has yielded an approach to this problem which seems to make large designs more feasible and understandable. This master's thesis is a description of a design system which attempts to follow this philosophy and to provide a means for verification of large designs.

### 1.1) Communication is the Key

It has been established [Mead and Conway, Rowson] that the primary factor influencing the size and speed of VLSI implementations is communication. Putting functionality and transistors into chips is easier, less critical, and more well understood then the usually random wiring that goes in between all the components.

### 1.2) Regularity and Interconnection

Random wiring can be reduced by designing modules which when placed side by side connect to one another exactly as they should. This technique increases regularity in designs. These so called "butting blocks" result in the production of many rectangular cells which fit together in a design with no other interconnection. The communication problem has not disappeared however, it has just become easier. Every design needs some sort of random wiring. In a "butting blocks" world random wires are put into cells of their own, making them easier to define and to modify. A design which is made completely of butting blocks must be planar and can be submitted to automatic stretching and placement algorithms.

### 1.3) Verification

To be sure that a large chip will function as it is supposed to it is important to have some tools for the verification of designs on the structural, physical, and behavioral domains of description. Structural integrity of a design which is made of only rectangular cells is equivalent to checking that all connections between all cells can

be made and that they are all of legal types (eg. input to output etc.). Physical integrity applies primarily to "leaf" cells which, when a design is complete, have physical descriptions (eg. shapes on silicon) and must be placed side by side on the chip. Elements of physical verification include placements and stretchings of leaf cells and design rule checking. Behavioral verification usually implies some sort of simulation of designs. Simulators have been designed in many different ways. One of the primary tasks of this study has been the building of a hierarchical simulator in which the logical function of more abstract descriptions can be compared with that of more concrete cells as a chip design progresses. This technique should be understood in the light of a discussion of the nature of the design hierarchy; therefore the complete subject of simulation is left for later sections.

## 2) SPAM

SPAM is the result of a synthesis of two spheres of thought. Gray and Buchanan worked together on the topic of leaf cell verification in an attempt to help unify the structural and physical domains of description in chip design [Buchanan and Gray]. One result of this work was the invention of the coordinode. The coordinode is an object which can be used in the description of a chip to refer to a physical or structural position. The value of this object is to allow for the production of structural and physical descriptions using one set of constructs. This keeps the descriptions isomorphic and, as in SPAM itself, allows physical characteristics to be implied from what appears to be purely structural description. The final result of this work was the production of the ICSYS program. ICSYS is a design tool which allows embedded definitions to be translated to a structure that can be used for performing design rule checking, circuit level simulation, and providing fabrication output.

The second influence was the Caltech hierarchical design philosophy. Given that a "leaf cell" system had been implemented, there was now room to create a program that allowed designers to produce structural and behavioral descriptions of their design hierarchies. The hope was that such a program could be made to output chip assembler code thus producing a system capable of providing all three levels of verification. Thus, SPAM, which stands for Structure, Placement, and Modelling, was born as a composition cell description, floor planing, and simulation tool. It was the task of the CAD course at Caltech to define the structural description language for SPAM. The author joined this group eventually under the guise of designing the behavioral language and simulator. All programs were written in the SIMULA language on the Caltech DEC-system 20. Here belong thanks to the CAD class which consisted of teacher Dr. John Gray and students Mike Sprietzer, Telle Whitney, and Ricky Mosteller.

## 3) Creating Structural Descriptions

The central thesis of the VLSI structured design philosophy has been that by combining the approach described in section 1 with a top-down design process generating a hierarchy of representations for any given chip a complex VLSI structure can be efficiently produced. This approach tends to produce highly regular or ordered placements of modules on chips and tends to decrease the wiring problem. SPAM is a system for describing the design hierarchy. No actual physical implementation of cells is assumed. Since the hierarchy is "separated" from the actual physical domain allowing the use of any actual leaf cells, (eg. NMOS, CMOS or graphics) Rowson's term "separated hierarchy" is reasonable to describe the purely structural function of SPAM.

### 3.1) The Hierarchy of Representations

At the "bottom" of a design hierarchy are the so-called "leaf cells." To keep with the structure and placement algorithms previously designed [Rowson, Kingsley] physical descriptions of leaf cells are contained in a rectangular bounding box. At the "top" of such a hierarchy is the single definition describing the entire chip. In between those two representations are the so-called "composition cells." That is, all non-leaf cells are just some sort of "composition" of "lower" leaf and composition cells. Naturally, composition cells also possess rectangular shapes.

Whether a design is thought of as "bottom-up" or "top-down" the designer will deal with an abstraction of the units of his/her system in order to build the next larger system rather then the complete description of each of the components [Rowson]. Once an abstraction is decided upon it is possible to build the individual units or subsystems of one's design. It is in this process of abstraction and refinement that design decisions become fixed. It is for this reason that it seemed wise to attempt to provide whatever verification we could, be it structural, physical, or behavioral at that stage in the process between levels of abstraction. This is in contrast to a system which perhaps produces a simulation by looking only at the leaf cells of the completed design. It is hoped that this will allow for the detection of more mistakes at a time closer to when they were originally produced.

### 3.2) Composition Rules

In order to provide any real tools to a designer the "composition rules" which describe how low level cells are combined to make new composition cells must be discovered. The task is to provide the user with some way of producing a cell definition which is to be considered as the sum of several parts. The designer has already produced some cells. S/he has completely designed a leaf cell which s/he would like to use with some specific combination of the other leaf cells also already designed. In other words, the definition which is to be expanded was a leaf cell with no components. New leaf cells will be added to this definition to make up the components of the new composition cell. In the following section this process is more completely examined as leaf cells, composition cells, and the rules that relate them are precisely defined.

### 3.3) Interface Specification and Floor Plan

The primary ingredients in a circuit layout are wires. Wires pass from module to module as described earlier by abutting blocks. Therefore, any cell definition must have some connections to the outside world. From here on these connections will be called "pins."

The components of a composition cell can be said to be instances of the definitions of those components. Each of those definitions must have had pins too. In a complete design there will be an arbitrary number of composition cells in an arbitrarily deep hierarchy. If we are to be able to identify individual nodes in the circuit there must be some way to connect all these components into some sort of real data structure which looks like a real hierarchy. For example, a pin on the outside of the top level cell must somehow connect to some real component which was defined at a later date. So, the description of cells must contain enough information to compose the entire hierarchy and all connections between all pins on all instances of all definitions. The result is called a "fully instantiated" hierarchy.

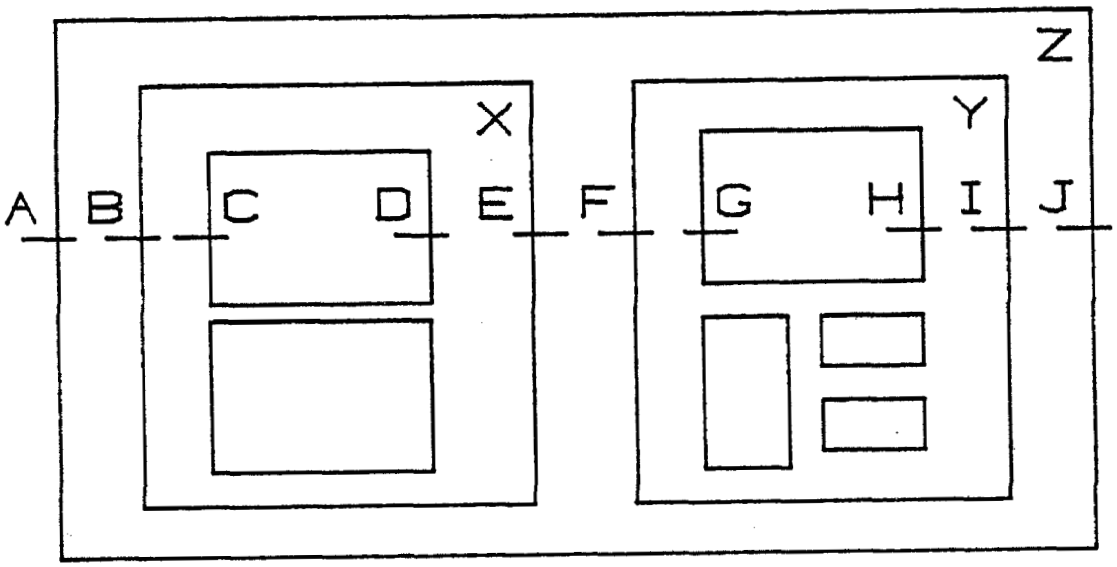



Figure 3.1

When user defines Z, E would connect to F but D is not connected to E until X is defined and G is not connected to F until Y is defined. If we are interested in how X and Y's components communicate it must be known that D connects to G.

Every cell definition starts out as a leaf cell and is then expanded to a composition cell. So let us first consider leaf cells. First off, every cell needs a name. Now, if a leaf cell is to be used as a component for another cell we must at least know what pins it has. We need a naming convention and a method for pin declaration. Another useful piece of information is the pin type. The value of "typing" is to allow a more complete structural verification. When the hierarchy is instantiated every connection of two pins can be checked for legality. SPAM provides the six types: input, output, io, power, ground and clock to provide a sufficient set of types to catch most illegal connections. This makes the task of describing circuit "behavior" somewhat easier and improves design documentation through annotation. Since connections contain just two wires, a simple set of legal connections can be produced:

```
 Input <-> Output
    IO <-> (Anything)
 Clock <-> Clock
 Power <-> Power
Ground <-> Ground
```

Since individual physical devices are not important in SPAM the task of designing the structure of a design is primarily the task of "interface specification." That is, we are searching for the organization of communication between modules. The three items: cell name, pin declaration, and pin typing are all the parameters necessary to completely describe the structure of one individual leaf cell.

The next step is to use that specification and produce a complete layout. For this it is necessary to produce a "floor plan." A floor plan is a description of the physical positioning of instances of cells in relation to one another. This is the descriptive task which ties the composition cell together.

To begin with, each composition cell needs to include a list of all components. This list will show the correspondence between "real" cell definitions and the instance names they are given for use as components of this cell. A cell may have any number of repetitions of any component. Therefore, it is useful to allow the user to

specify the inclusion of arrays of components. That is, we need to be able to specify something like this: "I have an 8x8 array of ADDER cells called ADD[1,1] to ADD[8,8]." Once the complete list of components is declared the only remaining task (besides modelling - see section 4) is to describe the abutment of those components. Cells abut only along horizontal and vertical edges. A description of the floor plan of a composition cell can now be created by simply listing the abutments of sets of components edges against each other and against the sides of the composition cell itself. Notice, no pins are ever mentioned in abutment declarations. Only edges are referenced. For example, "the left of X abuts the right of Y" tells us that all the pins on the left of X connect to all the pins on the right of Y. Notice too that since actual leaf cells are unknown and may require stretching and placement before they can actually be hooked up, we have created a technique in which a purely structural description is given and from which a physical positioning can be implied.

SAMPLE.TXT by Richard Segal 1980-10-14

Floor Plan of 'SAMPLE':

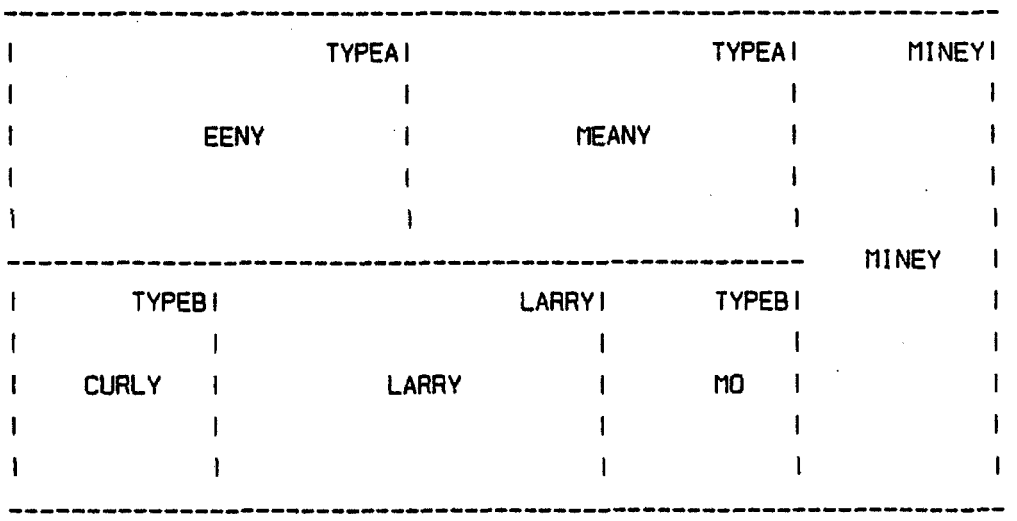



**Figure 3.2**

Sample SPAM floor plan output to demonstrate that pin names not explicitly specified in abutment list

This description will be complete as long as there are an equal number of pins along each set of edges which abut. However, since it is occasionally desired that some pins be left unconnected this condition may not always hold. For this reason an "omit" clause must be added to abutment declarations. For example, we must be able to say "the left of X OMIT PINP abuts the right of Y" where the left of X was declared to have one more pin the we could have actually connected to the right of Y.

These are the elements of a structural/physical design in SPAM. Every composition cell has a name, a set of pins and types, and a set of components and abutments. Given this, the notion is that a SPAM compilation can now perform design checking and structural verification tasks. One primary task is to go through all the abutment statements and attempt to produce a data structure representing all the connections which are implicit in them. There is only one correct connection of the components in a cell. Equivalent to the task of checking to see if such a legal set of interconnections exists for each cell is the determination that the graph of the connections is "planar." That is, all connections can be made without crossing wires. This is a relatively simple task and a SPAM compilation will report any structural errors at this point. It will also check that any connections that are made are of one of the legal types outlined above.

If a cell is planar and all connections are of legal types, then one's trust in the structural integrity of the cell can be increased greatly. As the design progresses and each level of definition is checked, more potential errors should have been removed. By the time the design is complete, assuming the leaf cells are correctly represented, one can feel confident that all the nodes in one's design are connected in legal ways to one another. Since most errors made in VLSI design involve wiring and structural integrity, it is hoped that much can be gained from the use of this type of design system.

### 3.4) Automatic Documentation

Included in the SPAM compiler is the generation of the following information:

1) Hierarchy Composition

2) Interface Specification

3) Floor Plan Organization

Since all three of these pieces of information are of vital importance to the description of a circuit, any documentation provided by the SPAM system will be of great value to the designers, managers, and future users or modifiers of the specified circuit. Therefore, once a circuit description has been compiled, the user may request that such automatic documentation be produced. The result is the production of a hierarchical map for the entire circuit, an interface specification diagram for each cell definition, and a floor plan diagram for each composition cell in the descripton. Examples of all three of these appear in appendix C.

## 4) Behavioral Simulation

Until it is possible to "compile" function into designs in a general way it will be necessary to use simulation to assist in the verification of behavioral integrity. Simulation has taken many forms. They include detailed low level circuit element simulation, switch level transistor network simulation, as well as several flavors of logic or register transfer simulation.

Detailed circuit element simulation is of very limited value in VLSI implementation. Even though these simulators often involve very complex mechanisms they still rely on the specification of all the process parameters of individual fabrications. These are rarely known with precision. The result is the expending of much effort for little practical gain. If precise delay estimation of circuit behavior is required it is usually easier and just as valuable to do paper and pencil tabulations. Such techniques are fully explained in text books [Mead and Conway]. Furthermore, VLSI circuits are much too large to even attempt to perform detailed simulations of entire chip designs even with the largest modern computers.

Switch level simulation has been used extensively in LSI design. It has the advantage of requiring a relatively small amount of storage and the ability to produce quick results. Unfortunately these simulators do not allow complex testing in time. The primary value of simple switch simulators is the verification of the logical function of small circuit elements whose correct behavior is well defined. No attempt is made to model complex communication enviornments or to relate circuit behavior to high level logical function.

The final category is the high level, time delay or register transfer level of simulation. Generally these are simulators which require the users to describe their circuits in a specially designed input language. These languages allow the user to specify function of modules in terms of "nodes," "pins," "registers" or whatever blocks the particular system is built of. The simulator reads these descriptions and simulates the network of modules as a connected design. Since structural descriptions are new, this is generally accomplished by specific reference to neighboring modules in the behavioral description. [Ko, Cory et al] The primary advantage of this level of simulation is the ability to simulate complex structures in a simple, compact way. That is, objects such as registers,

adders, and gates are usually included in the semantics of the simulator. The result should be the ability to describe entire circuits with a minimum of effort.

The SPAM simulator is of the last type. The task of this thesis is to discover what it takes to build a good high level simulator. To begin with we have been given the framework of SPAM. That is, a hierarchical description of a chip exists and we desire to add the ability to describe the behavior of cells.

This framework indicates that simulations will be described and performed in a top-down manner. That is, an initial chip description will be produced and simulated. Once the results are correct, the design is broken up into components and each module is given a behavior. The resulting network is simulated and the results are compared to the original test. If the same results are produced then this refinement has been successful. The process continues until the "leaf cells" have been described.

Some of the features required by the SPAM behavioral description language are:

1) Ability to reference "pins" of the SPAM description.
2) Ability to simulate to an arbitrary depth in the SPAM hierarchy.
3) Compact descriptions
4) General descriptions

There are two parts to this exploration: 1) What circuit operations should be simulated, and 2) How is communication between modules and module activation (scheduling) accomplished? The result is that SPAM includes all the features described in "Operations" and both scheduling mechanisms described in "Scheduling and Communication."

### 4.1) Operations

The desiderata of both compact descriptions and generality can often produce conflict. If a description language is too general it may require the specification of a lot of detail. On the other hand if a language is too simple serious limitations may be placed on the types of operations which can be easily simulated. In order to decide what the behavioral language should look like let us start by considering what operations do actually take place on chips. It should be noted that what is

being presented is a description of what is necessary. The actual formats chosen appear in the appendices of this report.

### 4.1.1) Data Storage

Data must be communicated between cells through pins. Naturally, it must be possible to reference data residing at the pins. The most common operation in integrated circuits is data storage. To allow for this additional type of node, the "internal storage" node is defined. Since their real counterparts usually come in sets, like registers and arrays, it must be possible to declare arbitrary arrays of internal storage.

### 4.1.2) Data Transfer

Operations on data can now take place. First of all some facility must be provided to transfer data from one set of nodes (pins or internal storage) to another. Since data transfers really take place on wires or through networks all of which have real delays, this mechanism must have the ability to specify the delay to be modelled in each such transfer. Since the operation here is to "set" the "value" of nodes it is called a "setval" statement.

### 4.1.3) Node Specification

Setval statements are the place at which operations are most easily specified. For example, we will want to say something like, "register A bits 1 thru 4 get the result of an AND operation on registers B and C." Clearly the setval operation looks like an assignment statement where the left side specifies the destination and the right side specifies the sources, operations, and delay to be used. Here is a place to make a choice. Is it better to allow very general node specification in this assignment statement risking some complexity in the parser and in the BNF for the language or should only simple register transfer operations be allowed? One example which indicates the need for a very general referencing scheme is the interlaced bus of the OM chip [Mead and Conway]. This is a set of 16 wires (which will look like "pins" in SPAM) which are to be used as two buses. A creative wiring strategy interlaces the buses. That is, all the even numbered wires are called the A bus while all the odd numbered wires are called the B bus. If only straight sets of nodes could be referenced in the setval statement it would probably be necessary to include 16

different such statements for each operation required. Another example is an adder in which the overflow bit is to be gated to a special error flag register. This indicates the need to allow both multiple destinations and multiple sources in the setval statement. Quite clearly this is one situation where more generality will mean less confusion. The reason is the variety of structures that can be built on a chip. It is concluded then that setval statements should include arbitrary references into sets or arrays of nodes and should allow the concatenation of sources and destinations in any given operation.

### 4.1.4) Logical and Mathematical Functions

The setval statement generally represents the transfer of data between nodes. To completely specify the function of such a transfer we need both logical bit functions such as AND, OR, and NOT as well as mathematical functions to allow modelling of arithmetic components like adders or multipliers. Bit operations act on binary words and we generally think of arithmetic in terms of decimal numbers. Since the binary nodes may be referenced in an arbitrary manner there must be some facility to allow sets of nodes to be used as either logical or arithmetic values. Consider the following example: "register X gets (NOT register A plus register B) divided by 19." This demonstrates the type of activity that must be allowed. We must be able to invert the bits of register A; add the value of the result to the value contained in register B and divide that result by the decimal value 19. The final result must be placed in register X which may be of an arbitrary length requiring pading with zeros or truncation. Notice that arithmetic must always be decimal arithmetic. Nevertheless, we could simulate a 2's complement subtraction of 'A-B' by 'A + ((NOT B) + 1).' All mathematical operations available in SIMULA have been included in SPAM, and all SIMULA logical functions are implemented as operations on SPAM node sets.

### 4.1.5) Supplementary Features

Since chips can perform any logical function it is necessary to provide a few programming constructs as flow control mechanisms. In SPAM, the SIMULA FOR, WHILE, and IF-THEN-ELSE constructs are provided. The additional word 'EF' as an abbreviation for 'ELSE IF' is added. Generally, SIMULA arithmetic expressions are replaced by SPAM arithmetic expressions as described in the context of setval statements (section 4.1.4). Boolean expressions for use in IF and WHILE statements

different such statements for each operation required. Another example is an adder in which the overflow bit is to be gated to a special error flag register. This indicates the need to allow both multiple destinations and multiple sources in the setval statement. Quite clearly this is one situation where more generality will mean less confusion. The reason is the variety of structures that can be built on a chip. It is concluded then that setval statements should include arbitrary references into sets or arrays of nodes and should allow the concatenation of sources and destinations in any given operation.

### 4.1.4) Logical and Mathematical Functions

The setval statement generally represents the transfer of data between nodes. To completely specify the function of such a transfer we need both logical bit functions such as AND, OR, and NOT as well as mathematical functions to allow modelling of arithmetic components like adders or multipliers. Bit operations act on binary words and we generally think of arithmetic in terms of decimal numbers. Since the binary nodes may be referenced in an arbitrary manner there must be some facility to allow sets of nodes to be used as either logical or arithmetic values. Consider the following example: "register X gets (NOT register A plus register B) divided by 19." This demonstrates the type of activity that must be allowed. We must be able to invert the bits of register A; add the value of the result to the value contained in register B and divide that result by the decimal value 19. The final result must be placed in register X which may be of an arbitrary length requiring pading with zeros or truncation. Notice that arithmetic must always be decimal arithmetic. Nevertheless, we could simulate a 2's complement subtraction of 'A-B' by 'A + ((NOT B) + 1).' All mathematical operations available in SIMULA have been included in SPAM, and all SIMULA logical functions are implemented as operations on SPAM node sets.

### 4.1.5) Supplementary Features

Since chips can perform any logical function it is necessary to provide a few programming constructs as flow control mechanisms. In SPAM, the SIMULA FOR, WHILE, and IF-THEN-ELSE constructs are provided. The additional word 'EF' as an abbreviation for 'ELSE IF' is added. Generally, SIMULA arithmetic expressions are replaced by SPAM arithmetic expressions as described in the context of setval statements (section 4.1.4). Boolean expressions for use in IF and WHILE statements

may include any legal logical combinations of boolean variables and conditions where a condition is a pair of SPAM arithmetic expressions separated by any SIMULA boolean operator (eg. =, <, \=).

The "NEXT" statement also influences flow of control. It is used to assist in the simulating of microprocessors or other instruction processors and will be explained in that context in section 4.2.3.

## 4.2) Scheduling and Communication

Once SPAM has parsed the behavioral descriptions of a circuit the output code (in SIMULA) is produced. This resulting program contains a brand new data structure and mechanisms needed to actually simulate the circuit.

### 4.2.1) Simulation Hierarchy

The SPAM design hierarchy no longer exists in the actual simulation. The designer has specified that certain cells should be treated as leaf cells for this particular simulation. Therefore, only those cells are used to produce the simulation of interest. The result is a new data structure which can be thought of as one complete tiling of the chip (or component) being simulated. For example, the designer may have specified that the left half of the circuit be simulated by a single cell definition which is described near the top of the hierarchy, while the right half is to be simulated by a dozen lower level cells which were described at a different time.

#### 4.2.1.1) Connections

Type checking and planarity do not require a fully instantiated hierarchy. That is, if all local connections are legal then all global connections will be legal. However, to turn one specific set of cells into a connected communicating network requires that all paths of communication between them be discovered.

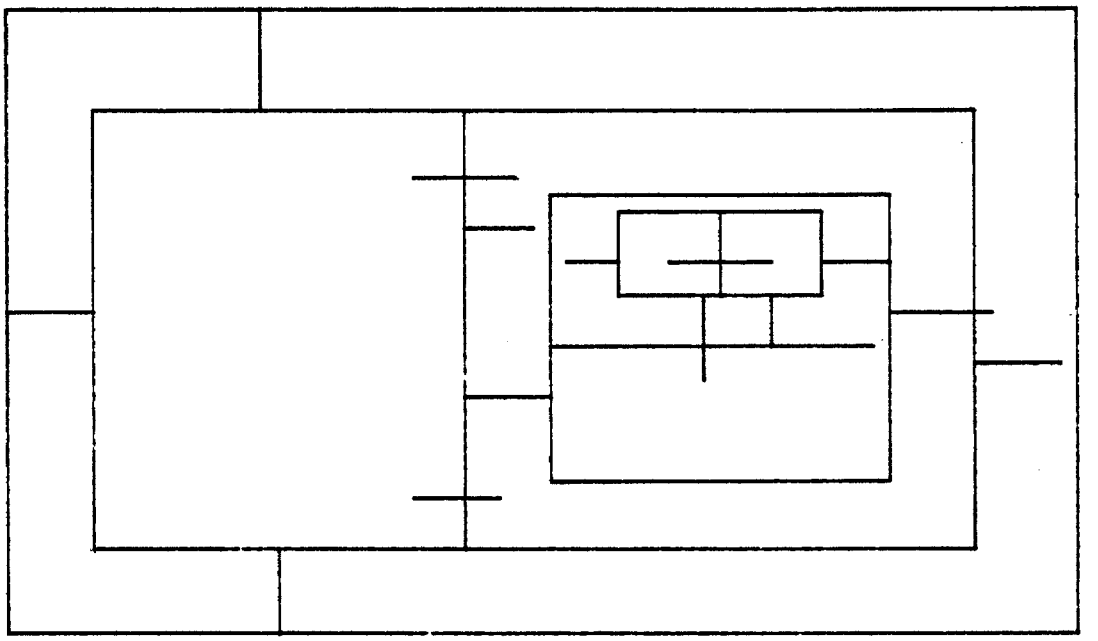


Figure 4(a)
Uninstantiated Hierarchy

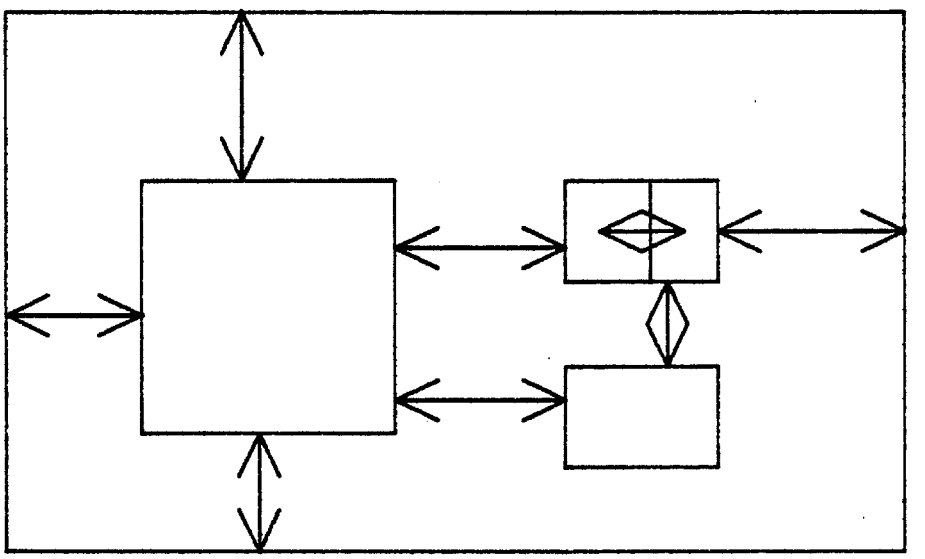


Figure 4(b)
Simulation Data Structure

In diagram 4 an example hierarchy is shown with several connections. Figure 4(a) represents the SPAM data structure and all known connections are shown. Assuming the lowest level cells shown are to be simulated the resulting simulation data structure is shown in figure 4(b).

Strictly speaking this is not a completely instantiated hierarchy because it contains only the information required to run a simulation. That is, no table is actually created which keeps track of all nodes which are electrically equivalent. The fact that one node is connected to another is found instead by following pointers and searching for the correct pin in the appropriate enclosing or abutting cell. The simulation data structure (with its limited information) appears in the output code of the compiler. The result is, that though a fully instantiated hierarchy is required, it need not ever actually exist. (See section 5 for an explanation of the functional organization of SPAM software.)

Because circuit behavior is in terms of data movement and operations on data with delay, at their borders, each individual definition will look like a finite state machine or some combinational network with delay. That is, changes in inputs will produce changes in outputs in time. In order to turn a combination of these cells into one single circuit we need to provide some mechanism to transfer data across the connections. As far as the data structure is concerned this implies only that each pin in the data structure of cells be given a single state value (0, 1, X, U) and a single pointer to the pin with which it is to be communicating. Every time some of the external data on some cell is to be changed a global "Wake-Up" procedure will be called. This procedure will be responsible for actually changing the data and waking up the neighboring cell.

#### 4.2.1.2) Wires and Included Buses

One common practice in designing cells to be placed on integrated circuits is to include a wire which has no function other then to provide a signal to this cell or to cells on either side of this cell. The most common form of this structure are power and ground buses which pass through many cells uninterrupted. Another example is the bus of a microprocessor which carries data back and forth between many components of the device. In terms of SPAM simulation cells this introduces a slight complication. Usually when a cell puts new data on its external pins this

data will be transmitted (through the "Wake-Up" procedure) to the pin it is connected to. When the receiving cell gets this data it can decide what changes it then needs to make. In the case of wires however, no behavior has been provided specifying that the signal should be propagated to the other side of the cell. One simple solution to this problem is to notice when a cell has more then one pin with the same name. Then, all such pins can be considered as a single node. The Wake-Up procedure can then transmit the new data to a list of successor pins in addition to the single connection already known. That is, while most pins will have just one pointer to one other pin, pins which are on wires will point to as many pins as there are ends to the wire in that cell. In this way a single change on one pin can directly cause changes to many cells by being propagated through the wires of those cells. Notice that the Wake-Up procedure must be smart enough not to propagate the signal to pins which already have the same value as that being propagated. This avoids circular and backwards propagations as well as eliminating redundant signal changes.

### 4.2.2) Simulating Combinatorial Logic

Once the above data structure is in place it is a relatively simple matter to actually run a simulation. However, before anything can happen a few runtime facilities are needed. These facilities include a decimal to binary and binary to decimal converter; logical operations for lists of nodes (random words) such as AND, NOT, and EQV; SIMULA class definitions for cells and nodes; connection mechanisms; a facility to allow the user to set and query node values, to set break points, and to clock nodes at regular intervals; as well as the Wake-Up procedure alluded to earlier. Once these are in place the user can set initial conditions in the circuit (the undefined condition 'U' is assumed for all uninitialized nodes) and start the simulation. From then on simulation proceeds as follows. A cell wishes to produce some changes on some of its nodes after a specified delay. This is accomplished by scheduling a "Wake-Up" procedure to become active after that specified delay period. The parameters to the Wake-Up are simply a list of nodes to be changed and values to place there. When the delay period is up SIMULA will automaticaly schedule the Wake-Up. Then for each node in the list which actually needs changing, the Wake-Up will place the new value in each of the nodes to which the input nodes are connected. It will then schedule the cells of all those nodes to become active immediately. Finally, it will schedule a new Wake-Up procedure for all nodes connected to each destination node via wires. These last nodes may be scheduled with some delay to simulate wire delay. This process simply continues with break points interspersed as specified by the user. The result is a data structure which acts like a combinatorial network. That is, all the individual cells, specified to act like combinatorial systems, now communicate with one another in a delay / time axis enviornment to produce a larger network of behavior in such a way as to simulate a combinatorial system.

### 4.2.3) The Sequential Behavior of an Instruction Processor

The preceeding description of the normal operation of the SPAM simulator is satisfactory to describe any combinatorial system. It is interesting however, that not all systems are easily thought of as combinatorial. An example of this is the microprocessor. While combinatorial systems can be said to be "data driven" microprocessors are usually thought of as being "clock driven." That is, depending on the current state of the system, including the value of the instruction register

and other state information, different activities will take place on the chip each clock cycle. Many instructions will take more then one clock cycle and cannot be easily described in terms of the purely combinatorial nature of the system. At the least, it would take some very difficult rethinking of one's circuit and probably a great deal of repetition to accomplish that descriptive task. On the other hand, there is usually a very complete description of the behavior of the processor residing in the microcoded instructions which actually organize the behavior of the combinatorial components to begin with.

It is desirable to provide some alternative simulation technique to allow modelling of the process of instruction fetch, execution, and loop which takes place in any instruction processing machine. It would be nice if there was some way of transforming one's microcode directly into SPAM behavioral descriptions. To accomplish this the usual scheduling mechanism must first be eliminated. The Wake-Up procedure is now responsible for changing the data on the inputs of the cell, but not for activating the module. Cells which are to be modelled as sequential processes rather then combinatorial systems still exhibit the external behavior of inputs and outputs. Therefore, there is no problem in interfacing these cells with other cells in one's circuit.

A Microprogram is a sequential program each step of which is activated by a system clock. Conceivably this activation signal could be any external signal. All that is neccesary to allow microcode-like simulation to take place then, is that an instruction be added to the SPAM repetoire which halts simulation of this cell until some specified signal makes the appropriate transition (positive or negative). In SPAM this instruction is called the "NEXT" instruction meaning wait for the "next" occurrence of the specified signal. In this way one can describe the behavior of a cell as an infinite loop which simply decodes the instruction register and conditionally executes the appropriate instruction code. Each instruction segment is made up of register transfers or operations or whatever and may take an arbitrary number of cycles. Since microcode refers to actual elements of one's circuit and the tranfers and operations which are actually available in the processor, translation between microcode and SPAM behavioral syntax should be relatively simple. This technique greatly simplifies the task of writing behavioral descriptions for instruction processing systems. Notice, the SPAM "next" statement differs from the ISP notation [Bell and Newell] in that this "next" requires a specific signal to trigger the following section of code.

Delays will still be modelled in the same way. That is, a Wake-Up procedure will be scheduled after a specified time interval causing the data to be changed. Ordinarily, any cell modelled as "sequential" rather then normal will not be activated by the Wake-Up. However, if any of the signals being altered are control signals specified in "NEXT" statements and the transitions are in the right directions then each such occurrence must cause the appropriate cell to be activated. Simulation continues just like before and the user can set whatever break points one requires allowing questions such as, "Did all operations complete before the finish of the clock cycle?" or "Was the correct instruction executed?" Other cells connected to the sequential cell will be activated in the usual combinatorial way since the Wake-Up procedure can easily check to see if a cell is one type or the other.

## 5) Software Organization

Figure 5.1 below shows the processes involved in using SPAM. The original SPAM description is fed to the SPAM compiler. On request, the compiler produces two different sets of output. One of these is the documentation file as described in section 3.4. The other is a SIMULA file which is then compiled to produce a simulation of the described circuit. Once the simulation file has been compiled into executable form it may be invoked by the user directly (the SPAM compiler is not needed to run simulations).

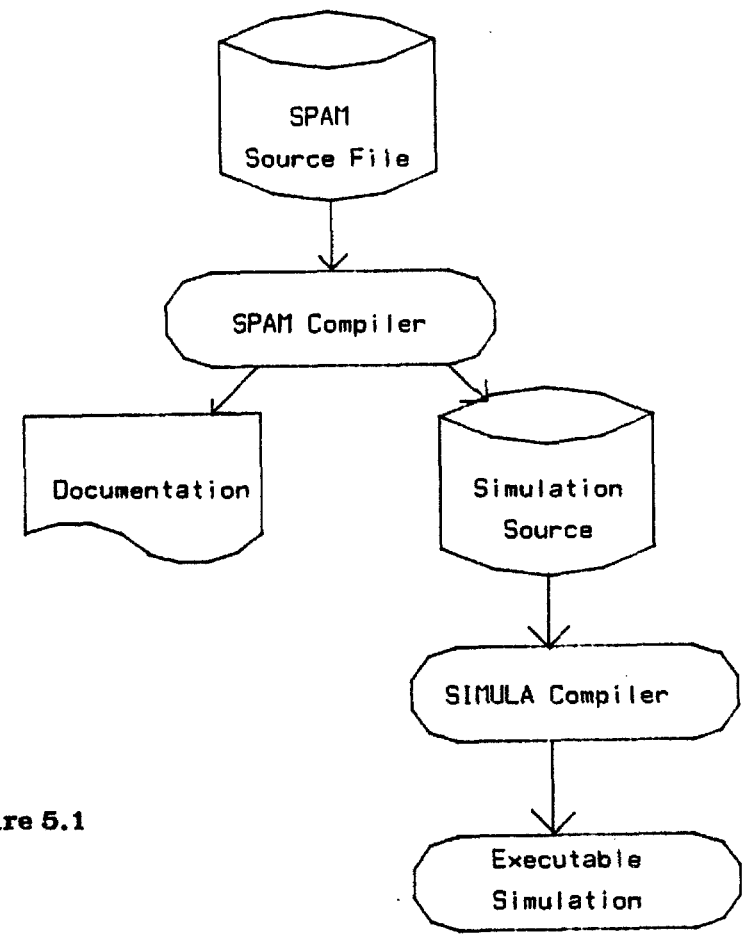



**Figure 5.1**

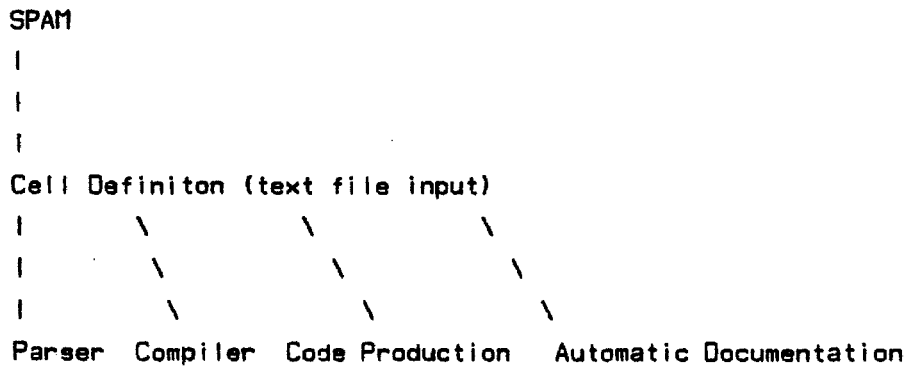
The SPAM program itself is divided into four major parts. They are diagramed in figure 5.2. The parser is responible for recognizing input forms and activating routines in the compiler to keep track of actual data. The compiler contains the actual hierarchical data structure and all the routines for modification of that structure. The parser continues to read data and communicate with the compiler until a complete set of input is read.

At that time the user may request that an output file be produced. This will cause simulation code to be written into a text file. The code includes the simulation data structure, initialization routines and the "maps" described in previous sections. SIMULA compilation of this file results in an executable program (phase 2) containing all the features of a simulation as described. The simulation will be of the circuit described in the original input file.

The final element of phase 1 is the automatic documentation production system. The user may request that an output file be produced for the input design. This file contains a structure map, cell specifications for every cell definition, and floor plan diagrams for all composition cells (see section 3.4). The input text file name, date, and designer name is included on each page of this document to provide some organizaton and to avoid confusion between successive iterations of designs.

Complete specifications of the input language for structure, behavior, and simulation execution are given in the appendices. Examples of all these as well as of the automatic documentation feature are also included.

**Phase 1:**




**Phase 2:**

Output Code + MAPS = Simulation Code

**Figure 5.2**

## 6) Conclusions

The reasons for designing a chip using SPAM should include the following: SPAM forces the designer to clearly define chip components including their external interfaces, exact behavior, and wiring strategy. By checking connectivity, SPAM increases the designer's faith in the integrity of interface specifications. By providing simulation, the step-wise refinements of one's chip can be checked for logical integrity exposing errors very early in the design process. The design process itself can become more flexible and open to change. The system can create some meaningful documentation for user's cells which might otherwise not have been produced. The value of this is the automatic production of workbooks or project reports. The notation is compact. SPAM descriptions are easy to produce since they are made in parallel with the design process. The existence of a separated hierarchy allows easy switching between leaf cell implementations. The job of designing leaf cells might be made easier by using floor planning information in SPAM to perform automatic cell stretching. Future versions of SPAM might include silicon compiling or interfaces to other low level chip assembly tools such as ICSYS [Buchanan and Gray] or Bristle Blocks [Johannsen].

To determine which of these goals are successfully realized in the SPAM system they are reviewed one by one. Possible directions for future development are discussed.

It is true that SPAM forces formalization of one's circuit to some extent. The value of this is that by bringing details of the designer's cell specification to paper, errors are more likely to be detected faster. By forcing the specification of a wiring strategy on designers of high level components, better integrated circuit layouts are likely to be developed. Planarity and connectivity checking is certainly worthwhile since it keeps the designer in touch with the layout as it progresses by noticing when errors in interface specifications are made.

Providing a simulation tool along with the design process will be valuable as long as it is easy, requires only a relatively small amount of extra effort, and produces reasonable verification of circuits. The SPAM simulator has been tested in several of its capacities. One discovery was that it is very easy to produce top level microprocessor descriptions once a microprogram exists using the idea of sequential

simulation described in section 4.2.3. (See appendix C.2 for the example.) The only chip actually completely designed and debugged in SPAM was a miniturized data path chip described in appendix C.1. The behavior could be described in a matter of hours, and bugs in those descriptions were quite obvious when simulation was attempted. No attempt has been made to exhaustively test the chip but it is clear that even just a small amount of simulation could reveal a great deal about the nature of the circuit described. A complete answer to this question can only come when several real chips are put through the complete design process using SPAM with a wider community of users. The general idea that the presence of a hierarchical simulator should make errors more accessible and open to change is subject also to the fact that it is still up to individual designers to provide test patterns which will sufficiently test their chips. Designing for test is an important concept if testing is to be made feasible. This problem is left as the responsibility of the designer. Finally, there is a trade-off between the difficulty of using a simulator and the value of the resulting simulations. If descriptions are easy and compact enough then the small amount of extra time required may be very worthwhile.

Appropriately, the next point is that descriptions should be easy to produce and compact both in the size of the actual description as well as in the size of the resulting data base. Great care was taken first in the CAD project class in the development of the structural notation, and later by the author to develop behavioral notation, to keep the language simple and general. The structural descriptions require a minimum of information (see section 3). One possible addition to this, however, is a graphical front end which generates the structural/physical definitions of cells. It would provide a mechanism for producing component declarations and abutment lists. The combination of graphic floor planning and the compact pin naming conventions already defined will really make the structural description process simple.

The behavioral descriptions allow any register transfer, mathematical/logical, and flow control mechanism one could desire. The main feature which keeps behavioral descriptions short is the general way in which nodes can be referenced. Examples in section 4.1.3 show that much simplification can be gained by using the full power of this syntax.

To have the resulting data structure as compact as possible is one goal not yet completely realized. The largest circuit description yet compiled contained 41 instance elements (number of actual instances including repititions). Some of these cells were very large so, that this took only half of our DEC-20's 400 pages of (low segment) memory is promising. Nevertheless, the structural description parser and data base were not designed with the intention of minimizing space or time considerations. The goal was to produce a working model, an existence proof, to show that something like SPAM was really possible. The test case with 41 elements is the completely expanded version of the GR2 stack oriented microprocessor shown in appendix C.2. Anyway, to really turn SPAM into an efficient large scale tool several modifications will be desirable. These include eliminating the restriction that vectors be numbered from top to bottom and left to right as well as improving the efficiency of space utilization. This all implies the need for a second iteration of SPAM. The behavioral parser requires almost no space above the structural data structure since all output data is temporarily placed in disk files.

The simulator part of SPAM is somewhat more careful with memory usage. Two reasons are that 1) Space was a known premium in the design simulation output code and 2) As described in section 4.2.1 simulation does not require the presence of the entire hierarchy. The biggest actual simulation tried so far had 28 (leaf cell) instances. The runtime structure of this example fitted easily within the core limits of the machine.

Several layers of modifications are now present in the SPAM system. Though most of these appear in the appendices, they are not all completely described in this thesis. That is because of the time span in which this report was written. That is, the original SPAM specification spurred the writing of this thesis and since its implementation many desirable simplifications and improvements have been made in SPAM. (The formats appearing in the appendices reflect most of these improvements.) One feature mentioned nowhere else is the interface with a graphical floor planning tool as a front end for SPAM. Anyway, now that so much has been changed and improved in SPAM the code is just that much more in need of reiteration. In any case examples of what has been accomplished appear throughout the appendices.

The idea that SPAM should provide some meaningful extra documentation has been fulfilled. Upon request, SPAM will produce a hierarchical map, interface

specification diagrams for all cells, and floor plans for all composition cells.

The final categories of reasons for using SPAM all come under the heading "Future uses of SPAM." Since SPAM knows the floor plans of the designs it is given it would seem quite appropriate for SPAM to be interfaced with some tool for stretching and placement of cells. As the STICKS standard [Trimberger] comes into use there will undoubtedly be some program available to take a set of cell specifications from leaf cell design tools together with floor planning information from composition cell tools like SPAM and create a complete and implementable design. Silicon compilers will be a breed of design tools which produce as much of the design of chips as possible in an automatic fashion. The clear separation of the design process into two parts: leaf cell design and composition cell design, may be a very important step in reaching towards a world full of silicon compilation.

To finally sumarize SPAM is to do an injustice to the future. The author encourages all readers of this document to design their own integrated circuits using SPAM and draw their own conclusions. It is the thesis of this report that "Structure, Placement, and Modelling" implies a hierarchical approach to design and simulation. Through clean interface specification and wiring strategy a structural description can be developed. Through multi-model simulation behavioral integrity can be scrutinized and the behaviors of each abstraction of one's design can be compared. The author recomends the SPAM program to all designers willing to risk a try.

## Appendix A) The SPAM Notation

The following is a BNF description of the SPAM structural, physical, and behavioral notations. An explanatory approach to the language can be found in appendix B which is the SPAM User's Guide.

BNF symbols: ::= | { } * + [ ]
All other symbols are terminals in the grammar.

Cell Specification

```
<description>          ::= <cell header> {<S&P part>} <behave part> <cell end>

<Cell header>          ::= CELLDEF <name> (<cell con spec>); <t spec list>

<S&P part>             ::= <cell declarations> <cell body>

<cell declarations>    ::= <declaration> +

<declaration>          ::= COMPONENT <component list> ;

<cell body>            ::= <cell statement> +

<cell statement>       ::= <con exp>;

<cell end>             ::= ENDDEF ;
```

Cell Interface Specification

```
<cell con spec>        ::= <side spec> *

<side spec>            ::= <side> <con list>
```

```
<side>              ::= TOP | BOTTOM | LEFT | RIGHT

<con list>          ::= <con list element>
                        | { <con list element> , <con list> }

<con list element>  ::= <name> [ (<con list>) | <one dimension> ]

<one dimension>     ::= "[" <integer> "]"
```

Type Specification

```
<t spec list>       ::= <type spec> *

<type spec>         ::=  <type> <con nams list> ;

<type>              ::= POWER | GROUND | CLOCK | INPUT | OUTPUT | IO

<con nams list>     ::= <con nams> {, <con nams>}*

<con nams>          ::= { <side>(<con name list>) } | <con name>

<con name list>     ::= <con name> | { <con name> , <con name list> }

<con name>          ::= <name> [(<con name list>)]
```

Internal Components

```
<component list>    ::=<com list element> {, <com list element>}*

<com list element>  ::=<component> {.<orientation>}*

<orientation>       ::=MIRRORX | MIRRORY | ROTATE90 | ROTATE180 | ROTATE270
```

<component> ::= <name> [<two dimension>] [(<inst name>)]

<inst name> ::= <name>

<two dimension> ::= "[" <integer> , <integer> "]"

Abutment

<con exp> ::= <sidelist> <operator> <sidelist>

<operator> ::= ABUTS | <=>

<side list> ::= <side reference> {, <side reference>}*

<side reference> ::= <side> <name> [OMIT <omission>*]

<omission> ::= [<two dimension>.] <pin name>

<pin name> ::= <name> [ (<pin name>) ]

BEHAVIORAL SPECIFICATIONS

```
<behave part>       ::= <busy header> <busy declarations> <start> <busy body>

<busy header>       ::= BEHAVE NOW | BEHAVE LATER
<busy declarations> ::= [<comment>] [<sim declarations>] [<comment>]
                        [<internal declares>]
<sim declarations>  ::= [REAL | INTEGER] <simvar>;
<internal declares> ::= INTERNAL <storage list>;
<start>             ::= START | SEQUENCE
<busy body>         ::= <comment> <busy body> | <busy stmt> <busy body>
                        |<comment> | <busy stmt>
<storage list>      ::= <store> | <store><storage list>
<store>             ::= <store name> |<vector store>
<vector store>      ::= <store name> [<integer>, <integer>]
<store name>        ::= <name>
<simvar>            ::= <name>
```

BEHAVIOR STATEMENTS

```
<busy stmt>         ::= <busy exp>;
<busy exp>          ::= <for stmt> |<while stmt> |<if stmt> |<setval stmt>
                |<assignment stmt> | <next stmt> | BEGIN <busy exp list> END
<busy exp list>     ::= <busy stmt> <busy exp list> | <busy stmt>
<for stmt>          ::= FOR <simvar>:= <arithmetic exp> STEP
                        <arithmetic exp> UNTIL <arithmetic exp> DO <busy exp>
<while stmt>        ::= WHILE <boolean exp> DO <busy exp>
<if stmt>           ::= IF <if part>
<if part>           ::= <boolean exp> THEN <busy exp> [ELSE <busy exp>]
                        |<boolean exp> THEN <busy exp> EF<if part>
<setval stmt>       ::= <valued var list> := <match value list>
                        [AFTER <delay>]
<delay>             ::= <arithmetic exp>
<assignment stmt>   ::= <simvar> := <arithmetic exp>
<next stmt>         ::= NEXT [NOT] <connector name>
```

VALUED EXPRESSIONS

```
<arithmetic exp>    ::= <match value list> | <simvar>
                        | ([NOT] <arithmetic exp> <bin op> <arithmetic exp>)
                        | <arithmetic exp> <dec op> <arithmetic exp>
                        | (<arithmetic exp>)
<bin op>            ::= AND | OR | EQV | IMP
<dec op>            ::= + | - | * | /
<boolean exp>       ::= <arithmetic exp> <bool op> <arithmetic exp>
                        | (<boolean exp>) | NOT <boolean exp> |
                        | [ALL] <non-number> IN <match value list>
<bool op>           ::= = | >= | <= | > | <
<match value list>  ::= <match value><match value list> | <match value>
<match value>       ::= <valued var> | <node value>
<node value>        ::= 0 | 1 | <non-number>
<non-number>        ::= X | U
<valued var list>   ::= <valued var><valued var list> | <valued var>
<valued var>        ::= <connector set> | <store set>
<store set>         ::= <store name>
                        | <store name> {[<index range>, <index range>]}
<index range>       ::= <arithmetic exp>
                        |<arithmetic exp>:<arithmetic exp>[@arithmetic exp]
                        |<arithmetic exp> & <index range>
<connector set>     ::= <connector name> | <vector connector>
<vector connector>  ::= <connector name> {[<index range>]}
```

## Appendix B) SPAM User Guide

This user's guide is divided into four sections. The first two sections describe the way in which cell desrciptions are formulated. The third part instructs the user on running the SPAM compiler and producing output. The final section is really a MAPS user guide. It describes the language and nature of the simulation user interface.

### B.1) Cell Specification

A SPAM leaf cell structural description contains a cell header and pin typing declarations. A composition cell structural description contains those two parts plus a list of components and a list of the abutments of those components.

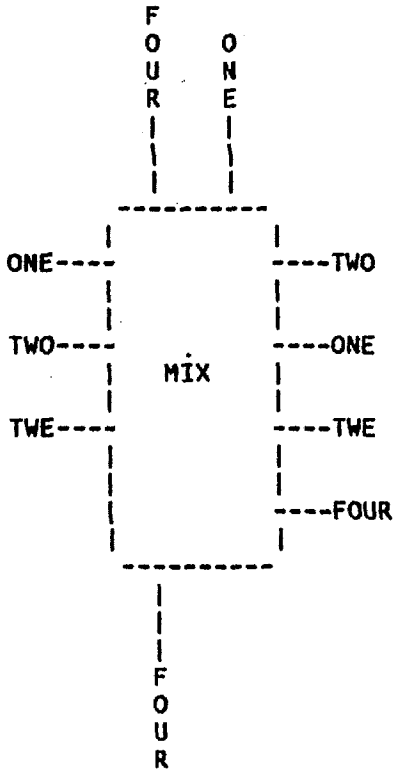
A sample cell header follows:

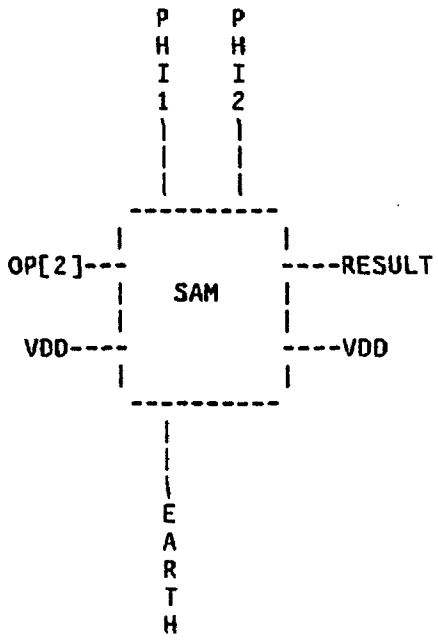
```
CELLDEF sam(TOP phi1,phi2 LEFT op[2],vdd RIGHT result,vdd BOTTOM earth);
```

This cell header declares a new cell named "sam." It has 8 pins. Names are always listed from top to bottom and left to right. The same holds true for sets of pins like "op" above. Op's pin 1 is to the left of its pin 2. Notice that the name vdd is used on both the left and right sides of the cell. This implies that there is a wire connected between those two positions. The following example is a cell with a lot of wires inside.

```
CELLDEF mix(LEFT one,two,twe RIGHT two,one,twe,four TOP for,one BOTTOM four);
```

The following diagrams represent the cells that would result from these two examples:




Pin Diagrams Generated by SPAM

Following the cell header all pins must be "typed". This is accomplished with the use of the key words: INPUT, OUTPUT, IO, CLOCK, POWER, GROUND. The "SAM" cell above might be typed this way:

```
CELLDEF sam(TOP phi1,phi2 LEFT op[2],vdd RIGHT result,vdd BOTTOM earth);
INPUT op;
OUTPUT RESULT;
CLOCK phi1,phi2;
POWER vdd;
GROUND earth;
```

This is the complete interface specification and thus completely describes a leaf cell. The next step is to turn a leaf cell into a composition cell. We must have already defined some other cells (which may themselves already be composition cells). Once that is completed those other cells can be referenced for use as components in this cell. Vectors or matrices of cells may be declared. The components declarations immediately follow the pin typing declarations. Here are some example uses of components:

```
COMPONENT chico(MARX), harpo(MARX), KARL, shoes[1,2](SHOE), toes[2,5](TOE);
```

The cell in which this declaration exists will contain 15 instance elements. Chico and Harpo are instances of the cell definition MARX. KARL is an instance of the cell KARL. (Generic name not required when the same as the instance name.) Shoes is an array of two instances of SHOE just as toes is an array of 10 instances of TOE. Numbering of instances is always from top to bottom and left to right. Numbers in brackets are [rows, columns].

To complete the definition of a composition cell the user must specify the abutments of the components in the cell. Abutments are always given in terms of sides of components. Pin names are only referenced when they are to be omitted from an abutment. Abutments follow the component declarations. Examples follow:

```
1)  LEFT chico <=> RIGHT harpo;
2)  BOTTOM karl, BOTTOM shoes <=> TOP toes;
3)  RIGHT chico <=> LEFT toes[2,1];
4)  LEFT karl ABUTS RIGHT harpo OMIT hat;
```

Notes:

1) Left of chico has same number of pins as right of harpo.

2) Indices are not specified when abutment is for entire matrix. The bottom of two components are abuting the top of one. Lists are always top to bottom and left to right.

3) Right of chico abuts the single instance element toes[2,1] rather then the entire matrix of toes.

4) Right of harpo has one more pin then left of karl but they abut. The pin which is omitted was specified to be "hat." The word "ABUTS" may be substituted for the symbol "<=>."

Always remember to abut the outside of a cell to the outer-most components whenever pins are to be connected. For example, if a cell called "big" has a component on the left side called "little" the declaration "LEFT little <=> LEFT big" must be included if there are pins there to be connected.

The combination of interface specification (names, pins, and types) with components and abutments completely defines the structural description of a composition cell. From this a layout can be inferred, connectivity can be checked and a data structure can be built allowing simulation and automatic documentation.

If no behavior is to be included in a cell, composition cells defined as above can be compiled be adding these two lines:

```
BEHAVE LATER
ENDDEF;
```

"LATER" will change to "NOW" when the behavior is defined.

For complete examples of cell definitions and layouts see appendix C.

## B.2) Behavioral Descriptions

Using programming constructs the chip designer produces a behavioral description of circuit elements. The result is a set of components which act like finite-state machines or combinatorial networks where delay can be specified. A simple non-error detecting description of an NAND gate might look like this:

```
BEHAVE NOW
START
result := (NOT (a AND b)) delay 15;
ENDDEF;
```

"BEHAVE NOW" is the header used when a behavioral description is to be included in the cell definition ("BEHAVE LATER" otherwise). Following that go any integer, real or "internal storage" declarations required by the user. Since none are used here this section is empty and the key word "START" indicates the beginning of the behavior. In the example above the pins "result", "a" and "b" must have been declared in the cell header. When this cell becomes part of a chip, changes on either of its two input pins will cause activation of the behavior. The system then propagates the results of the behavior (in this case just the single pin "result") to whatever pins the changed nodes are structurally connected. In other words if "result" was on the right of this cell and an OR gate was abutting there, the OR gate will be activated in 15 periods. Periods are generally nanoseconds but may be assumed to be any period appropriate to the simulation. If no "after" clause appears the delay is assumed to be zero.

This is the basic operation of the simulator. Changes on inputs cause activation of the behavior described for that cell. Behavior may include various programming constructs but the activity which causes other cells to fire is the activity specified by the SETV or "setval" statement. In the NAND gate example above if the result pin was fed back to one of the inputs of the gate (with a wiring cell) and the other input was tied high (to value 1) then the result pin will oscillate between 1 and 0 every 15 periods. Notice, if the NAND gate was the only cell in the design (ie. it is simulated just like a top level cell description) then it will take specific action on the part of the user to activate the cell. That is, cells only become active when changes occur on their boundries. When simulating top level cells those changes are initiated by the user when running

the simulator (MAPS - appendix B.4).

Declarations of user variables was mentioned earlier. Declarations are placed in between the "BEHAVE" statement and the "START" statement. Three types of declarations are possible:

```
INTEGER i,j,count;
REAL rok,rol;
INTERNAL reg[8] arr[16,16];
```

INTEGER and REAL declarations are used to specify variables for use in designing FOR and WHILE loops, storing values of node sets which need to referenced repeatedly (for example of this, see "ctl" in appendix C.1.) or whatever purpose the user would like. Internal declarations are used to declare bits of internal storage for use in simulation. The internal statement above declares a register of eight bits called reg and a matrix of 16 by 16 bits called arr. Other names for these "bits" of "internal storage" are "storage nodes," "memory cells," and "latches." In other words, the INTERNAL declaration provides the user with a new set of nodes (0, 1, X, U) which can be accessed and changed just like pins. When writing simulation code these "internal" nodes and the external "pin" nodes can be treated identically. Any place that a pin name is permitted, an internal storage name is permitted. Internal storage nodes need not be used in any correspondence with real registers or latches. They are simply for use as temporary storage of values across time. Of course the most common use of these nodes will be for storing values that will eventually be stored in real latches of some sort. Notice this does not affect the way in which a latch is simulated. The internal node simply stores a value (something like a capacitance). The control logic surrounding the device can be arbitrarily designed. At the lowest levels of design it is of course possible to build flip-flops or some real latches to store values.

The important difference between integers or reals and internal nodes is that when integers or reals are changed no cell activation takes place and the new values can be referenced immediately. Internal nodes always cause activation of the cell after a delay period. Even if the delay is zero the change does not take place in this particular activation of the cell and will not be noticed until it is immendiately reactivated. The reason is that a change in the value of a node may infer that some activity elsewhere in the cell is to fire. Any change to a node (internal or external) must cause activation of the entire cell so as to insure

that all possible activities are investigated.

In future paragraphs node "sets" or "lists" are often mentioned. What is being refered to is a set of nodes described in the behavioral notation. Various sytactic structures are used to allow general referencing of nodes. A colon is used to indicate a range of indices into a vector or matrix of nodes. The at sign (@) means "counting by" and allows referencing of every Nth node of the range of nodes specified by the previous colon. A comma separates dimensions in a matrix of nodes. A single node list is either a node name or a node name followed by square brackets in which indices is specified. Indices may be one individual value, a vector or matrix of values, or several sets of indices separated by a space. A node list is then defined to be either a single node list or a concatenation of node lists. Examples follow in the context of the SETV statement:

```
1)  bus[1:5] := (wheels[1:5] AND childs[21:25]);
2)  out[1:top] := 2;
3)  a[1:4] := 1 0 X U;
4)  result[1:4] := ps dr cs ly AFTER 5;
5)  res[1:8] := inp[1:15@2] * 5 + 22;
6)  a[1 5 7:10 12] := b[1:7] AFTER 10;
7)  a := mat[1:3 , 10:30@10];
8)  big_one :=sis bro[2 5] pop[1:13@3,5] X X X mom[1:7@2,7:1@-2];
```

Notes:

1) Boolean operation on node lists must be surrounded by parentheses.
2) Right side will be made of equal length to left side by padding with zeros or truncation. Values in square brackets may be any integer aritmetic exp.
3) 0, 1, X, and U can appear on right side of SETV only.
4) Concatenation of sereral signals to make up a node set.
5) At sign indicates counting by 2
   or "from vector 'inp' use nodes 1 to 15 by 2's."
   Also notice use of arithmetic on node lists.
6) Use of several individual indices.
7) Use of matrix indices returns a list of nine nodes. On the left hand side, 'a' implies a[1:n] where n is the length of the vector a as declared.
8) Use of negative index step on last element on right. Big'one's length>26.

Now all the elements needed to write behavior exist. The actual simulation is designed by the SPAM user and should include whatever error checking deemed needed. For example, if an undefined signal 'U' is found on a node which was supposed to have a value, it is a good idea to propagate that 'U' rather then to produce a good (0 or 1) value. Some conditions may demand that the result of an operation is in an undefined or error state. It is recommended that the 'X' signal be propagated in this situation. Finally, an 'X' on an input should probably cause an output 'X' somewhere. The idea is, that when the user runs a simulation, s/he will be able to determine whether the error came from an internally generated condition or from the use of nodes which had not yet been defined. This will assist in discovering bugs in the logical design of the chip.

All behavioral statements end with a semicolon. Examples of the remaining constructs available to the behavioral designer follow:

```
IF phi1 = 1 THEN
BEGIN ... END ELSE ...
```

This is the form of the IF-THEN-ELSE statement. The boolean expression may include any arithmetic comparison of any nodes. Comparisons can be prefixed with NOT or linked together with AND, OR, IMP, or EQV all of which are functions defined in SIMULA. The statement following the THEN can be a single statement or a block of statements surrounded by BEGIN END;. The term "EF" can be substituted for the words "ELSE IF" when possible. The above example is intended to demonstrate the way a section of code can be partitioned so as to execute on a clock signal. The BEGIN-END block shown will be executed whenever the clock signal phi1 is high. (Sequential "clock driven" simulation is different - see discussion of "SEQUENCE.")

```
FOR i:=1 step 1 until j DO <statement>;
WHILE i <= nody[1:9@2] DO <statement>;
```

These two constructs are similar to the SIMULA FOR and WHILE loops and should be easy to understand. The term <statement> means a single SPAM statement or a BEGIN-END block.

```
n:=3 + nody[5 10];
```

This is the final construct in the standard SPAM notation. It is a simple numerical assignment statement. The left side must be a single INTEGER or REAL variable. The right side may contain any combination of node lists, operations or variables desired. No delay may be specified.

Comments may interspersed with SPAM code by use of the exclamation mark (!).

```
!This is a comment;
```

The above descriptions apply to the standard SPAM simulation technique. The "sequence" type of simulation as described in the thesis section 4.2.3 requires one additional construct. The basic idea behind sequential simulation is "clock-driven" activation of the cell. The purpose of this technique is to produce a microcode-like simulation of instruction processors. The most common instruction processor is the microprocessor. The idea is that the information which is of most interest in the top-level simulation of a microprocessor is information regarding longest delay paths and behavior and timing of individual instructions. Each instruction may take a different number of clock cycles to complete and each clock cycle is uniquely determined by microcode rather then simply by hardware. In order to allow simulation of this type of enviornment and to make it simple to translate between microcode and SPAM notation, the "sequence" domain was invented. The difference is that in a sequential simulation, a cell is activated only by the occurence of a specified signal. In a microprocessor that signal will be a clock signal though any signal is permitted.

In order to indicate that a cell is to be simulated in this manner replace "START" with "SEQUENCE." The format of the statement which causes the process to wait for the next occurence of the specified signal is demonstrated demonstrated in the following examples:

```
NEXT phi1;
NEXT NOT sig;
```

The inclusion of "NOT" means that simulation of the cell should continue when the specified node makes a negative transition (1 to 0). Without the "NOT" it is the positive transition (0 to 1) which triggers reactivation of the cell.

Further examples of both types of simulations can be found in appendix C.

### B.3) Compiling Input

**The following example shows how to run SPAM:**

```
@spam
From where? (TTY:) mini.om                    (This is the SPAM source file)
Do you want to make a simulation file?yes
Name for output files (up to 5 chars.):mini   (Actually creates two files.
                                                minis.sim and minic.sim)
Choose top level cell name from the following:
 CTL, ALU, PORT2, LATCH, PORT1, PROCESSOR     (Name of circuit actually being
 ADD, MEMCELL                                  simulated.  Leaf cells chosen
processor                                      by BEHAVE NOW/LATER statements)
Do you want automatic documentation?yes
Name for output file:hier.pic                 (File will contain all pictures
Choose top level cell name from the following:  of all cells and the hierarchy
 CTL, ALU, PORT2, LATCH, PORT1, PROCESSOR      specified by the top level cell)
 ADD, MEMCELL
processor
Enter designer name:
Richard Segal                                 (Used in header in doc file)

End of SPAM execution.
```

The result of all this is to create three output files. The two simulation files contain SIMULA code which can then be compiled and simulated as follows:

```
@simula                          {Invoking the simula compiler}

*minis                           {This file contain cell definitions
                                  and initialization}

NO ERRORS DETECTED

*minic                           {This file connects up the data
                                  structure of pins and wires and
                                  starts up MAPS}
NO ERRORS DETECTED

*↑Z                              {Control-Z exits from SIMULA}
EXIT
@load minic,libsim/lib           {Loading the simulator}
LINK: Loading

EXIT
@save                            {Saving the executable simulator file}
 [[MINIC.EXE.1 Saved
@minic                           {Starting the simulation}
MAPS>end;                        {Ready to run simulation. See MAPS user guide}
End of SPAMAPS execution.
```

The third output file of the SPAM compilation is the automatic documentation. This includes a hierarchical map of the circuit, pin diagrams of all cells, and floor plans with generic and instance names for all composition cells. Examples are shown in appendix C.

Special late note: Upon request, SPAM will now perform the above compilation automatically.

## B.4) Running a Simulation - MAPS User's Guide

MAPS is the program which allows the user to communicate with an ongoing simulation. The prompt for a user command is "MAPS>". At that point the user can enter any MAPS command. Commands can be abbreviated to any number of letters. Since each command begins with a unique letter, one letter abbreviations will suffice. For example, "OUTPUT" can be abbreviated by "O" and the phrase "BREAK AT" (for setting break points) can be abbreviated to just "B".

Since a component can be uniquely specified only by its embeddedness in the hierarchy there is a problem with long names. For example we might have an instance (component) of an adder cell which could be called PROCESSOR/ALU/ARITH_PART/INT[1,2]/ADDER[2,2]. Since there might be several groups of adders in the chip possibly even with the same names, that long list might be the only way of refering to that particular instance if names are to be used. Since instances need to specified for most commands this could be very unwieldy. In order to eliminate this problem completely each unique instance is given a number. There is a table of instances and corresponding instance numbers which is listed with the use of the TABLE command.

INPUT, OUTPUT, and BREAK commands may be scheduled repeatedly with the use of an EVERY clause (see specific command). Whenever such a "timed device" is specified with one of those commands it is placed in a list of timed devices. The list can be looked at with the LIST command and cancelled with the REMOVE command.

The commands are all summarized below. Commands may be longer then one input line, but all commands must be terminated with a semicolon. Example simulations are shown in appendices C.1 and C.2.1. Optional phrases are surrounded by {}'s. The terms inside of <>'s should be replaced with the actual parameters of the specified type when using the command.

PROCEED
-Continue simulation.

START <instance>
-This command is used to start the execution of a SEQUENCE simulation. It is needed because no NEXT command has been executed (the only regular way of activating a SEQUENCE) until it is started the first time. This command should be issued once and only once.

TABLE {file name}
-Display table of instance names and corresponding numbers {Place table in specified file name}.

LIST
-List active timed devices.

REMOVE <device number>
-Remove timed device specified by the integer <device number>.

HELP
-Display this text.

USE <file name>
-Read commands from <file name>.

BREAK AT <time> {EVERY <interval>}
-Stop simulation and return to "MAPS>" at time = <integer> {repeating every <interval>}. <time> and <interval> must be integers.

INPUT <instance> , <node list> := <node value list>
{AT <time> {EVERY <interval> {FOR <duration>}}}
-Set nodes in <node list> to the values (0,1,X,U) or nodes in <node value list> {at integer <time> {repeating every integer <interval> {for length integer <duration>}}}. The FOR clause assists in specifying clock inputs. For example INPUT 2,clk:=1 AT 1 EVERY 20 FOR 5; makes a clock signal with a period of 20 and a high time of 5. (Duty cycle = 5/20)

OUTPUT <instance> , <node list> {AT <time> {EVERY <interval>}}
-Display value (0, 1, X, or U) of all nodes in <node list> of <instance>
{at integer <time> {repeating every integer <interval>}}

END
-End simulation

Special late note: INPUT and OUTPUT now also work with files allowing specification of sequences of vectors and easier comparisons between different simulations.

## Appendix C) Examples

This section contains examples of cell design, interface specification, automatic documentation, and MAPS simulation. Two machines are described. The first is described at a fairly low level of abstraction with several composition cells. The second is first described at the highest possible level using "sequential" simulation. It is then is broken up into five composition cells and many leaf cells to demonstrate the approximate capacity of the structural compiler.

### C.1) A Miniture OM Chip

The following example is a simplified version of the OM data path chip [Mead and Conway]. The basic structure can be gathered from the floor plan diagrams produced by SPAM. There is a 16 wire bus which runs across all eight registers, the ALU, and the ports on either side. The even numbered wires on this set of 16 are called the A bus and the odd are called the B bus. The ALU, adder, latch and register cells should be straightfoward. The controller is a bit more complex. Its function is to decode the sixteen bit control word which comes in from the bottom and produce the correct signals on the registers, ALU, or ports to read or write. The control word is made up of four fields of four bits each. The first two fields are the A and B sources and the second two are the A and B destinations respectively. The four bit codes which can be placed in these fields are summarized as follows:

| Four Bit Code | Associated Device |
|---|---|
| 0 | No op |
| 1 | Register 1 |
| 2 | Register 2 |
| 3 | Register 3 |
| 4 | Register 4 |
| 5 | Register 5 |
| 6 | Register 6 |
| 7 | Register 7 |
| 8 | Register 8 |
| 9 | ALU (Source Only - Latch ALWAYS latches) |
| 10 | Port 1 |
| 11 | Port 2 |

**Registers and ports are accessed on the phi1 clock cycle and the ALU is accessed on phi2. All lines go to 0 in between clock phases.**

```
CELLDEF processor (LEFT a[8] RIGHT b[8] BOTTOM control[16],phi1,phi2 );
IO a,b;
INPUT control,phi1,phi2;
COMPONENT   port1[8,1], port2[8,1],
            ALU,scratch[1,8](MEMCELL),CTL;

            LEFT CTL,LEFT port1 <=> LEFT processor;
            RIGHT CTL,RIGHT port2 <=> RIGHT processor;
            BOTTOM CTL <=> BOTTOM processor;
            BOTTOM port1,BOTTOM scratch,
                BOTTOM ALU,BOTTOM port2 <=> TOP CTL;
            RIGHT port1 <=> LEFT scratch;
            RIGHT scratch <=> LEFT ALU;
            RIGHT ALU <=> LEFT port2;
BEHAVE LATER
ENDDEF;

CELLDEF memcell (LEFT bigbus[16] RIGHT bigbus[16] BOTTOM lfa,lfb,wta,wtb);
!         lfa/b -> load from bus A/B   -   wta/b -> write to bus A/B  ;
IO bigbus,lfa,lfb,wta,wtb;
BEHAVE NOW
INTERNAL n[8];
START
IF lfa lfb=X THEN n:=X AFTER 5
    EF lfa lfb=U THEN n:=U AFTER 5
    EF lfa=1 THEN n:=bigbus[2:16@2] AFTER 5
    EF lfb=1 THEN n:=bigbus[1:15@2] AFTER 5;
IF wta=X THEN bigbus[2:16@2]:=X AFTER 5
    EF wta=U THEN bigbus[2:16@2]:=U AFTER 5
    EF wta=1 THEN bigbus[2:16@2]:=n AFTER 5;
IF wtb=X THEN bigbus[1:15@2]:=X AFTER 5
    EF wtb=U THEN bigbus[1:15@2]:=U AFTER 5
    EF wtb=1 THEN bigbus[1:15@2]:=n AFTER 5;
ENDDEF;
```

```
CELLDEF port1 (LEFT pad RIGHT b,a BOTTOM la,lb,da,db,dp,rp
                                    TOP la,lb,da,db,dp,rp);
! la=load a, lb=load b, da=drive a, db=drive b, dp=drive pad, rp=read pad  ;
IO pad,a,b,la,lb,da,db,dp,rp;
BEHAVE NOW
INTERNAL n;
START
IF la=1 THEN n:=a
    EF lb=1 THEN n:=b
    EF da=1 THEN a:=n
    EF db=1 THEN b:=n
    EF dp=1 THEN pad:=n
    EF rp=1 THEN n:=pad;
ENDDEF;

CELLDEF port2 (RIGHT pad LEFT b,a BOTTOM rp,dp,db,da,lb,la
                                    TOP rp,dp,db,da,lb,la);
!!Mirroring not implemented, so port2 is reflection of port1;
IO pad,a,b,la,lb,da,db,dp,rp;
BEHAVE NOW
INTERNAL n;
START
IF la=1 THEN n:=a
    EF lb=1 THEN n:=b
    EF da=1 THEN a:=n
    EF db=1 THEN b:=n
    EF dp=1 THEN pad:=n
    EF rp=1 THEN n:=pad;
ENDDEF;
```

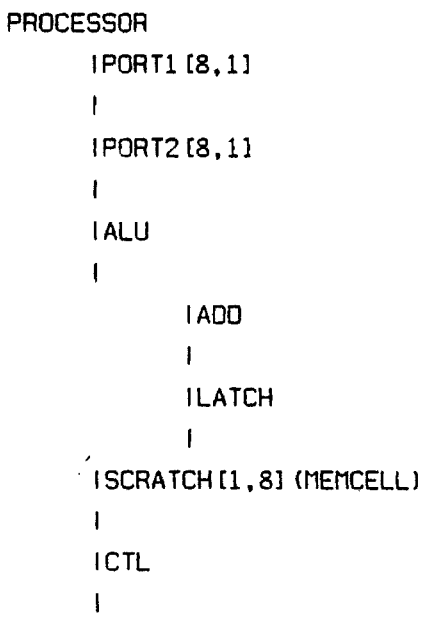
```
CELLDEF ctl (BOTTOM cin[16],phi1,phi2 TOP a[6],c[32],alua,alub,b[6]);
! BOTTOM signals are Control word (cin) and Clocks (phi1, phi2);
! TOP signals are for port1, register control, ALU write to A&B, and port2;
INPUT cin,phi1,phi2;
OUTPUT a,c,alua,alub,b;
BEHAVE NOW
INTEGER ARRAY REG(1:4);
INTEGER I;
START
FOR I:=1 STEP 1 UNTIL 4 DO
    REG[I]:=cin[I*4-3:I*4];
IF PHI1=1 THEN
BEGIN                                                    !On phase 1;
    FOR I:=1 STEP 1 UNTIL 2 DO       !SOURCES;
    IF REG[I]<9 AND REG(I)>0 THEN
        c[REG(I)*4+I-2]:=1 AFTER 15
    EF REG(I)=10 THEN a[I+2]:=1 AFTER 15
    EF REG(I)=11 THEN b[5-I]:=1 AFTER 15;
    FOR I:=3 STEP 1 UNTIL 4 DO       !DESTINATIONS;
    IF REG(I)<9 AND REG(I)>0 THEN
        c[(REG(I)-1)*4+I-2]:=1 AFTER 15
    EF REG(I)=10 THEN a[I-2]:=1 AFTER 15
    EF REG(I)=11 THEN b[9-I]:=1 AFTER 15;
END
EF PHI2=1 THEN
BEGIN
    IF REG[1]=9 THEN alua:=1;                        !On phase 2;
    IF REG[2]=9 THEN alub:=1;
END
ELSE c[1:32] a[1:6] b[1:6] alua alub := 0 AFTER 10;
ENDDEF;
```

```
CELLDEF alu (LEFT inp[16] RIGHT out[16] BOTTOM wta,wtb);
INPUT inp,wta,wtb;
OUTPUT out;
COMPONENT add,latch;
        LEFT add <=> LEFT alu;
        RIGHT latch <=> RIGHT alu;
        RIGHT add <=> LEFT latch;
        BOTTOM add,BOTTOM latch <=> BOTTOM alu;
BEHAVE LATER
ENDDEF;

CELLDEF latch (LEFT bus[16],num[8] RIGHT bus[16] BOTTOM wta,wtb);
INPUT num,wta,wtb;
IO bus;
BEHAVE NOW
INTERNAL result[8];
START
result:=num AFTER 5;
IF wta=U OR wta=X THEN bus[2:16@2]:=U AFTER 10
    EF wta=1 THEN bus[2:16@2]:=result AFTER 10;
IF wtb=U OR wtb=X THEN bus[1:15@2]:=U AFTER 10
    EF wtb=1 THEN bus[1:15@2]:=result AFTER 10;
ENDDEF;

CELLDEF add (LEFT bus[16] RIGHT bus[16], out[8]);
IO bus;
OUTPUT out;
BEHAVE NOW
START
out:=bus[1:15@2]+bus[2:16@2] AFTER 25;
ENDDEF;
```

The following pages contain the documentation file from the above design.

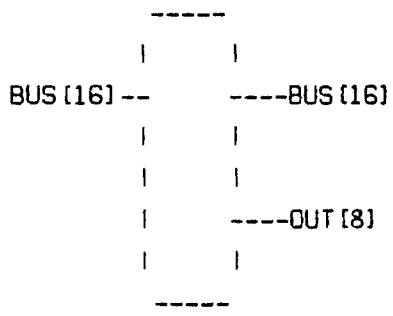
MINI.OM  by Richard Segal  1980-10-31

Structure Map:



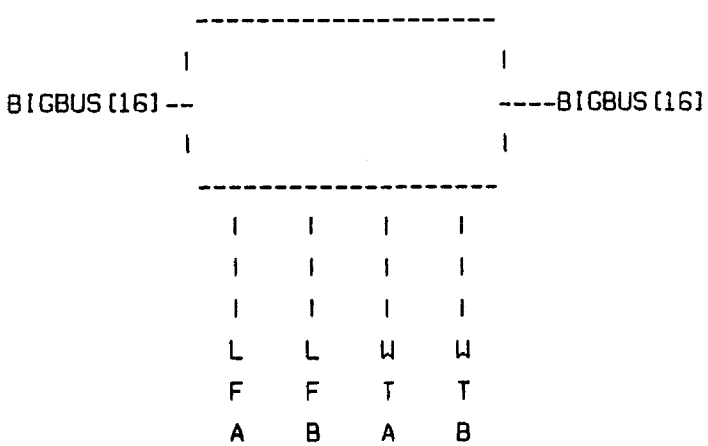
MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'ADD':



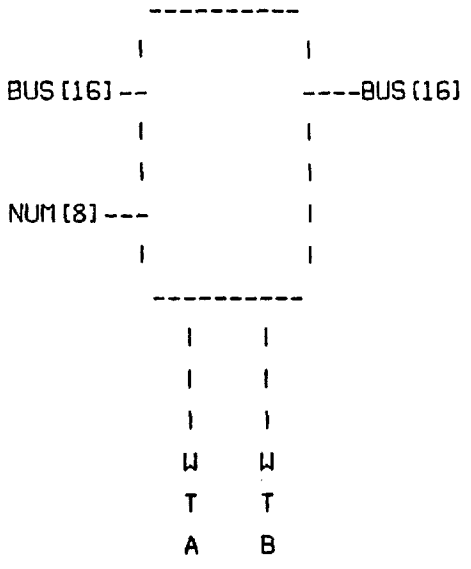
MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'MEMCELL':



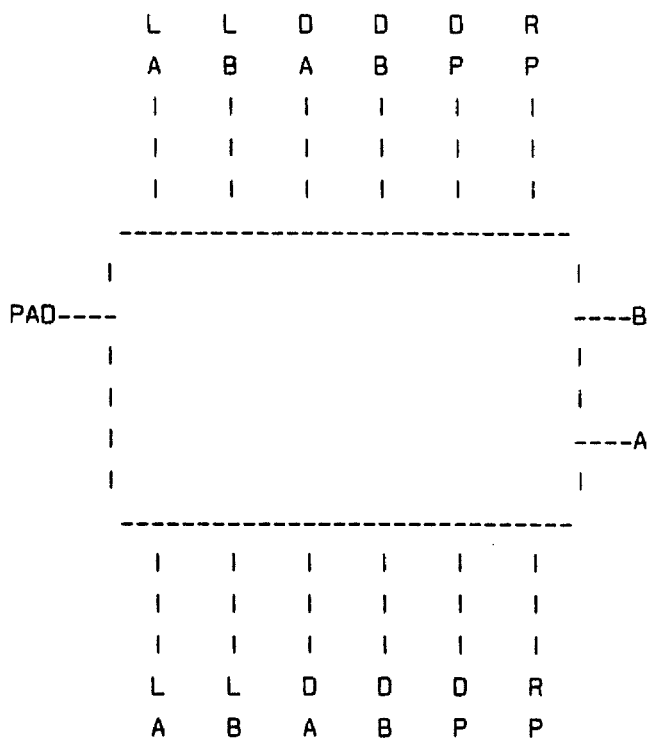
MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'LATCH':



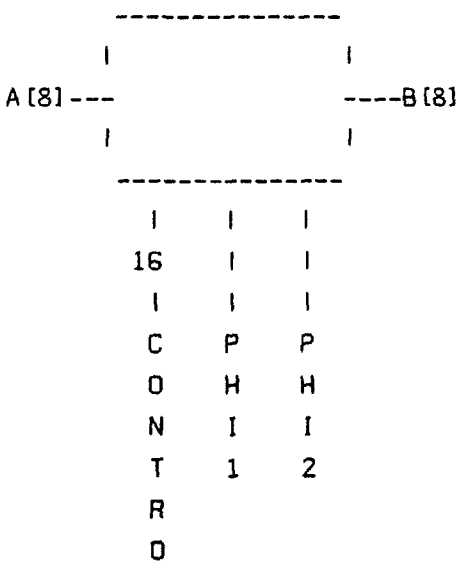
MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'PORT1':



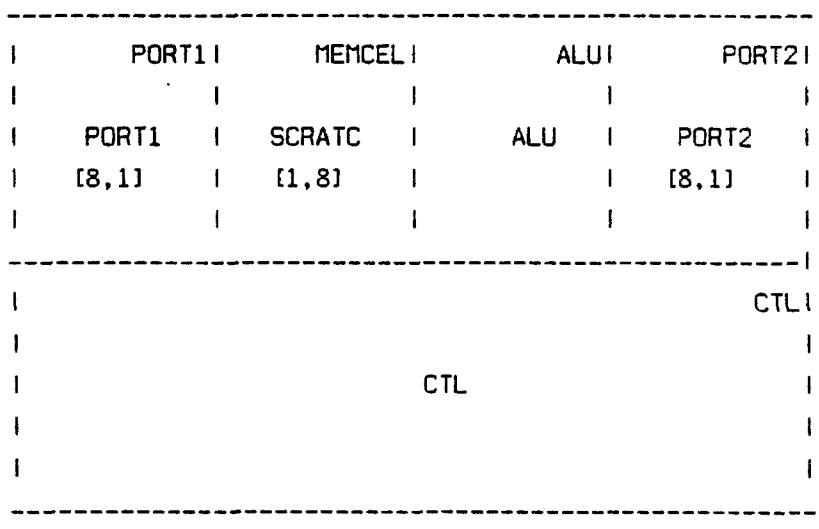
MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'PROCESSOR':



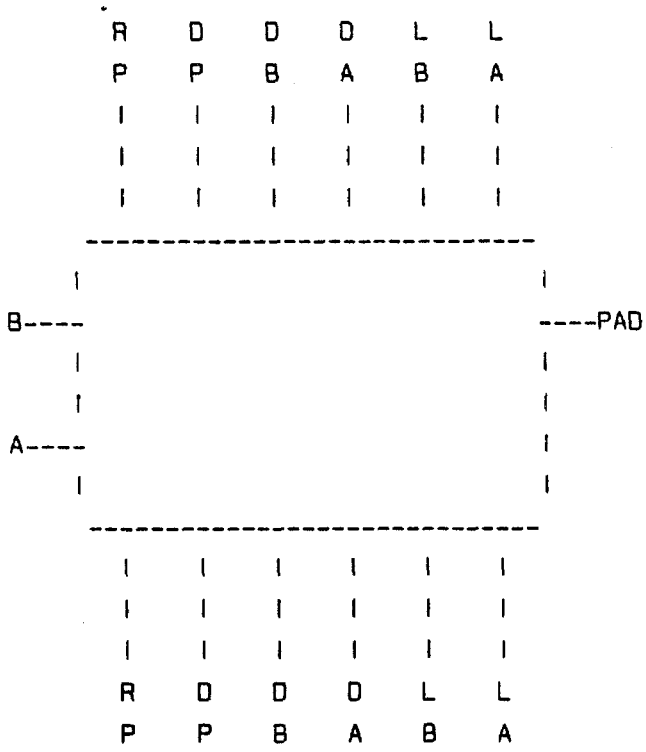
MINI.OM by Richard Segal 1980-10-31

Floor Plan of 'PROCESSOR':



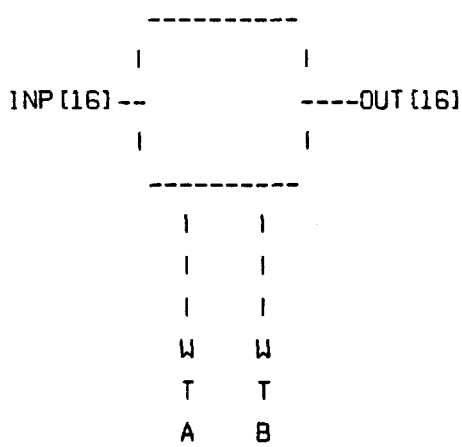
MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'PORT2':



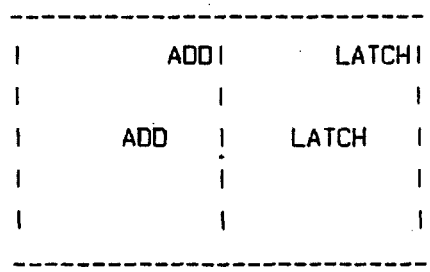
MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'ALU':



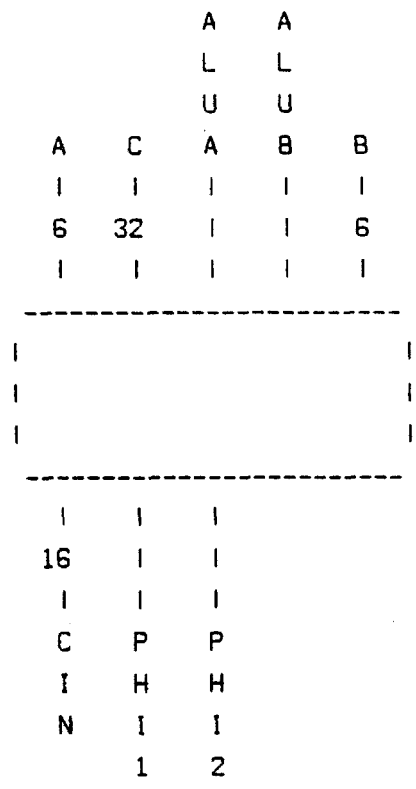
MINI.OM by Richard Segal 1980-10-31

Floor Plan of 'ALU':



MINI.OM by Richard Segal 1980-10-31

Cell Specification of 'CTL':



MAPS simulation of above chip:

```
@minic
MAPS>table;                                   {List instance table}
    1          PROCESSOR
    2          PROCESSOR/CTL[1,1]
    3          PROCESSOR/SCRATCH[1,1]
    4          PROCESSOR/SCRATCH[1,2]
    5          PROCESSOR/SCRATCH[1,3]
    6          PROCESSOR/SCRATCH[1,4]
    7          PROCESSOR/SCRATCH[1,5]
    8          PROCESSOR/SCRATCH[1,6]
    9          PROCESSOR/SCRATCH[1,7]
   10          PROCESSOR/SCRATCH[1,8]
   11          PROCESSOR/ALU[1,1]/LATCH[1,1]
   12          PROCESSOR/ALU[1,1]/ADD[1,1]
   13          PROCESSOR/PORT2[1,1]
   14          PROCESSOR/PORT2[2,1]
   15          PROCESSOR/PORT2[3,1]
   16          PROCESSOR/PORT2[4,1]
   17          PROCESSOR/PORT2[5,1]
   18          PROCESSOR/PORT2[6,1]
   19          PROCESSOR/PORT2[7,1]
   20          PROCESSOR/PORT2[8,1]
   21          PROCESSOR/PORT1[1,1]
   22          PROCESSOR/PORT1[2,1]
   23          PROCESSOR/PORT1[3,1]
   24          PROCESSOR/PORT1[4,1]
   25          PROCESSOR/PORT1[5,1]
   26          PROCESSOR/PORT1[6,1]
   27          PROCESSOR/PORT1[7,1]
   28          PROCESSOR/PORT1[8,1]
MAPS>input 1,control:=0;                      {Set control word to no op}
MAPS>inp 1,phi1:=1 at 50 every 120 for 50;    {Set up two-phase clocks}
MAPS>inp 1,phi2:=1 at 110 every 120 for 50;
MAPS>break at 100 every 50;                   {Set wrong break point}
MAPS>list;                                    {So list timed devices}
Timed device 1:
  INP 1,PHI1:=1 AT 50 EVERY 120 FOR 50;
-------------------------------------------------------------
```

```
Timed device 2:
  INP 1,PHI2:=1 AT 110 EVERY 120 FOR 50;
-------------------------------------------------------------
Timed device 3:
  BREAK AT 100 EVERY 50;
-------------------------------------------------------------
MAPS>remove 3;                                  {And remove offender}
MAPS>break at 45 every 120;                     {Break before each phase 1}
MAPS>lis;
Timed device 1:
  INP 1,PHI1:=1 AT 50 EVERY 120 FOR 50;
-------------------------------------------------------------
Timed device 2:
  INP 1,PHI2:=1 AT 110 EVERY 120 FOR 50;
-------------------------------------------------------------
Timed device 3:
  BREAK AT 45 EVERY 120;
-------------------------------------------------------------
MAPS>o 3,bigbus[1:16];
  O 3,BIGBUS[1:16]   at time = 0:
U U U U U U U U U U U U U U U U                 {Both busses undefined}

MAPS>p;                                         {Proceed}
Break at time = 45                              {Still before first clock pulse}
MAPS>p;                                         {Proceed to test no op}
Break at time = 165
MAPS>o 3,bigbus;
  O 3,BIGBUS   at time = 165:
U U U U U U U U U U U U U U U U                 {Still nothing on buses}

MAPS>o 4,n;                                     {Nothing in register 2}
  O 4,N   at time = 165:
U U U U U U U U

MAPS>i 4,n:=5;                                  {Put value 5 into reg 2}
MAPS>i 6,n:=7;                                  {Put 6 into reg 4}
MAPS>i 1,control[1:4]:=2;                       {Set A Bus source to reg 2}
MAPS>i 1,control[5:8]:=4;                       {B bus source to reg 4}
MAPS>p;                                         {GO!!}
Break at time = 285
MAPS>o 3,bigbus[2:16@2];
  O 3,BIGBUS[2:16@2]   at time = 285:
```

```
0 0 0 0 0 1 0 1                          {A bus has correct value}

MAPS>o 3,bigbus[1:15@2];
  O 3,BIGBUS[1:15@2]    at time = 285:
0 0 0 0 0 1 1 1                          {B bus too}

MAPS>o 11,result;
  O 11,RESULT    at time = 285:
0 0 0 0 1 1 0 0                          {LATCH has sum since ADD
                                          always adds}
MAPS>i 1,control:=0;                     {Reset control word}
MAPS>i 1,control[1:4]:=9;                {Use ALU (latch) as A source}
MAPS>i 1,control[9:12]:=10;              {And port1 as A destination}
MAPS>p;
Break at time = 405
MAPS>o 3,bigbus[2:16@2];
  O 3,BIGBUS[2:16@2]    at time = 405:
0 0 0 1 0 0 1 1                          {New sum of 12 (old sum) and
                                          7 (still on B bus) = 19}
MAPS>o 3,bigbus[1:15@2];
  O 3,BIGBUS[1:15@2]    at time = 405:
0 0 0 0 0 1 1 1                          {B bus had a 7}

MAPS>i 1,control[5:8]:=7;                {Load an empty register onto
MAPS>p;                                   B bus to stop the mad adder}
Break at time = 525
MAPS>o 3,bigbus[2:16@2];
  O 3,BIGBUS[2:16@2]    at time = 525:
0 0 0 1 0 0 1 1                          {Sum on A bus still 19}

MAPS>o 3,bigbus[1:15@2];
  O 3,BIGBUS[1:15@2]    at time = 525:
U U U U U U U U                          {Contents of register 7 on
                                          B bus now}
MAPS>o 1,control;
  O 1,CONTROL    at time = 525:
1 0 0 1 0 1 1 1 1 0 1 0 0 0 0 0          {Current state of control word}

MAPS>i 1,control:=0;                     {Reset it to zero}
MAPS>i 1,control[9:12]:=8;               {New destination is register 8}
MAPS>i 1,control[1:4]:=2;                {Source is register 2}
MAPS>p;
```

```
Break at time = 645
MAPS>o 10,n;
  0 10,N    at time = 645:
0 0 0 0 0 1 0 1                          {Successful transfer}

MAPS>i 1,control[1:4]:=10;               {Keeping same destination try
MAPS>p;                                   getting saved sum from port1}
Break at time = 765
MAPS>o 10,n;
  0 10,N    at time = 765:
0 0 0 1 0 0 1 1                          {19}

MAPS>end;
End of SPAMAPS execution
```

## C.2) The GR2

The GR2 is a stack data processor [Efland and Mosteller]. This appendix presents two views of the chip. The first is a behavioral description of the highest level of abstraction of the chip using "sequential" behavior to simulate its microcode enviornment. The second part is a structural description only, including several layers of abstraction. Some of the documentation produced by SPAM is shown.

### C.2.1) A Sequential Description

```
!=================================================================;
!                                                                 ;
!                      G R 2   SPAM DESCRIPTION                   ;
!                              August 1980                        ;
!=================================================================;

CELLDEF GR2(
        TOP     reset,rdy,ph1,ph2,s1,s2,din
        RIGHT   GND,VDD
        BOTTOM  ds,cd,rw,as,dout,ad[8]
        );

        INPUT   reset,rdy,s1,s2,din;
        OUTPUT  ds,cd,rw,as,dout;
        IO      ad;
        CLOCK   ph1,ph2;
        POWER   VDD;
        GROUND  GND;
```

```
BEHAVE NOW
INTERNAL a_reg[8] b_reg[8] top_reg[8] base_reg[8] program_counter[8]
         ir[4] ea[8] n[8] off[8];

SEQUENCE
    IF reset=1 THEN BEGIN
        ! Clear the output pins like a good machine;
        ds cd rw as dout :=0;
        ! Initialize the internal state;
        program_counter:=0; base_reg:=0;
        top_reg:=0;
          NEXT ph1;
    END;
    WHILE TRUE DO BEGIN
        ad[1:8]:=program_counter; ! Instruction fetch;
        as ds cd rw := 1 0 1 1;
          NEXT ph1;
        as ds cd rw := 0 0 0 0;
        program_counter:=program_counter+1;
          NEXT ph1;
        as ds cd rw := 0 1 0 0;
        ir=ad[5:8];
          NEXT ph1;

        IF ir=0 0 0 1 THEN BEGIN          ! ADD;
            b_reg:=b_reg+a_reg;
              NEXT ph1;
              NEXT ph1;
        END ELSE IF ir=0 0 1 0 THEN BEGIN ! SUB;
            b_reg:=b_reg-a_reg;
              NEXT ph1;
              NEXT ph1;
        END ELSE IF ir=0 0 1 1 THEN BEGIN ! AND;
            b_reg:=(b_reg AND a_reg);
              NEXT ph1;
        END ELSE IF ir=0 1 0 0 THEN BEGIN ! OR;
            b_reg:=(b_reg OR a_reg);
              NEXT ph1;
        END ELSE IF ir=0 1 0 1 THEN BEGIN ! NOT;
            a_reg:=(NOT a_reg);
```

```
      NEXT ph1;
END ELSE IF ir=0 1 1 0 THEN BEGIN ! LIT;
    ad:=program_counter;
    as ds cd rw := 1 0 1 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
    program_counter:=program_counter+1;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    a_reg:=ad;
      NEXT ph1;
END ELSE IF ir=0 1 1 1 THEN BEGIN ! LOD;
    ad:=program_counter; ! get <dif> field;
    as ds cd rw := 1 0 1 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
    ea:=base_reg;
    program_counter:=program_counter+1;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    n:=ad;
      NEXT ph1;
    WHILE n>0 DO BEGIN ! Follow up pointer chain;
        ad[1:8]:=ea;
        as ds cd rw := 1 0 0 1;
          NEXT ph1;
        as ds cd rw := 0 0 0 0;
        n:=n-1;
          NEXT ph1;
        as ds cd rw := 0 1 0 0;
        ea:=ad[1:8];
          NEXT ph1;
    END;
    ad:=program_counter; ! get <off> field;
    as ds cd rw := 1 0 1 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
    program_counter:=program_counter+1;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    off:=ad;
      NEXT ph1;
```

```
    ad[1:8]:=ea; ! get the data;
    as ds cd rw := 1 0 0 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    a_reg:=ad;
      NEXT ph1;
END ELSE IF ir=1 0 0 0 THEN BEGIN ! STO;
    ad:=program_counter; ! get <dif> field;
    as ds cd rw := 1 0 1 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
    ea:=base_reg;
    program_counter:=program_counter+1;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    n:=ad;
      NEXT ph1;
    WHILE n>0 DO BEGIN ! Follow up pointer chain;
        ad:=ea;
        as ds cd rw := 1 0 0 1;
          NEXT ph1;
        as ds cd rw := 0 0 0 0;
        n:=n-1;
          NEXT ph1;
        as ds cd rw := 0 1 0 0;
        ea:=ad;
          NEXT ph1;
    END;
    ad:=program_counter; ! get <off> field;
    as ds cd rw := 1 0 1 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
    program_counter:=program_counter+1;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    off:=ad;
      NEXT ph1;
    ad:=ea; ! Save the data;
    as ds cd rw := 1 0 0 0;
      NEXT ph1;
```

```
    as ds cd rw := 0 0 0 0;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    ad:=a_reg;
      NEXT ph1;
END ELSE IF ir=1 0 0 1 THEN BEGIN ! CALL;
    b_reg:=base_reg;
      NEXT ph1;
    a_reg:=program_counter;
      NEXT ph1;
    base_reg:=top_reg;
      NEXT ph1;
    ad:=ea; ! Get jump address;
    as ds cd rw := 1 0 1 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
      NEXT ph1;
    as ds cd rw := 0 1 0 0;
    program_counter:=ad;
      NEXT ph1;
END ELSE IF ir=1 0 1 0 THEN BEGIN ! EXIT;
    base_reg:=base_reg-1;
      NEXT ph1;
    top_reg:=base_reg;
      NEXT ph1;
    base_reg:=b_reg;
      NEXT ph1;
    program_counter:=a_reg;
      NEXT ph1;
END ELSE IF ir=1 0 1 1 THEN BEGIN ! EQ;
    IF a_reg=b_reg THEN b_reg:=1
    ELSE b_reg:=0;
      NEXT ph1;
      NEXT ph1;
      NEXT ph1;
    IF b_reg=1 THEN NEXT ph1;
END ELSE IF ir=1 1 0 0 THEN BEGIN ! JCT;
    ad:=program_counter; ! Get jump address;
    as ds cd rw := 1 0 1 1;
      NEXT ph1;
    as ds cd rw := 0 0 0 0;
    program_counter:=program_counter+1;
```

```
                NEXT ph1;
              as ds cd rw := 0 1 0 0;
              IF a_reg[8]=1 THEN program_counter:=ad;
                NEXT ph1;
                NEXT ph1;
                NEXT ph1;
              IF a_reg[8]=1 THEN NEXT ph1;
            END ELSE IF ir=1 1 0 1 THEN BEGIN ! PUSH;
              top_reg:=top_reg+1;
                NEXT ph1;
              ad:=top_reg; ! save b register in stack;
              as ds cd rw := 1 0 0 0;
                NEXT ph1;
              as ds cd rw := 0 0 0 0;
                NEXT ph1;
              ad:=b_reg;
              as ds cd rw := 0 1 0 0;
                NEXT ph1;
              b_reg:=a_reg;
                NEXT ph1;
            END ELSE IF ir=1 1 1 0 THEN BEGIN ! POP;
              a_reg:=b_reg;
                NEXT ph1;
              ad:=top_reg; ! Get b register from stack;
              as ds cd rw := 1 0 0 1;
                NEXT ph1;
              as ds cd rw := 0 0 0 0;
                NEXT ph1;
              as ds cd rw := 0 1 0 0;
              b_reg:=ad;
                NEXT ph1;
              top_reg:=top_reg-1;
                NEXT ph1;
            END;
        END;
ENDDEF;
```

Simulating the GR2:

```
egr2c
MAPS>table;                                  {Only one instance being simulated - GR2}
    1          GR2
MAPS>start 1;                                {Start up needed since its a SEQUENCE}
MAPS>input 1,ph1:=1 at 5 every 5 for 2;          {Set up clock, period=5}
MAPS>output 1,program_counter[1:8] at 4 every 5; {Some signals we need to see}
MAPS>out 1,ir[1:4] at 4 every 5;                 {Instruction register}
MAPS>out 1,as ds cd rw at 4 every 5;             {Status bits}
MAPS>break at 4 every 5;
MAPS>p;                                          {Start up and initialize}
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 4:
U U U U U U U U

  OUT 1,IR[1:4]    at time = 4:
U U U U

  OUT 1,AS DS CD RW    at time = 4:
1 0 1 1

Break at time = 4                            {Ready for first clock cycle}
MAPS>list;
Timed device 1:
  INPUT 1,PH1:=1 AT 5 EVERY 5 FOR 2;
-------------------------------------------------------------
Timed device 2:
  OUTPUT 1,PROGRAM_COUNTER[1:8] AT 4 EVERY 5;
-------------------------------------------------------------
Timed device 3:
  OUT 1,IR[1:4] AT 4 EVERY 5;
-------------------------------------------------------------
Timed device 4:
  OUT 1,AS DS CD RW AT 4 EVERY 5;
-------------------------------------------------------------
Timed device 5:
  BREAK AT 4 EVERY 5;
-------------------------------------------------------------
MAPS>p;                                          {GO!!}
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 9:
```

```
0 0 0 0 0 0 0 1                                  {Ready for first instruction}

  OUT 1,IR[1:4]    at time = 9:
U U U U

  OUT 1,AS DS CD RW    at time = 9:
0 0 0 0

Break at time = 9
MAPS>i 1,ad[1:8]:=0 1 1 0;                       {'LIT' instruction given}
MAPS>p;
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 14:
0 0 0 0 0 0 0 1

  OUT 1,IR[1:4]    at time = 14:
0 1 1 0                                          {IR now contains instruction}
                                                 {It is a "literal" instruction
  OUT 1,AS DS CD RW    at time = 14:              which means, put value on
0 1 0 0                                          AD bus into A register}

Break at time = 14
MAPS>p;
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 19:
0 0 0 0 0 0 0 1

  OUT 1,IR[1:4]    at time = 19:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 19:
1 0 1 1

Break at time = 19
MAPS>o 1,ad[1:8];
  O 1,AD[1:8]    at time = 19:
0 0 0 0 0 0 0 1                                  {Bus was given value on PC}

MAPS>i 1,ad[1:8]:=8;                             {Put on new value to "LIT"}
MAPS>p;
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 24:
0 0 0 0 0 0 1 0                                  {Second cycle}
```

```
  OUT 1,IR[1:4]    at time = 24:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 24:
0 0 0 0

Break at time = 24
MAPS>o 1,a_reg[1:8];
  O 1,A_REG[1:8]    at time = 24:
U U U U U U U U                                    {Waiting for LIT to finish}

MAPS>p;
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 29:
0 0 0 0 0 0 1 0

  OUT 1,IR[1:4]    at time = 29:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 29:
0 1 0 0

Break at time = 29
MAPS>o 1,a_reg[1:8];
  O 1,A_REG[1:8]    at time = 29:
0 0 0 0 1 0 0 0                                    {DONE.  A register got the 8}

MAPS>p;                                            {Proceed through loop}
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 34:
0 0 0 0 0 0 1 0

  OUT 1,IR[1:4]    at time = 34:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 34:
1 0 1 1

Break at time = 34
MAPS>p;
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 39:
0 0 0 0 0 0 1 1
```

```
  OUT 1,IR[1:4]    at time = 39:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 39:
0 0 0 0                                        {Ready for next instruction}

Break at time = 39
MAPS>i 1,ad[1:8]:=13;                          {13 is PUSH instruction}
MAPS>o 1,b_reg[1:8] at 43 every 5;             {Since this means "push" A reg
MAPS>p;                                         into B reg, we will want to
  O 1,B_REG[1:8]    at time = 43:               watch the B register}
U U U U U U U U

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 44:
0 0 0 0 0 0 1 1

  OUT 1,IR[1:4]    at time = 44:
1 1 0 1

  OUT 1,AS DS CD RW    at time = 44:
0 1 0 0

Break at time = 44
MAPS>p;
  O 1,B_REG[1:8]    at time = 48:
U U U U U U U U                                {It takes a bunch of clock cycles}

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 49:
0 0 0 0 0 0 1 1

  OUT 1,IR[1:4]    at time = 49:
1 1 0 1

  OUT 1,AS DS CD RW    at time = 49:
0 1 0 0

Break at time = 49
MAPS>p;
  O 1,B_REG[1:8]    at time = 53:
U U U U U U U U
```

```
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 54:
0 0 0 0 0 0 1 1

  OUT 1,IR[1:4]    at time = 54:
1 1 0 1

  OUT 1,AS DS CD RW    at time = 54:
1 0 0 0

Break at time = 54
MAPS>p;
  O 1,B_REG[1:8]    at time = 58:
U U U U U U U U

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 59:
0 0 0 0 0 0 1 1

  OUT 1,IR[1:4]    at time = 59:
1 1 0 1

  OUT 1,AS DS CD RW    at time = 59:
0 0 0 0

Break at time = 59
MAPS>p;
  O 1,B_REG[1:8]    at time = 63:
U U U U U U U U

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 64:
0 0 0 0 0 0 1 1

  OUT 1,IR[1:4]    at time = 64:
1 1 0 1

  OUT 1,AS DS CD RW    at time = 64:
0 1 0 0

Break at time = 64
MAPS>p;
  O 1,B_REG[1:8]    at time = 68:
0 0 0 0 1 0 0 0                                    {DONE!  8 was PUSHed}
```

```
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 69:
0 0 0 0 0 0 1 1

  OUT 1,IR[1:4]    at time = 69:
1 1 0 1

  OUT 1,AS DS CO RW    at time = 69:
0 1 0 0

Break at time = 69
MAPS>p;
  O 1,B_REG[1:8]    at time = 73:
0 0 0 0 1 0 0 0

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 74:
0 0 0 0 0 0 1 1

  OUT 1,IR[1:4]    at time = 74:
1 1 0 1

  OUT 1,AS DS CO RW    at time = 74:
1 0 1 1

Break at time = 74
MAPS>p;
  O 1,B_REG[1:8]    at time = 78:
0 0 0 0 1 0 0 0

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 79:
0 0 0 0 0 1 0 0

  OUT 1,IR[1:4]    at time = 79:
1 1 0 1

  OUT 1,AS DS CO RW    at time = 79:
0 0 0 0                                    {Ready for next instruction}

Break at time = 79
MAPS>i 1,ad[1:8]:=6;                       {LIT}
MAPS>p;
  O 1,B_REG[1:8]    at time = 83:
0 0 0 0 1 0 0 0
```

```
  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 84:
0 0 0 0 0 1 0 0

  OUT 1,IR[1:4]    at time = 84:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 84:
0 1 0 0

Break at time = 84
MAPS>p;
  O 1,B_REG[1:8]    at time = 88:
0 0 0 0 1 0 0 0

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 89:
0 0 0 0 0 1 0 0

  OUT 1,IR[1:4]    at time = 89:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 89:
1 0 1 1

Break at time = 89
MAPS>i 1,ad[1:8]:=7;                          (LIT value is 7 this time)
MAPS>p;
  O 1,B_REG[1:8]    at time = 93:
0 0 0 0 1 0 0 0

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 94:
0 0 0 0 0 1 0 1

  OUT 1,IR[1:4]    at time = 94:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 94:
0 0 0 0

Break at time = 94
MAPS>p;
  O 1,B_REG[1:8]    at time = 98:
```

```
0 0 0 0 1 0 0 0

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 99:
0 0 0 0 0 1 0 1

  OUT 1,IR[1:4]    at time = 99:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 99:
0 1 0 0

Break at time = 99
MAPS>o 1,a_reg[1:8];
  O 1,A_REG[1:8]    at time = 99:
0 0 0 0 0 1 1 1                                    (Success)

MAPS>p;
  O 1,B_REG[1:8]    at time = 103:
0 0 0 0 1 0 0 0

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 104:
0 0 0 0 0 1 0 1

  OUT 1,IR[1:4]    at time = 104:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 104:
1 0 1 1

Break at time = 104
MAPS>p;
  O 1,B_REG[1:8]    at time = 108:
0 0 0 0 1 0 0 0

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 109:
0 0 0 0 0 1 1 0

  OUT 1,IR[1:4]    at time = 109:
0 1 1 0

  OUT 1,AS DS CD RW    at time = 109:
0 0 0 0                                    (Ready for next instruction)
```

```
Break at time = 109
MAPS>i 1,ad[1:8]:=1;                          {ADD instruction to add A and B}
MAPS>p;
  O 1,B_REG[1:8]    at time = 113:
0 0 0 0 1 0 0 0                               {Result will end up in B}

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 114:
0 0 0 0 0 1 1 0

  OUT 1,IR[1:4]    at time = 114:
0 0 0 1

  OUT 1,AS DS CO RW    at time = 114:
0 1 0 0

Break at time = 114
MAPS>p;
  O 1,B_REG[1:8]    at time = 118:
0 0 0 0 1 1 1 1                               {DONE! 8+7=15}

  OUTPUT 1,PROGRAM_COUNTER[1:8]    at time = 119:
0 0 0 0 0 1 1 0

  OUT 1,IR[1:4]    at time = 119:
0 0 0 1

  OUT 1,AS DS CO RW    at time = 119:
0 1 0 0

Break at time = 119
MAPS>end;                                     {End of demonstration of GR2}
End of SPAMAPS execution
```

**The following simulation is just to show how assignment statements work both in SPAM and in MAPS.**

```
@gr2c
MAPS>break at 1 every 1;
MAPS>inp 1,n[1:8]:=0;                    {If the right hand side is a single "0"}
MAPS>p;
Break at time = 1
MAPS>o 1,n[1:8];
  O 1,N[1:8]    at time = 1:
0 0 0 0 0 0 0 0                          {The result is all zeros}

MAPS>inp 1,n[1:8]:=1;                    {A single "1"}
MAPS>p;
Break at time = 2
MAPS>o 1,n[1:8];
  O 1,N[1:8]    at time = 2:
0 0 0 0 0 0 0 1                          {Results in a one padded with zeros}

MAPS>inp 1,n[1:8]:=U;                    {A single "U"}
MAPS>p;
Break at time = 3
MAPS>o 1,n[1:8];
  O 1,N[1:8]    at time = 3:
U U U U U U U U                          {Is extended to left side of the word}

MAPS>inp 1,n[1:8]:=X;                    {So is a single "X"}
MAPS>p;
Break at time = 4
MAPS>o 1,n[1:8];
  O 1,N[1:8]    at time = 4:
X X X X X X X X

MAPS>inp 1,n[1:8]:=1 0 X U;              {Any other value}
MAPS>p;
Break at time = 5
MAPS>o 1,n[1:8];
  O 1,N[1:8]    at time = 5:
0 0 0 0 1 0 X U                          {Is padded with zeros}
```

```
MAPS>inp 1,n[1:8]:=0 X;                    {ANY other value}
MAPS>p;
Break at time = 6
MAPS>o 1,n[1:8];
  O 1,N[1:8]    at time = 6:
0 0 0 0 0 0 0 X                            {!}

MAPS>inp 1,n[1:8]:=72;                     {A decimal number works too}
MAPS>p;
Break at time = 7
MAPS>o 1,n[1:8];
  O 1,N[1:8]    at time = 7:
0 1 0 0 1 0 0 0

MAPS>end;
```

### C.2.2) A Bigger Picture

The description on the following pages contains no behavior. It is simply multi-level description of the GR2 chip. Since the documentation produced for this chip was rather long, only the hierarchical map, the composition cell floor plans, and the GR2 pin diagram are included in this report.

```
!=============================================================;
!                                                             ;
!                      G R 2   SPAM DESCRIPTION               ;
!                              August 1980                    ;
!=============================================================;

CELLDEF GR2(
        TOP    reset,rdy,ph1,ph2,s1,s2,din
        RIGHT  GND,VDD
        BOTTOM ds,cd,rw,as,dout,ad[8]
        );

        INPUT  reset,rdy,s1,s2,din;
        OUTPUT ds,cd,rw,as,dout;
        IO     ad;
        CLOCK  ph1,ph2;
        POWER  VDD;
        GROUND GND;

    COMPONENT controller,data_path,bus_pads,lower_control_pads,
              upper_control_pads;

    RIGHT controller ABUTS LEFT upper_control_pads,
                           LEFT data_path;
    TOP data_path ABUTS BOTTOM upper_control_pads;
    TOP lower_control_pads ABUTS BOTTOM controller;
    RIGHT lower_control_pads ABUTS LEFT bus_pads;
    TOP bus_pads ABUTS BOTTOM data_path;

    BEHAVE LATER
ENDDEF;
```

```
CELLDEF controller(
        BOTTOM VDD,GND, ds,cd,rw,as, ph1,ph2, portenin,portenout, dout
        RIGHT  VDD,GND, reset,ph2,ph1,na[4],rdy,ph1,ph2,s2,s1,din,
               s1,din,ircntl[3],flags[4],aluouten,aluop[9],acntl[3],
               bcntl[3],basecntl[3],topcntl[3],pccntl[3],tmpcntl[3],
               portcntl[3],portenin,portenout,s2,dout,ph2,GND
        );

        INPUT  reset,na,rdy,s2,s1,din,flags;
        OUTPUT ds,cd,rw,as,portenin,portenout,dout,ircntl,aluouten,aluop,
               acntl,bcntl,basecntl,topcntl,pccntl,tmpcntl,portcntl;
        CLOCK  ph1,ph2;
        POWER  VDD;
        GROUND GND;

    COMPONENT rom,input_latch,output_latch,microsubroutine_latch,
              wiring1;

    RIGHT rom ABUTS LEFT input_latch,
                    LEFT output_latch;
    LEFT microsubroutine_latch, LEFT wiring1 ABUTS
    RIGHT input_latch, RIGHT output_latch;

    BEHAVE LATER
ENDDEF;
```

```
CELLDEF data_path(
        LEFT    s1,din,ircntl[3],flags[4],aluouten,aluop[9],acntl[3],
                bcntl[3],basecntl[3],topcntl[3],pccntl[3],tmpcntl[3],
                portcntl[3],portenin,portenout,s2,dout,ph2,GND
        TOP     ph1,ir[4],VDD,GND
        BOTTOM  port[8],GND
        );

        INPUT   s1,din,ircntl,aluouten,aluop,acntl,bcntl,basecntl,topcntl,
                pccntl,tmpcntl,portcntl,portenin,portenout,s2;
        OUTPUT  flags,dout,ir;
        IO      port;
        CLOCK   ph1,ph2;
        POWER   VDD;
        GROUND  GND;

    COMPONENT control_drivers,instr_reg,alu,store[6,1](REGISTER),
              port_reg;

    BOTTOM instr_reg <=> TOP alu;
    BOTTOM alu <=> TOP store;
    BOTTOM store <=> TOP port_reg;
    RIGHT  control_drivers <=> LEFT instr_reg,
                               LEFT alu,
                               LEFT store,
                               LEFT port_reg;

    BEHAVE LATER
ENDDEF;
```

```
CELLDEF lower_control_pads(
        TOP     VDD,GND,dsin,cdin,rwin,asin,ph1,ph2,portenini,portenouti,
                doutin
        RIGHT   portenin,portenout,GND,VDD
        BOTTOM  ds,cd,rw,as,dout
        );

        INPUT   dsin,cdin,rwin,asin,portenini,portenouti,doutin;
        OUTPUT  ds,cd,rw,as,dout,portenin,portenout;
        CLOCK   ph1;
        POWER   VDD;
        GROUND  GND;

    COMPONENT wiring2,ds(left_padout),opads[1,4](PADOUT);

    BOTTOM wiring2 <=> TOP ds,TOP opads;
    RIGHT ds <=> LEFT opads;

    BEHAVE LATER
ENDDEF;
```

```
CELLDEF bus_pads(
        LEFT   portenin,portenout,GND,VDD
        TOP    port[8],GND
        BOTTOM ad[8]
        );

        INPUT  portenin,portenout;
        IO     port,ad;
        POWER  VDD;
        GROUND GND;

    COMPONENT wiring3,ad[1,7](pad_tristate),right_pad_tristate;

    BOTTOM wiring3 ABUTS TOP ad, TOP right_pad_tristate;
    RIGHT ad ABUTS LEFT right_pad_tristate;

    BEHAVE LATER
ENDDEF;
```

```
CELLDEF upper_control_pads(
        LEFT   VDD,GND,reset,ph2,ph1,na[4],rdy,ph1,ph2,s2,s1,din
        TOP    resetin,rdyin,ph1,ph2,s1in,s2in,dinin
        RIGHT  GND,VDD
        BOTTOM ph1,ir[4],VDD,GND
        );

        INPUT  resetin,rdyin,s1in,s2in,dinin;
        OUTPUT reset,na,rdy,s2,s1,din,ir;
        CLOCK  ph1,ph2;
        POWER  VDD;
        GROUND GND;

    COMPONENT control_pads,power_pads,wiring4,connect_box,buffer;

    BOTTOM control_pads ABUTS TOP wiring4,
                              TOP power_pads;
    BOTTOM wiring4 ABUTS TOP connect_box;
    RIGHT connect_box ABUTS LEFT buffer;
    RIGHT buffer ABUTS LEFT power_pads;

    BEHAVE LATER
ENDDEF;
```

**The following pages contain some of the documentation produced by SPAM:**

GRBIG.TXT by SEGAL 1981-01-21

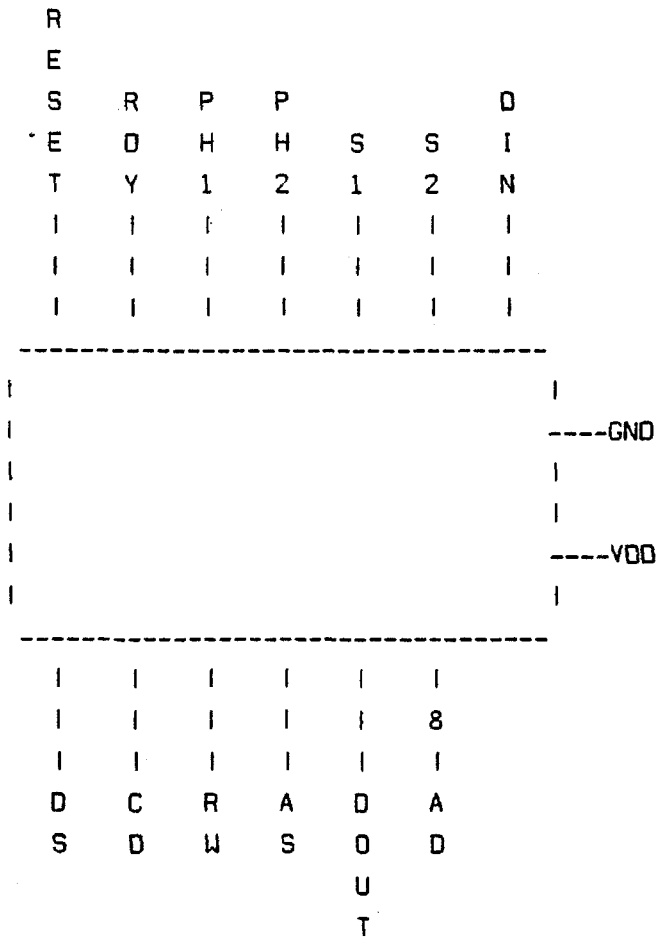
Structure Map:

```
GR2
     |CONTROLLER
     |
          |ROM
          |
          |INPUT_LATCH
          |
          |OUTPUT_LATCH
          |
          |MICROSUBROUTINE_LATCH
          |
          |WIRING1
          |
     |DATA_PATH
     |
          |CONTROL_DRIVERS
          |
          |INSTR_REG
          |
          |ALU
          |
          |STORE[6,1](REGISTER)
          |
          |PORT_REG
          |
```

```
!BUS_PADS
!
        !WIRING3
        !
        !AD[1,7](PAD_TRISTATE)
        !
        !RIGHT_PAD_TRISTATE
        !
!LOWER_CONTROL_PADS
!
        !WIRING2
        !
        !DS(LEFT_PADOUT)
        !
        !OPADS[1,4](PADOUT)
        !
!UPPER_CONTROL_PADS
!
        !CONTROL_PADS
        !
        !POWER_PADS
        !
        !WIRING4
        !
        !CONNECT_BOX
        !
        !BUFFER
        !
```

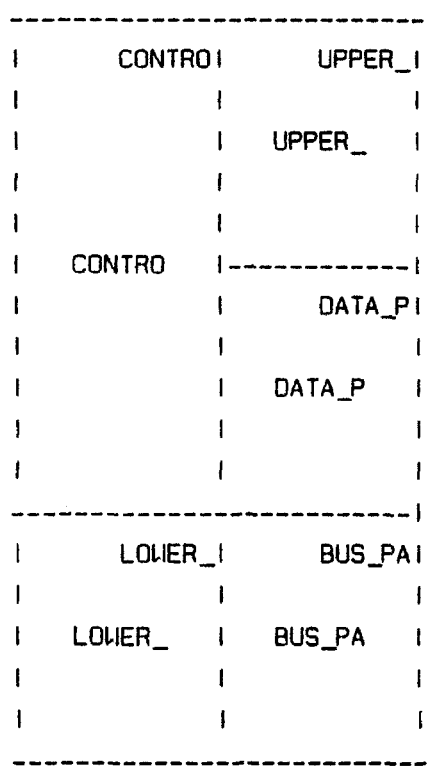
GRBIG.TXT by SEGAL 1981-01-21

Cell Specification of 'GR2':



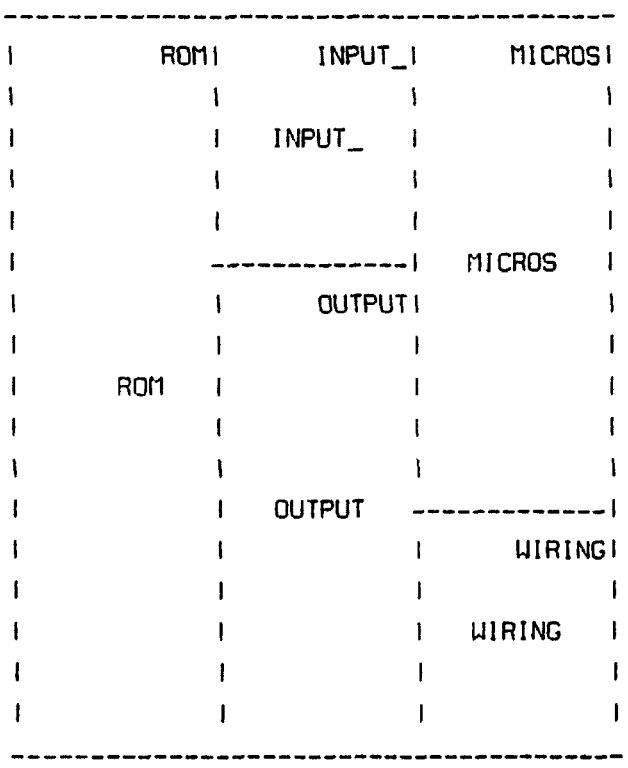
GRBIG.TXT by SEGAL 1981-01-21

Floor Plan of 'GR2':



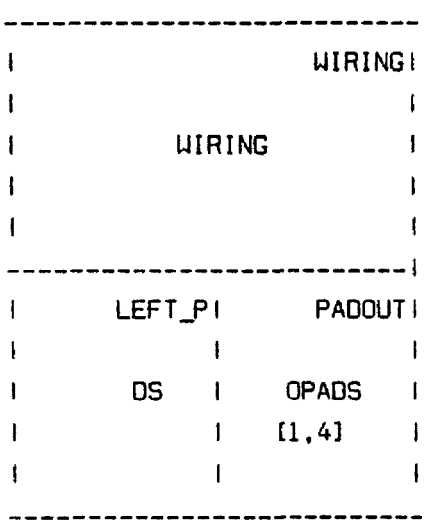
GRBIG.TXT by: SEGAL 1981-01-21

Floor Plan of 'CONTROLLER':



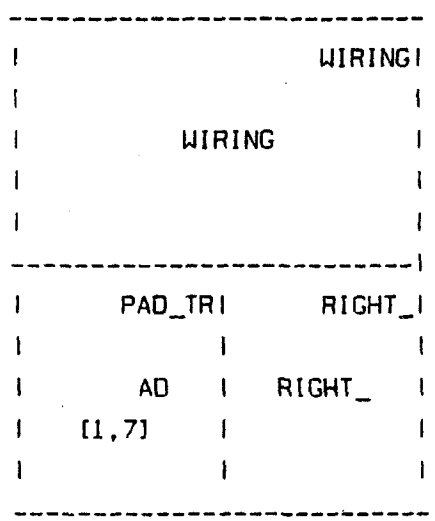
GRBIG.TXT by SEGAL 1981-01-21

Floor Plan of 'LOWER_CONTROL_PADS':



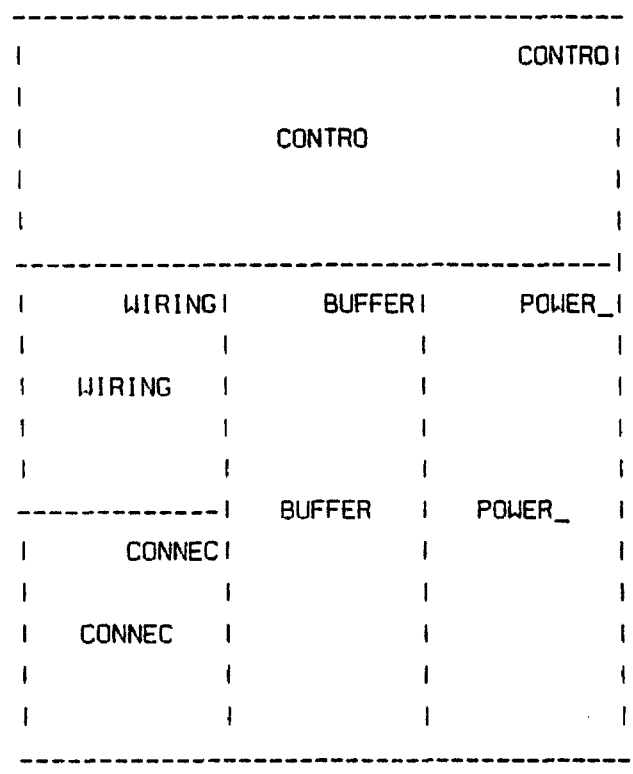
GRBIG.TXT by SEGAL 1981-01-21

Floor Plan of 'BUS_PADS':



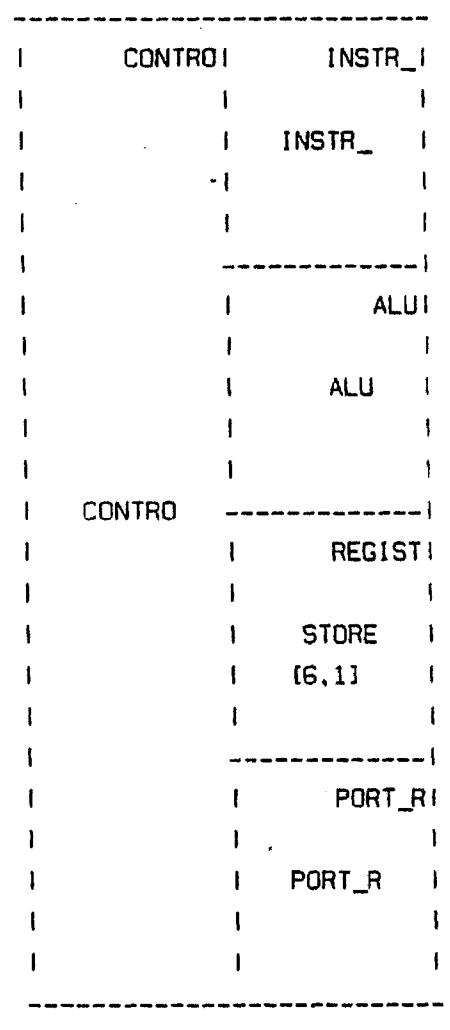
GRBIG.TXT by SEGAL 1981-01-21

Floor Plan of 'UPPER_CONTROL_PADS':



GRBIG.TXT by SEGAL 1981-01-21

Floor Plan of 'DATA_PATH':



## References


C. Gordon Bell and Allen Newell, "Computer Structures: Readings and Examples," McGraw Hill, 1971

Irene Buchanan and John Gray, "Models For Structured IC Design," Caltech SSP File #3230, 1979

W.E.Cory, J.R.Duley, W.M.vanCleemput, "An Introduction to the DDL-P Language," Stanford Computer Systems Laboratory Technical Report no. 163, 1979

Greg Efland and Richard Mosteller, "Stack Data Engine," Caltech SSP File #3364, 1979

Dave Johannsen, "BRISTLE BLOCKS: A Silicon Compiler," Caltech SSP File #2587, 1979

Danny C. Ko, "BUILD User's Manual," Burroughs Corporation - Mission Viejo, 1979

Carver Mead and Lynn Conway, "Introduction to VLSI Systems," Addison Wesley, 1980

Jim Rowson, "Understanding Hierarchical Design," Phd. Thesis, Caltech S File #3210, 1980

Steve Trimberger, "Proposed Sticks Standard," Caltech SSP Display File #38 1980