CPS 420--COMPUTER ARCHITECTURE

LABORATORY NOTES

0. Preface

These notes contain reference material to support the laboratory activity in Computer Architecture. It is presupposed that the objective of the laboratory project work is to complete a working digital computer of sufficient power to execute non-trivial programs generated by a standard compiler.

The key laboratory element is the digital simulator described in Chapter 2 and its built-in components listed in the Appendix. Chapter 1 describes the support for combinational network design and implementation. Chapter 3 describes the SPARC microprocessor and its relations to programming at both the C++ and assembly-language levels.

Many of the changes to the simulator and these notes reflect the suggestions of students. Making things easier, more interesting or extending the capabilities were the usual bases of the suggestions. Some students use the simulator and design tools in activities other than in this laboratory--other classes, hobby activities or on the job, and their suggestions are welcomed also.

The simulator has been used to model large networks. Several computers complete with disk I/O and primitive operating systems that execute code produced by standard compilers have been constructed. Some advanced students have completed projects using over 15,000 components. These large networks require a lot of computing time, sometimes tens of minutes just to read in the network. In your sim usage, you probably will not have designs extending much beyond several hundred components, but even these few will allow you to implement a complete computer.

Some of the software tools are also available to run on PC's. Students have used the PC version of sim on many different kinds of computers, from Tandy laptops to 486's. PC limitations usually come from the smallness of memory and sometimes the slowness of the processor, although, in some instances, even 386 PC's have performed better than workstations.

Development of these design tools is continuing. There may never be a low level click, drag and connect tool for the simulator components. For very small networks this is an interesting mode in which to build, but in larger designs, such as complete computers as done in 420, placing things together in this bottom/up manner is not very helpful. A top/down version of interactive network construction, based on xfig, is being developed for experimentation. The xfig output will be translated to the simulator netlist format automatically.

Some simulator components can effectively use a more interactive Graphical User Interface (GUI) than the base simulator provides. The first such component (now available) is SimSwitch, a set of switches activated by "clicking". Components in this category are being introduced as separate (Unix) processes, with asynchronous interfacing to the simulator.

The simulator has been repaired, modified and extended many times, but alas, will never be bug free. A section reporting known bugs is at the end of Chapter 2 of these notes. You may encounter one or more of these during the term. Your TA and/or instructor may be able to confirm your encounter with a known or new bug, and provide you with a work-around and/or deploy the repair crew.

0.1. Recent Changes

This edition adds some larger Ram's to accommodate SPARC memory implementation. A few new primitive components have been added to facilitate the implementation of 32-bit processors, SPARC in particular.

New asynchronous pipe components have been added to facilitate bridging GUI components to sim.

Edition 5.0. August 1992. Professor Richard J. Reid, Computer Science Department, Michigan State University, E. Lansing, MI 48824.

Chapter 1: Combinational Networks

1. Introduction

Combinational networks are collections of gates that form functions of sets of input variables. For combinational networks the output is always known when the input values are known, and the output is independent of any past states of the inputs. That is, the outputs are a decoding of the present *combination* of input values.

This chapter describes programs that are available to assist in the design of combinational networks. These programs are:

rtk -- (Real-time Karnaugh) This program is run in the OpenLook environment and provides the real-time algebraic output for a mouse-click-entered Karnaugh map.

mdpic -- (Minterm/don't-care/prime-implicant/cover) accepts lists of ON and Don't-Care terms and produces a minimal cover for the function.

cnd -- (Combinational Network Design) Accepts combinations of ON, OFF and Don't-Care terms and finds covers adhering to given constraints.

rd -- (Reduce) Provides a more general processor for reducing symbolic Boolean expressions. All conceivable logical operators are supported.

bft -- (Boolean Form Translator) Converts input forms to output forms where the forms include: algebraic, minterm, and cubical-complex.

net -- (Network) Accepts a two-level, sum-of-products Boolean expression and produces a netlist suitable for input to *sim* (the simulator).

1.1. Real-Time Karnaugh Map

The program *rtk* (Real-Time Karnaugh Map) translates binary Boolean algebraic functions entered graphically to their minimal, sum-of-products algebraic form. The cells of the Karnaugh Map are "clicked" to their values of 0, 1 or "don't care" (-), and the minimal functional form follows the entries.

The implementation of *rtk* is the same as that of *mdpic* described later in this chapter.

1.2. Minimized Networks

The program *mdpic* (minterm/don't-care/prime-implicant/cover) has been developed to accept minterm lists of ON terms and don't-cares, and produces a minimal cover for the function using a subset of the prime implicants. The minterms are given as decimal values. Mdpic is used as:

```
mdpic  [-v]
```

taking its input from standard input, and writing the output to standard output. The optional -v (verbose) flag provides a commentary on the steps of prime implicant determination and the selection of the minimal cover. The commentary is written on the standard output.

Functions can have a maximum of eight variables. The function result is expressed in sum-of-products form using the variable identifiers: a, b, c, ...

The cubical complex method of prime implicant determination is used. Cubical complex notation uses a list of implicants similar to sum-of-products form. Each cube denotes which literals are present in true form by the appearance of a one, and those present as complements by a zero. If a literal is absent in a term, an "X" represents its vacant position. For example, the function:

a b' + a' c + a' b c'

is given as:

(10X) (0X1) (010)

Prime implicant determination begins with the single implicant, (XX...), the universe, and progressively removes (sharp-product) individual vertices. This single vertex (at a time) removal leaves the remaining cubes (implicants) as large as possible, and when completed leaves prime implicants. Some intermediate cubes are generated that are non-prime, but these are always covered by single implicants when they arise, and consequently, are easily recognized and removed.

Covering is done using essential-prime-implicant determination, and row and column dominance criteria. If a cyclic covering table arises, terms with fewer literals are considered less expensive, and their use is given preference. If a cycle should have all equally-expensive terms, an arbitrary one is selected for discard. The covering procedure assumes terms with fewer literals are less expensive.

1.3. Combinational Network Design

The program *cnd* (Combinational Network Design). has been developed to accept lists of ON, OFF, and Don't-Care (DC) terms, and produce a minimal sum-of-products cover for the function using a subset of the prime implicants.

This program supports single-output functions using two-valued logic. The design of networks with a large number of input variables is accommodated by this program. There is also a wide latitude of input specifications and finely resolved control over the solution process.

The commentary options provided by *cnd* are also a reason for its existence. The commentary is intended to supplement and re-enforce studies in combinational network design. The commentary is brief and is intended to be meaningful and helpful to students in Logical Design, Digital Design, Computer Engineering and Computer Architecture courses.

1.3.1. Notation

Program input and output uses cubical-complex notation. The individual product terms in the sum-of-products form are called cubes. Each cube denotes which literals are present in true form by the appearance of a one (1), and those present as complements by a zero (0). If a literal is absent in a term, an "X" represents its vacant position. For example, given the function:

$$ab'd + bc'd' + a'd$$

The corresponding cube notation would be:

10X1 X100 0XX1

1.3.2. Input Specifications

The input is free form, with cubes separated by white-space. The sets of cubes are given in three partitions corresponding to the ON, DC and OFF terms successively. Each of the three partitions is terminated by a term with all variables missing, e.g. XXXX (for 4 variables), if another partition will occur, or by the end-of-file.

Any combination of ON, DC and OFF partitions can be used; using X... partition markers allows partitions to be empty. They must be given in that order, with X... cubes separating the partitions. This ordering facilitates moving the specifications between the ON and OFF partitions to alternatively evaluate the complement function. A warning about any overlaps between partitions will be given on the standard output.

The operator notation used in describing some of the operations is given in the following table:

Function Operators	
Operator	Description
~	Complement
|	Union
&	Intersection

Comments in the input file can only be at the beginning of the input file, and are distinguished by a hash (#) as the first character on a line.

1.3.3. Program Use

The program reads from stdin and writes to stdout and is executed using:

```
cnd  [-abcCeElnpv]  [-o <filename> ]  <in-file  >out-file
```

The optional [ ] command line flags can be given in any order and provide:

a: If branching is used for resolving cyclic structures, give ALL the solutions. Without this flag, only one of the solutions with the fewest terms and fewest literals will be reported.

b: This flag *inhibits* branching if a cyclic structure occurs during the covering process.

If branching is inhibited, and a cyclic covering structure arises, terms with fewer literals are considered less expensive, and their use is given preference. If a cycle should have all equally-expensive terms, an arbitrary one is selected for discard. The procedure assumes terms with fewer literals are less expensive. With no branching, only one solution will be produced, and it generally will not be the minimal solution.

With branching (the default) all possible (but see the -p {pruning} flag discussed below) covers are considered from the point where the cyclic structure is encountered.

The default is "branching enabled" so cnd won't inadvertently (on the user's part) produce a non-minimized solution. However, the computation time grows substantially when all these possible solutions are examined.

c: This commentary flag provides brief comments on the steps of the solution process. The comments are intended to reinforce the users' knowledge about combinational network design. The commentary is written on the standard output.

C: This Commentary flag provides more detailed comment about the steps of the solution process. This flag turns on the c-flag above.

e: Echo user comments, that may appear at the beginning of the input file, to the standard output.

E: Echo the input cubes to the standard output.

l: This flag *inhibits* the use of the "less-than" relation for discarding prime-implicants during the covering process. A cube is deemed "less-than" another, and is discarded, if it costs no more than the other, and covers no more (of what remains to be covered, at a given stage of covering) than the other. Cost is the count of literals (true's plus complement's) present in a cube.

n: Find a non-redundant cover, rather than a minimum cover. This option can execute significantly faster than pushing to the "minimum" cover. The cover found will be a subset of the prime-implicants that has no redundant (extra PI's) covering of the ON array.

o: Direct the solution(s) to a separate output file.

p: This flag *inhibits* the pruning that would normally occur during branching within cyclic covers. The default, pruning enabled, terminates a branching alternative exploration when a partial-cover being developed becomes larger than a known minimum-cube cover already found.

v: Verify that the results obtained are, at least, a valid covering. Computes the cubes that were in ON and are not in the Cover reported, and computes the cubes in the Cover reported that are outside ON and DC. Both computations should yield null results, thus establishing cover equivalence.

1.3.4. Performance

Single-output functions can have a maximum of 32 variables on Unix and 16 on DOS (PC) systems. Functions of many variables can take a very long time to minimize. The present performance on a Sun SPARC 1 shows that with seven variables, using no command-line options, common minimization times are just over one second. From this point upward, expect each additional variable to take several times as long as the one-fewer-variable minimizations.

1.3.5. Solution Method -- The solution process is as follows:

1) Read the input. Compress the given arrays using consensus.

2) Process the input specifications.

   ON, DC and OFF--all given:

   All vertices must be therein somewhere.

   No overlaps are allowed.

   ON and OFF given:

   DC is determined as: DC = ˜ (ON | OFF).

   OFF not given:

   DC can be given as overlapping ON, but will be trimmed to: DC = DC - ON.

   ON not given:

   ON is formed as: ON = ˜ (DC | OFF).

   DC can overlap OFF, but will be trimmed to: DC = DC - OFF.

   By entering what is really the ON as the OFF array, you get a solution for product-of-sums, or alternatively, can use the solution and the output-complement gate that's common on PLA's.

3) Form the union the ON and DC cubes.

4) Find the prime-implicants of the above union using generalized consensus.

5) Drop those prime-implicants totally outside the ON space.

6) Add to the Partial-Cover those prime-implicants which are essential.

   a) For the non-redundant cover option, choose a subset of prime-implicants that cover ON with no redundancy. The cover is completed with this step.

7) [Flag-controlled optional step] Discard those prime-implicant cubes that are "less-than" others.

8) Loop back to Step 6) while the function is not covered and the Cover is continuing to expand towards a solution.

9) If not covered at this point, then there exists a cyclic covering structure.

   a) Branching Option Selected (default)--For each PI cube not used, alternatively: 1) force that cube into the Partial-Cover (if it would not be redundant), and, 2) discard that cube. For each alternative disposition, loop back to Step 6), possibly returning here recursively, for further covering cyclic-structures, to process other cubes in this same two-alternatives manner.

      If the all-solutions (-a) flag is not specified, and the partial cover grows beyond the smallest solution thus far, prune the branching at that point.

   b) Branching Option Not Selected--Find the maximum cost cube of the unused PI cubes, and throw it away. Loop back to Step 6), possibly returning here recursively, to process other cubes in this draconian manner.

1.3.6. Troubles

If you encounter problems using cnd, please:

- Turn on the verify option to check that you actually got a legitimate cover.
- If you obtained a legitimate cover, but don't believe you have a minimum cover, review your use of the command-line options, possibly trying other combinations.
- Turn on the commentary option(s) and see if you can discern any incorrect steps.
- Last resort: Pare down your problem to six or fewer variables still exhibiting the fault, and e-mail this reduced problem along with your comments about the nature of the failure.

  If it's not feasible to pare down your problem, just e-mail your comments about your suspicions.

1.4. Expression Reduction

The program *rd* (reduce) has the ability to process Boolean expressions symbolically, to allow an expressive set of operators, and to reduce the expressions effectively and provides a valuable design and validation tool.

Boolean expressions could be formed from just a few operators, as long as the operators chosen represented complete gate sets. A better design tool results when all the operators that might be expected to occur naturally, in various problem domains, are allowed.

The following table of functions of two variables gives the common, and some not-so-common, binary operators.

FUNCTIONS OF TWO VARIABLES								
(a,b)				F(a,b)	Binary Operator	Operator Name	Associative	Commutative
00	01	10	11					
0	0	0	0	0				
0	0	0	1	a b	*	And	Y	Y
0	0	1	0	a b’	-’	Inhibited-by	N	N
0	0	1	1	a				
0	1	0	0	a’ b	’-	Inhibits	N	N
0	1	0	1	b				
0	1	1	0	a’ b + a b’	^	Exclusive-or	Y	Y
0	1	1	1	a + b	+	Or	Y	Y
1	0	0	0	(a + b)’	+’	Nor	N	Y
1	0	0	1	a’ b’ + a b	<->	Equivalence	Y	Y
1	0	1	0	b’				
1	0	1	1	a + b’	<-	Implied-by	N	N
1	1	0	0	a’				
1	1	0	1	a’ + b	->	Implies	N	N
1	1	1	0	(a b)’	*’	Nand	N	Y
1	1	1	1	1				

The program *rd* reduces Boolean expressions to standard two-level, sum-of-products form.

The input Boolean expression for *rd* is formed using the operators given in the following table.

OPERATORS, ASSOCIATIVITY AND PRECEDENCE			
Symbol	Operation	Associativity	Precedence
*	And	Left	1
+	Or	Left	0
’	Not	Left	2
*’	Nand	Left*	1
+’	Nor	Left*	1
-’	Inhibited-by	Left	0
’-	Inhibits	Left	0
^	Exclusive-or	Left	0
->	Implies	Left	0
<-	Implied-by	Left	0
<->	Equivalence	Left	0

* Nand and Nor associativity is over their "and" and "or" constituency respectively.

Juxtaposition of variables with intervening white-space may also be used for implicit "And".

The constants (0,1) may be used freely in expressions. Variables begin with a letter and continue using letters and digits. Blanks and newlines serve as delimiters when variables are juxtaposed.

Sub-expressions may be grouped using { }, [] and ().

Comments may be included using the delimiters "/*" and "*/". Comments may not be included within a variable name and they may not be nested.

The following meta operators are provided.

META OPERATORS	
Symbol	Action
%	Print the current expression
$	Convert the current expression to sum-of-products form
@	Convert the current expression to and-exclusive-or form
&	Reduce the current expression (limited to covering relations)
#	Convert the current sum-of-products expression to prime implicants
;	Reduce, generate prime implicants, print and reset to accept another expression
EOF	Reduce, generate prime implicants and print

Rd can reduce any input expression to the sum of prime implicants. There will generally be redundancies among these implicants; however, there will be no static hazards in functions implemented using the complete set.

Rd reads from the standard input and writes to the standard output.

1.5. Boolean Form Translation

The program *bft* (Boolean Form Translator). accepts Boolean functions given in algebraic sum-of-products, or lists of minterms, or cubical-complex notation, and produces the function translated into any of the same three forms.

This program does not remove any redundancies that may be in the input. Furthermore, redundancy is often introduced in the output when the minterm form of output is selected, since the terms of the function are expanded to minterms independently of each other. These redundancies can be removed using the Unix *sort* utility with the "unique" specifier on the *sort* command-line.

1.5.1. Notation

Algebraic Form

A function can be specified in algebraic sum-of-product form as:

$$ab'd + bc'd' + a'd$$

The input variables begin with "a" as the first variable and proceed through the variable names in the sequence a-zA-Z. If other single-character names are used in a given source function, the utility $ tr $ can be used to shift the variable names to the required range.

There can be no whitespace within a product term, and whitespace must delimit the "+" operator.

Minterm Form

A list of minterms specifies the function as:

3 17 9 25 . . .

Whitespace delimits the integers specifying the minterms.

Cubical-complex Form

Individual product terms in the sum-of-products form are called cubes. Each cube denotes which literals are present in true form by the appearance of a one (1), and those present as complements by a zero (0). If a literal is absent in a term, an "X" or "x" represents its vacant position. For example, given the function:

$$ab'd + bc'd' + a'd$$

The corresponding cube notation would be:

10X1 X100 0XX1

Whitespace delimits the cubes of the function.

1.5.2. Program Use

The program reads from stdin and writes to stdout and is executed using:

bft -n<number-of-variables> [-i[a,m,c]] [-o[a,m,c]]

The default input and output form is cubical-complex notation. The -i flag argument allows selection from the three input forms: algebraic, minterm and cube; and the -o flag argument provides selection from the three output forms.

When this program is used on a machine that uses N-bit words to represent integers, the limit on the number of variables is N-1.

Special Forms

Inputs of zero length are translated to the constant zero (0) when the algebraic form of output is selected. Input cubical-complexes of the form "XXX..." are translated to the constant one (1) when the algebraic form of output is selected.

1.6. Equations to Netlists

The program *net* has been developed to accept a two-level, sum-of-products Boolean expression and produce a network description that is suitable as sim (the simulator) input.

Net is used as:

net [-n netname] [-t]

taking its input from standard input, and writing the output to standard output. The optional -n flag allows naming the network to something other than the default "F". The optional -t flag generates a Switch/Probe test fixture for the network, and includes the fixture in the output.

The format of input expression matches that produced by rd and consists of a number of terms joined with "+", where each term is one or more juxtaposed variables. The variables may have a suffix of (') indicating complement. Net does not accept comments in the input expression. An example input expression is:

$$a\ b' + a\ c + a'\ b\ c$$

Rd, net and sim can be piped together as:

rd <arb_Bool_exp | net -t | sim

accepting an arbitrary expression, reducing to the prime implicants, and producing a Switch-input/Probe-display, testable version of the completed network.

It is not possible to specify the order of the input and output variables that net constructs. Check the resulting network carefully to ascertain what the "pin-out" actually is before attempting to interface to the constructed component. If you desire some particular "pin-out", this can be achieved by editing the network produced by net.

Chapter 2: The Simulator

2. Introduction

Modern digital-design laboratories commonly use simulation to evaluate digital networks and proposed computer architectures. Simulation is an economical method of gaining confidence that a proposed design will function as intended. When potential difficulties are discovered, they are often easily remedied, and incur minimal costs compared to rectifying problems in hardware prototypes.

Designs created for simulation have outstanding documentation and portability characteristics. Design information is text-oriented and amenable to text-processing and data-base capture and control. A design simulation can be ported to a new computer as soon as a new simulator becomes available.

The use of computers and simulation makes adding new component types convenient, allows new debugging techniques to be developed, and provides graphical interactions with networks being tested. Furthermore, these networks can be tested more completely because models of element classes can be used rather than specific individual elements.

Breadboarding using actual components is not without merit, even though a great portion of digital-circuitry development no longer passes through this stage of evaluation. The problems of hardware prototyping, including: incorrect design, misconnections, intermittent connections, grounding, shielding and thermal-effects, are valuable laboratory experiences. Most of these prototyping problems will not arise in a simulation laboratory, however, being forsaken for the evaluation of more and larger-scale designs that simulation will allow in a given amount of time.

This simulator provides a modest library of built-in primitive elements as well as the capability for macro definitions and expansions. Primitive elements such as latches, shift registers, counters, multiplexers, decoders, rams, roms and microprocessors have functional implementations. That is, these components are atomic as far as the simulator is concerned and have no internal gate models. Some of the functionally-modeled elements such as counters and registers can have any bit-width as a single component. Also, the primitive, multiple-input gate types allow an arbitrary number of inputs. Coupled with the convenience of file editing, merging and macro processing, these features make it practical to design and evaluate large digital systems.

Regardless of the power of the computer used for simulation, it is easy to tax the computational power to its limits. Large network simulations will run slowly. Only the definition of "large" varies as more powerful computers become available. The size of networks evaluated are chosen so this slowness will be reasonably tolerable.

This simulator uses three-levels of signals: zero, indeterminate, and one. Three-level simulation requires twice the computations as two-level, since each logical change requires evaluation at the intermediate level also. This is not a real sacrifice in performance because the necessary attention to transition details is an essential element of design. Two-level simulation would not illuminate some critical circuit hazards. Even three levels are too few for some situations.

A netlist describes networks. Node-names label points of interconnection, and the use of these labels as component terminal points specifies the interconnections. An example of the lowest level modeling capability is shown in Figure 1. This shows a simple two-input nand gate, excited by manual switches (shown on the left and operated by keyboard keys) and driving a digital probe (shown on the right). This is a replication of the normal screen presentation.

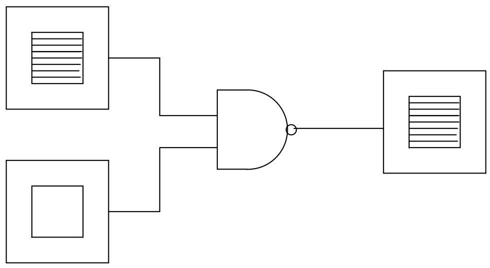


Figure 1. Two-input Nand in a Manual Test Setup

The ruled insets of one Switch and the Probe indicate a logical one in both those positions.

Networks are specified by netlists, ASCII files of component placements and their interconnections, having the following generic form:

<type> <layout_position> <inputs> <outputs> <initializing_parameters> ;

The netlist required to specify the network of Figure 1 is:

```
Switch 1a sw_x ONE;
Switch 3a sw_y ZERO;

Nand 2b sw_x sw_y out;

Probe 2c out;
```

Component placements are specified by giving the row and column coordinates of a virtual grid overlaying the display space, e.g. "1a,3a,2b,2c" above, specifying row 1 column a, row 3 column a, etc. The grid is virtual in the sense that the maximum extent of placements in the netlist is mapped to the full screen of the display.

Hierarchical collections of components can be formed as shown in Figure 2. This definition capability is implemented as a macro definition/use language facility.

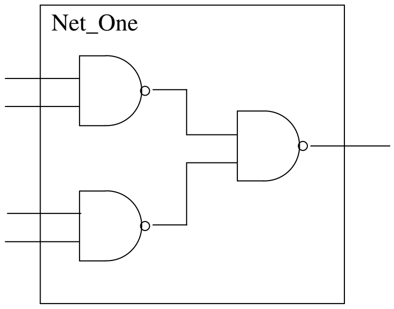


Figure 2. Hierarchical Network

The corresponding netlist defining this network would be:

```
Define Net_One input[3-0] output;

  Nand 1a input[3-2] internal_node_1;
  Nand 3a input[1-0] internal_node_0;

  Nand 2b internal_node_1 internal_node_0 output;

Endef;
```

After this definition the four-input, single-output component, Net_One, could be used in the same manner as any primitive components. Note that Net_One isn't given a placement field here in the definition, but would be in each of its instances of use. The use of subscripted signal names is also illustrated, e.g. input[3-0] specifies four signal lines: input[3], input[2], etc. in a bus-like manner. The signal names chosen for nodes within a definition is arbitrary. The names chosen are distinguished from the same signal names that might be used elsewhere, by the scoping provided by the Define...Endef block-structure organization. The names used at the interface have the same interpretation as the formal arguments in a macro. Similarly, the graphical placements within a definition are relative to the ultimate placements of instances of the defined network's positions of use.

In addition to describing the simulator, these notes describe a set of tools for designing combinational networks and creating a netlist from equations or minterm specifications. These tools have some powerful characteristics, but are limited in their scope--they do not treat multiple-output combinational networks, they offer very little help in the design of sequential machines, no help in the proposal of architectures, and no interface to VLSI layout and production mechanisms.

2.1. Netlist Form

The input for the simulator consists of a list of components indicating their interconnections. The input is free-form. White-space and comments may be used freely to enhance readability. Comments are bracketed with /* */. Comments may not be nested nor may they be included within tokens. There is a limit of 1024 characters per comment. As indicated below node names can contain the ’’ character, so be careful to separate comments from node names with white space.

The directive

```
#include "filename"
```

may be used either external to any component description or within a component’s body. In each case, it must begin in column one. This directive operates to include information from the referenced file in the position where the #include occurs. The usual usage of this feature is to allow external references to Rom contents without having to merge the Rom values into the netlist itself. This can be of particular benefit when the Rom contents are expected to change repeatedly, as they would when developmental program or microcode is being stored in the Rom. Include’s are a multi-level facility; include references may be included within include files themselves. The limit of include-nesting is presently six, if a greater depth is needed use the "soelim" Unix filter.

Fields are delimited by spaces, tabs and newlines. Individual components have the form:

```
<Name> [ "<label>" ] [ <cell-designator> ] [ <input-list> ]
   [ | ]  [ <output-list> ]  [ <symbolic-parameters> ]
    [ <numeric-parameters> ] ;
```

where [...] shows a field that may not be present in all record types.

The identifiers for component names conform to the regular expression:

```
[A-Z][-_A-Za-z0-9]*
```

Variables beginning with the prefix "ZZ" should be avoided since that prefix is used internally to create unique instances of local variables.

Screen cell positions are composed as:

```
({G}{G}"-"{G}{G}|{G}{G})
```

where {G} is defined to be:

```
[a-zA-Z0-9]
```

The cell-designator is either a two character sequence, denoting a screen row and column for placement of the component, or an extent sequence such as 1a-3b for a component occupying an extended screen region. Row and column extents on the screen will include all the ASCII characters between the extremes of those used. Row and column labeling are independent of each other and the same labels can be used for each.

Screen locations may use "dot" notation to achieve a hierarchy of relative positioning. For example, the cell designator

```
3B.2x
```

refers to row 2 column x within the cell at row 3 column B. The extent of the coordinate system within 3B depends on the usage within that cell. This can continue recursively as

```
3B.2x.1a-3c.b6.q8-r9. ...
```

More than a single component can be packed into a single cell by giving the identical cell designator for each. In these cases, the components are stacked vertically within the cell, as:

Switch 1a a ONE; Switch 1a b ZERO;

To stack components horizontally, dot notation must be used as:

Switch 1a.ab a ONE; Switch 1a.ac b ZERO;

If cell designators are not identical, they can overlay one another if they actually refer to the same space as:

Register Mp-Mp Din[3..0] Load | Dout[3..0];
Probe Mp Dout[3]; Probe Mp Dout[2];
Probe Mp Dout[2]; Probe Mp Dout[0];

placing four stacked probes overlaying the register.

Variables used as node labels for interconnections are formed as:

[-+_,.?’$%)(a-zA-Z0-9]+

Sim allows the use of singly subscripted variables in referring to nodes of interconnection. Single integer subscripts or integer ranges can be appended to node names as follows:

<node>[<decimal_integer>]
<node>[<decimal_integer>..<decimal_integer>]
or
<node>[<decimal_integer>-<decimal_integer>]

For example, a 3-bit counter connected to probes can be specified as:

Counter 4x reset clock| Out[2..0];
Probe 4y Out[2]; Probe 4y Out[1]; Probe 4y Out[0];

Non-subscripted variables and subscripted variables, with either single integer subscripts or subscript ranges, with ascending or descending range specification, can be intermixed freely. The guide to usage is whether ranges expanded to multiple, single-integer subscripted variables would be sensible in the given context.

Only non-negative, decimal integers may be used to form subscripts or subscript ranges.

2.2. Constants

There are three symbolic constants that are used in forming netlists:

ZERO, UNKNOWN, ONE

the two logical values and an intermediate, indeterminate value. (Be careful of the spelling, all the letters must be capitals.) The intermediate value can represent either a definite but unknown or an indefinite value depending on the situation. Any of these constants may be used as input values for components. The use of constants most frequently occurs when some functionality of a component is not being used. A shift-register never being used in the shift-left mode, for example, would not need a variable for its shift-left input. Sometimes, constants serve as place-holders for later connection to sub-networks yet to be implemented.

Constants are also used for setting the initial or quiescent values of Switches, Pulsers and Clocks, and for setting the arming and activation conditions for One-shot’s and Time-probe’s.

2.3. Variables

A preponderance of logical design is carried out using binary Boolean algebra. In this algebra there are two constants and any variable may take on one of two values. These two values are sufficient to characterize all the static conditions and many of the dynamic conditions that arise as well. However, in more involved designs, where register gating signals are derived from combinational circuitry for example, the hazards that may arise must be examined to ensure reliable performance.

Static hazards in combinational circuitry supplying not only register-loads, but also, counter-increment pulses, memory-write signals, and decoder-enable signals are especially menacing.

Two-level simulation is not sufficient to uncover even static hazards. Consider the function shown on the Karnaugh map below

			b	
	0	0	1	0
a	1	1	1	0
		c		

and implemented literally as:

f(a,b,c) = ab’ + bc

If the current inputs are (a,b,c)=(1,1,1), and the b variable changes to a zero, the output can be momentarily low because b goes low slightly before b’ goes high. This causes the initially "on" term, bc, to go low before the other term, ab’, comes high. In a strictly two-level analysis or simulation this hazard goes undetected since the output is one both immediately before and immediately after the change. If this function were used in a critical situation, such as an active-low register-load signal, this hazard would be disastrous. This particular hazard can be eliminated by adding the term ac to the above function.

The design process must avoid all significant hazards. One of the goals of simulation is the discovery of hazards that were not foreseen during design. Static hazards can always be eliminated by including terms covering every pair of adjacent ones on the Karnaugh map representation of the function. This ensures that the value of any single variable may change, within the sub-domain of constant function output, without the possibility of turning off all the terms of the function even momentarily.

In three-level simulation, a third, intermediate level is introduced. This third level can have at least three somewhat different interpretations, depending on the situation. In the first case, the intermediate level shows a transient condition, possibly of short duration, where the previous and following values are well defined. In a second interpretation, the intermediate level can persist for an extended interval, denoting the variance in gate switching times in producing the new output. Here, any single gate used in the circuit would switch within a small time interval, but just where this actual switching takes place varies among the same type of component according to manufacturing tolerances. In a third interpretation, the intermediate level shows a signal that might well be either a zero or one (definitely) in an actual circuit, but it is not possible to determine which. This third case arises commonly when initializing the state of flip-flops.

2.4. Component Modeling

In three-level modeling, the evaluations associated with the components processing their inputs must account for the third-level of the input signals. The symbol "?" is used here to represent the intermediate level.

The following tables show the resulting values for the primitive gates operating on inputs with three levels.

	And	in2: 0	?	1
	0	0	0	0
in1	?	0	?	?
	1	0	?	1

Or	in2: 0	?	1
0	0	?	1
?	?	?	1
1	1	1	1

Not	Out
0	1
?	?
1	0

Table I. Three-Level Representation of Gating Functions

The And behavior is to find the minimum over all its inputs, while the Or finds the maximum.

The result tables for other types of gates can be found by expressing any new gate type as a network of And’s, Or’s and Not’s, and finding the overall responses from the responses for the constituent elements.

The above tables alone are not sufficient for a realistic model of any gates however. They only reflect the gate's consideration of the current values of the inputs. A second factor of the model arises if a change in the current output should be called for as a result of the current evaluation of the inputs. To conservatively model the time-delays associated with signals propagating through gates, two times are included in the model. One is the shortest delay that any gate of the given type might have for its propagation. This shortest delay is the value used if the new signal that is to take effect is an uncertainty ("?"). The second delay known to the model is the longest delay that is possible. The longest delay is used if the new signal that is to take effect is a certainty--either a zero or a one. The claim for conservatism stems from propagating uncertainties most rapidly and certainties most slowly. The range reflects the normal manufacturing tolerances and usage degradation in the delay times exhibited by a given gate type.

The event queueing mechanism acts in the following manner in deciding on the fate of any proposed signal queueing:

Proposed Value	Present Value	Action
0	0	No queueing.
0	?	Schedule "0" after maximum delay.
0*	1	Schedule "?" after minimum delay, and "0" after maximum delay.
?	0	Schedule "?" after minimum delay.
?	?	No queueing.
?	1	Schedule "?" after minimum delay.
1*	0	Schedule "?" after minimum delay, and "1" after maximum delay.
1	?	Schedule "1" after maximum delay.
1	1	No queueing.

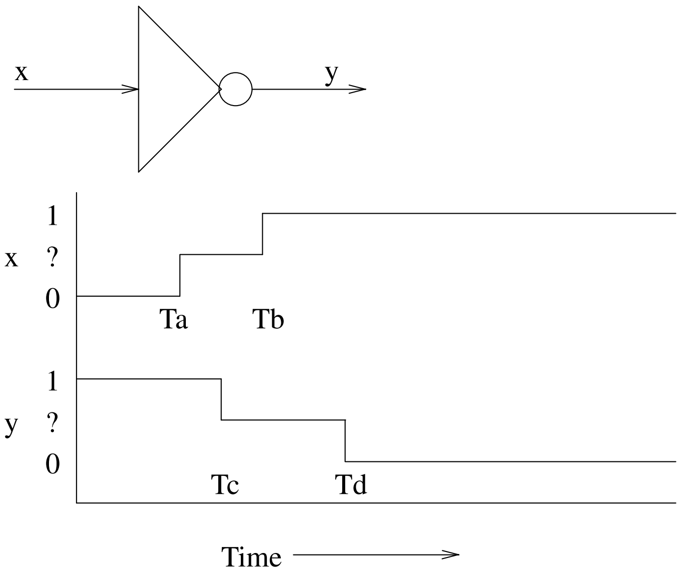
Table II. Queueing Function Representing Gate Delays

*These are unusual combinations, but they can arise if sources such as switches or clocks are modeled as having zero transition times. In any case, the Action given for these uncommon combinations is conservative and completes the disposition for all possible combinations.

2.5. Signal Uncertainties and Component Delays

Consider the inverter shown below and assume it receives the input signal X. X is making a transition from zero to one, and there is an uncertainty interval associated with this transition. Let X be uncertain from time Ta until time Tb. In most cases, this interval would not be the transition time for the signal involved; the transition time would be much shorter. The uncertainty interval usually arises because of component specification tolerances. That is, X probably changes rapidly somewhere in this interval, but it's not known exactly when this transition occurs. It may be anywhere in the interval as different actual components producing this signal are used in the network.

Assume the inverter has a minimum delay of Tmin and a maximum delay of Tmax.



Gate Delay Adding to Accumulated Uncertainty

In keeping with the conservative approach of propagating uncertainties as rapidly as possible and known levels as slowly as possible, the times for the output changes become:

$$Tc = Ta + Tmin$$

and

$$Td = Tb + Tmax$$

The uncertainty interval for the output is given as:

$$Td - Tc = Tb - Ta + Tmax - Tmin$$

This uncertainty interval compounds as levels of gates propagating uncertainties are cascaded. The termination of these propagating uncertainties by gates with dominating certain-inputs, such as And's with Zero inputs or Or's with One inputs can bound these accumulations of uncertainty.

2.6. High Impedance Outputs

The outputs of logic elements cannot be connected together because of the resulting ambiguity of the shared output. Electrically, this would amount to a tug-of-war as the separate components strove to establish the common output node according to their individual determinations.

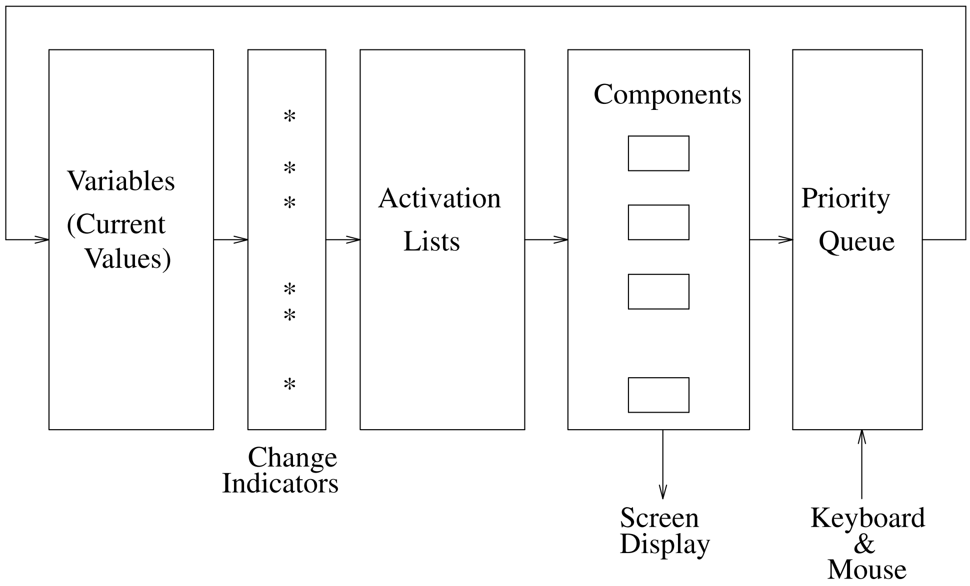
There are electrical elements, however, that can enter a high-impedance state. In this state they do not assert a logical value for their output. These types of elements can be safely connected together as long as only one gate's output is asserted for any node. The remaining outputs connected to the same node must be in the high-impedance state. Of course all the gates connected to a node could be in the high-impedance state, and the logical level of the node would be indeterminate.

Any of the signals in the simulator can enter the high impedance state provided they are driven by components whose outputs can enter this state.

2.7. Simulation Model

To conserve the limited computational power available, evaluations must be done only as necessary. An event-driven simulation accomplishes this conservation.

The netlist is translated to an internal form that executes on the simulation model shown in Fig. 3.



Simulation Model

The design features of this model are:

1. The variable values must maintain their proper time relations, there can be no artifactitious time-skew. This is an absolute, *sine qua non,* requirement.
2. Consult components as infrequently as possible for evaluation of their outputs based on the current inputs.
3. Limit or prevent iterative searches for any signal value or component.
4. Time order future events, variable-value changes, in the Priority Queue in an efficient manner to prevent extensive searches for event placement or retrieval.
5. Prevent unnecessary Priority Queue entries that merely repeat the currently scheduled or present values of variables.
6. In special cases, recall previously-scheduled Priority Queue events, and replace them with the appropriate new determinations.
7. Terminate gate evaluations where possible. For example, when evaluating an "And" gate, the discovery of a zero input obviates examination of any remaining inputs.
8. Components represented functionally, such as registers, can have their evaluation abbreviated if the load signal is at the refractory level, when activation has occurred because of changes in the input data lines.

The model flows as follows. The variables are updated to their values for the current time by extracting all the "due" events from the Priority Queue. Those variables that change are marked, and after all changes appropriate to the now-current time are effected, every component that has just had an input variable updated is evaluated. As components produce outputs changed from their previous determination, these changed values are entered into the Priority Queue, time-stamped with the future time, based on device delays, they are to become effective.

As part of the netlist analysis, a set of activation lists is prepared. Each list of the set lists the components that are activated by a particular signal. When a new value for a node is taken from the event-queue, all the components in that node's activation list are marked for evaluation. When all the nodes changing at a particular time have been updated, the marked components are evaluated.

There is some adjudication necessary in bringing new variable values from the Priority Queue to the Current Values list in the case of tri-state signals. If the last effective change of a variable value to a definite value (non high-impedance) was by component i, then i is the current prescriber for the value for that variable. Other components may not assert any definite values for that variable until component i relinquishes control by asserting high-impedance for its output. Failure to follow this protocol in using tri-state components yields non-fatal warnings, but the offending assertion is ignored, hopefully forcing the designer to fix the timing that caused this to happen.

The event queue allows periodic events to be treated simply. In addition to queue entries for changing signal values, an additional type of entry is provided. This added type of entry specifies procedures to be called at the appropriate future time, and these procedures generate the queue entries for the next period as well as the next "wake-up" call for the procedure itself. This is the mechanism for providing periodic elements such as clocks, as well as the basis for providing moving objects on the display.

Event driven simulation entails a large amount of queue activity. For this reason, the implementation of the queue is extremely important. The queue used here is implemented as a heap, with the next event at the top of the heap. New entries are made at the bottom of the heap and percolated up to their appropriate place in the heap. This implementation gives Order (log n), where there are n items in the queue, performance.

Some performance enhancement can be realized at the gate level. For example, when evaluating an And gate, the discovery of a zero input obviates examination of any remaining inputs. Also, components represented functionally, such as registers, can have their evaluation abbreviated if the load signal is at the refractory level, when activation has occurred because of changes in the input data lines.

The time-model for devices generally has two times associated with it. The fastest time any device of the type might respond, and the slowest. The time between these limits is taken to be indeterminate whenever a logical change is taking place. The simulation is done conservatively in that any indeterminates which arise are propagated through devices using their more-rapid response time, while determinate values are propagated using their slower response time. This conservatism prohibits disregard for the outcome of races generally. This feature makes simulation somewhat different than construction with physical components. For example, a physical network would go to some definite state as the outcome of a race, and the designer might not care which of several possible states that was; simulation takes the point of view that the state is indeterminate.

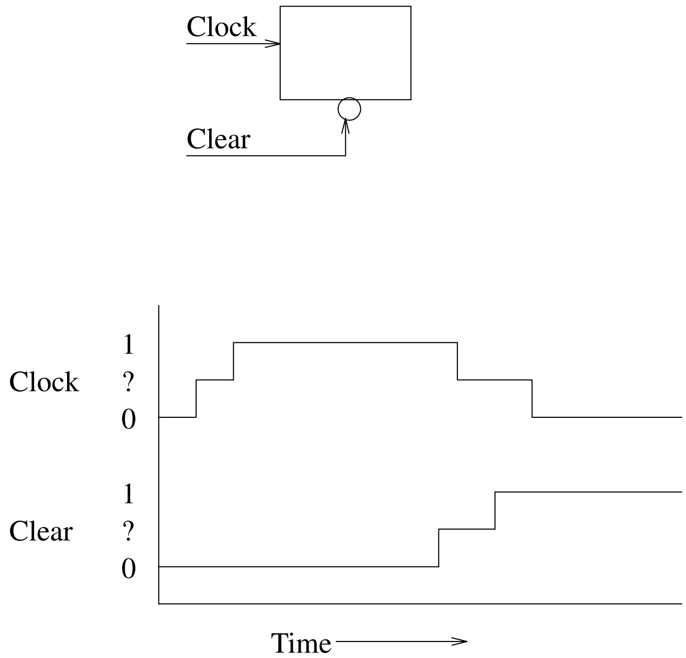
During simulation the network may be logically stimulated or the display may be controlled. This is generally accomplished using single-key actions. Logic stimulations include changing switch values and activating pulsers. Display control includes: zooming-in/out or panning, changing the level of module-hierarchy display, providing/inhibiting component and signal labeling, changing display colors, and changing window size, shape and placement. Other interactions include: resetting, slowing, speeding-up, halting or exiting the simulation, omitting/including the lines of component interconnection, and dumping diagnostic information.

2.8. Asynchronous Interface

The external signals controlling sequential machines are commonly of an asynchronous nature. That is, they are not, and can not reasonably be, synchronized to the clock for their changes. This commonly occurs when a manual input initializes or activates a transition of a sequential machine or when two or more independently clocked devices are in communication.

As a first example, consider a sequential machine that must be started or re-started in a particular state. Suppose there is an external switch that controls this (re-)initialization. The sequential machine most likely is made of flip-flops with asynchronous preset and clear inputs. Unfortunately, this does not mean that these inputs may be connected to asynchronous signals and they will automatically interact properly with the clocking signal.

Consider the figure below showing a clear signal, assumed active-low, occurring at the time of the clock pulse that is to process the normal synchronous inputs.



Clock-versus-Clear Timing-Conflict

The action of this machine is indeterminate because of the race conditions among the clearing action, the new excitations based on the cleared state, and the flip-flop's processing of the Clear versus its normal transition action based on the Clock. Of course, even if this race is critical it may not be disastrous if all possible outcomes are acceptable states as far as subsequent processing is concerned.

Conservative simulation, passing uncertainties rapidly and certainties slowly, prohibits disregard for the outcome of races generally. While a physical realization would go to some definite state, and the designer might not care which of several possible states that was, simulation takes the point of view that the state is indeterminate.

For a second example, consider a counter that is to count in either of two sequences, and an external switch that is to select which sequence is to be used at any given time. Here, there is an internal clock that controls when the transitions between count values are to occur, and an external switch that is to be controlled manually. Supposedly, the switch may be pushed at any time. How are the switch values to be included in the sequential machine's excitations?

There are at least two possibilities for merging external asynchronous signals and the internal clock. One is to stop the clock, so no input processing is attempted, while the asynchronous input is being changed. The second, is to create a version of the switch signal that is properly synchronized to the internal clock.

The simulator has three components available to help with this interfacing:

> The Gate element allows metering out an integral number of clock pulses using an asynchronous gating signal. The gating signal may occur at any time, even within a clock pulse, but the output will have the clock pulse continuing to completion.
>
> The Sync element synchronizes an asynchronous input to a clock, allowing the output to change to the value of the asynchronous input, but the change only appears at the output in synchronism with the clock.
>
> The Power-on element provides a signal that will make a single transition during its lifetime with a settable time for the transition. The time of transition would be set to occur in proper coordination with the clock. This component is a substitute for an R-C circuit commonly used for initialization. The Power-on differs markedly from an R-C circuit in that the precise settability of the Power-on transition time makes the Power-on component essentially a synchronous component. In practice,

the times of R-C networks switching between logical states is not closely controlled, plus or minus 20 percent being common.

When going from a simulated network to a hardware prototype, the three above elements will need careful translation. Sometimes, these translations will precipitate a significant redesign.

2.9. Network Construction

Although simple networks are constructed using an editor and a primitive gate collection, more complicated networks can only be created and managed effectively using higher-level schemes. Collections of gates implementing higher level functions occur frequently in digital design. Adders are an example of functional organization at the first level above the gate level.

While all networks could be constructed in a primitive manner using And's, Or's and Not's or just with Nand's or Nor's alone, it is essential, especially for large-scale design, that sub-networks be modeled and be available for inclusion at the higher levels of organization. Other components at the first level of integration include: flip-flops, multiplexers, decoders, latches and counters. At the next higher level there are register files and arithmetic logic components. One more level higher there are processors, memory modules and communications networks. And at the highest level there are networks of computer systems.

A means of originating sub-networks as macro definitions that are later expanded in their several instances of use is important, so we consider macro processing both generally and specifically in the case of digital design.

Macros are similar to procedures or functions in that they are defined initially, and then used as need be, later, by reference. Although procedures generally produce some desired side-effect and functions return values during computation, the role of macros is to ease the writing of programs or to aid in composing text streams using previously extracted definitions. Macros also allow individual expansions to vary slightly based on parameter substitution.

Macro processing is used both in fixed format and in stream contexts. In fixed format, such as assembly language programming, macro names are defined, and if a later operation field should reference that name, the expansion is provided in-line using the particular actual parameters supplied. Stream macro, or general purpose macro processing does not rely on fixed format input, and macro names may generally occur anywhere in the input text. These stream macro processors can be further broken down into two types. The first is always active, examining the input text for the occurrence of any previously defined name. On the recognition of such a name requiring expansion, the expanded item can be pushed back onto the head end of the input text stream, so that it can be rescanned for the possibility of further levels of substitutions in the cases of hierarchically arranged definitions. The second type can be considered to be relatively dormant, usually just copying the input text stream to the output stream without modification. However, in this type of macro processor certain input stream characteristics are reserved to awaken the macro processor, signaling that a definition is about to be made or that a macro expansion of what immediately follows is required. This second form is somewhat easier to implement, and the demarcation of the actual parameters is more easily accomplished.

The next section discusses the simulator's built-in methods of dealing with the hierarchical organization of systems.

Beyond the scope of these notes are the possibilities of using a general purpose macro processor, such as m4, or a preprocessor, such as the C-language preprocessor, as a means of organizing netlists at the highest level, and generating sim-netlist output.

2.10. Define/Use Built-In Facility

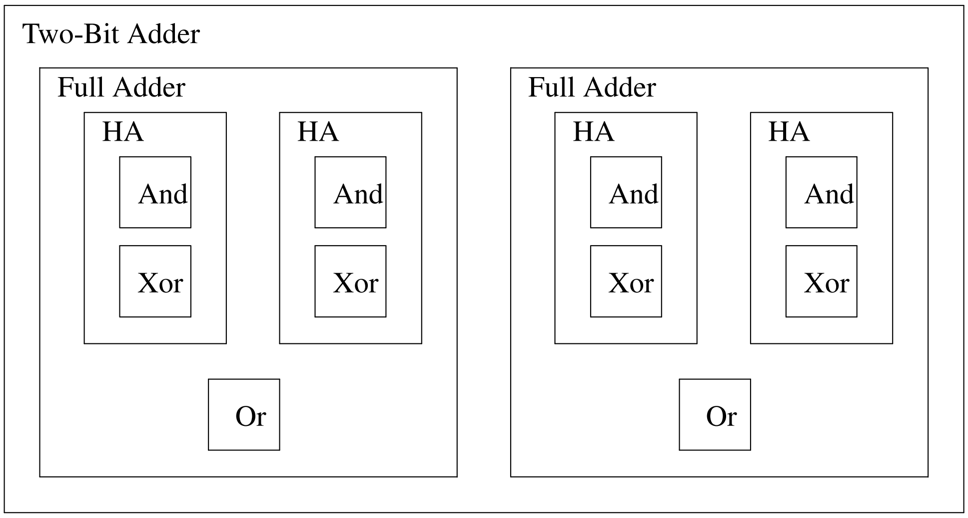
Consider the following hierarchy of definitions of networks, from half-adders through a two-bit adder.

```
Define Two_Bit_Adder a[1..0] b[1..0] Ci | Co t[1..0];
  Full_Adder H3 a[1] b[1] Ca | Co t[1];
  Full_Adder H4 a[0] b[0] Ci | Ca t[0];
 Endef ;

Define Full_Adder ix iy iz | Carry Sum;
  Half_Adder 3m ix iy | C1 S1;
  Half_Adder 3o S1 iz | C2 Sum;
  Or         4n C1 C2 Carry;
 Endef ;

Define Half_Adder  x y | C S;
  And 6s ONE x y C;
  Xor 7s x y S;
 Endef ;
```

This hierarchy of components is shown in the following.



Hierarchical Component Organization

If this two-bit adder is used in a Switch/Probe test fixture as:

```
Switch 1a x[1] ZERO; Switch 1a x[0] ZERO;
Switch 2a y[1] ZERO; Switch 2a y[0] ZERO;
Switch 3a Cin  ZERO;
Two_Bit_Adder 1b-3d x[1..0] y[1..0] Cin | Cout S[1..0];
Probe 1e Cout; Probe 2e S[1]; Probe 3e S[0];
```

then the following expansion results (the expansion shows the necessary Aliasing of interface variables, and the localization of internal variables and coordinate systems):

```
Alias x[1] x[0] y[1] y[0] Cin Cout S[1] S[0] |
  ZZ0a[1] ZZ0a[0] ZZ0b[1] ZZ0b[0] ZZ0Ci ZZ0Co ZZ0t[1] ZZ0t[0]  ;
Module  "Two_Bit_Adder"  1b-3d ZZ0a[1] ZZ0a[0] ZZ0b[1] ZZ0b[0] ZZ0Ci |
  ZZ0Co ZZ0t[1] ZZ0t[0]  ;
Alias ZZ0a[1] ZZ0b[1] ZZ0Ca ZZ0Co ZZ0t[1] |
  ZZ1ix ZZ1iy ZZ1iz ZZ1Carry ZZ1Sum  ;
Module  "Full_Adder"  1b-3d.H3 ZZ1ix ZZ1iy ZZ1iz | ZZ1Carry ZZ1Sum  ;
Alias ZZ1ix ZZ1iy ZZ1C1 ZZ1S1 | ZZ2x ZZ2y ZZ2C ZZ2S  ;
Module  "Half_Adder"  1b-3d.H3.3m ZZ2x ZZ2y | ZZ2C ZZ2S  ;
 And 1b-3d.H3.3m.6s ONE ZZ2x ZZ2y ZZ2C  ;
 Xor 1b-3d.H3.3m.7s ZZ2x ZZ2y ZZ2S  ;
 End ; /* Half_Adder */
Alias ZZ1S1 ZZ1iz ZZ1C2 ZZ1Sum | ZZ3x ZZ3y ZZ3C ZZ3S  ;
Module  "Half_Adder"  1b-3d.H3.3o ZZ3x ZZ3y | ZZ3C ZZ3S  ;
 And 1b-3d.H3.3o.6s ONE ZZ3x ZZ3y ZZ3C  ;
 Xor 1b-3d.H3.3o.7s ZZ3x ZZ3y ZZ3S  ;
 End ; /* Half_Adder */
 Or 1b-3d.H3.4n ZZ1C1 ZZ1C2 ZZ1Carry  ;
 End ; /* Full_Adder */
Alias ZZ0a[0] ZZ0b[0] ZZ0Ci ZZ0Ca ZZ0t[0] |
  ZZ4ix ZZ4iy ZZ4iz ZZ4Carry ZZ4Sum  ;
Module  "Full_Adder"  1b-3d.H4 ZZ4ix ZZ4iy ZZ4iz | ZZ4Carry ZZ4Sum  ;
Alias ZZ4ix ZZ4iy ZZ4C1 ZZ4S1 | ZZ5x ZZ5y ZZ5C ZZ5S  ;
Module  "Half_Adder"  1b-3d.H4.3m ZZ5x ZZ5y | ZZ5C ZZ5S  ;
 And 1b-3d.H4.3m.6s ONE ZZ5x ZZ5y ZZ5C  ;
 Xor 1b-3d.H4.3m.7s ZZ5x ZZ5y ZZ5S  ;
 End ; /* Half_Adder */
Alias ZZ4S1 ZZ4iz ZZ4C2 ZZ4Sum | ZZ6x ZZ6y ZZ6C ZZ6S  ;
Module  "Half_Adder"  1b-3d.H4.3o ZZ6x ZZ6y | ZZ6C ZZ6S  ;
 And 1b-3d.H4.3o.6s ONE ZZ6x ZZ6y ZZ6C  ;
 Xor 1b-3d.H4.3o.7s ZZ6x ZZ6y ZZ6S  ;
 End ; /* Half_Adder */
 Or 1b-3d.H4.4n ZZ4C1 ZZ4C2 ZZ4Carry  ;
 End ; /* Full_Adder */
 End ; /* Two_Bit_Adder */
```

2.11. Program Generation of Netlists

Networks with a significant amount of regularity can be generated indirectly, as the output of an executing program, instead of being entered in detail using an editor.

Consider the following example written in C:

```
#define rowlim 10
#define colim 20

main()
{char rowchst='a', colchst='A', rowch, colch;
 int row, col, node=0;
 printf("Switch %c%c 0 ZERO;\\n", rowchst, colchst);
 for ( row = 0; row < rowlim; row++ )
  for ( col = 0; col < colim; col++ )
   {printf("Not %c%c %d %d;\\n",
          rowchst+row, colchst+col, node, node+1);
    node++;
    }
 printf("Probe %c%c %d;\\n",
         rowchst+rowlim-1, colchst+colim-1, node);
 }
```

This program generates a cascade of over 200 components, and can be extended to arbitrarily larger networks by changing only the two initial preprocessor declarations.

If a network already exists in netlist form, a C program can often be made from it using the following "awk" program:

```
BEGIN { print "main()   {" }
  { print "printf(\\"%s\\n\\"" $0 ");"}
END   { print "}" }
```

If there are quoted strings in the original netlist, the C program generated will have to be edited because there will be nested strings.

2.12. Windows

Sim is currently supported on Sun workstations using the OpenLook environment, and on PC's using the DOS environment.

Within a a shell window give the sim command

```
csh> sim netlist
```

Sometimes it may be desirable to have more that one simulation active at a time using the separate-window capabilities of suntools. In this case sim should be started with

```
csh> sim netlist  &
```

The & begins a separate autonomous process for the simulation. In this case the original shell is still active for further commands including any additional sim initiations.

All the OpenLook windows features are available, including: opening, closing, moving, resizing, hiding, exposing, redisplaying and quitting. These features are selected using the mouse.

2.13. Command Line Arguments

Sim usage requires the command line entry:

```
sim [-e] [-f flags] [-l libname] [-v views] [-W ol_window_spec] [-g pc_graphics_spec] filename(s)
```

The flags are optional; they can be given in any order but must precede any filename(s). The -e flag suppresses the echoing of the input file(s) being read, the -f flag requires a supplemental hexadecimal value to set the desired flags, the -v flag provides multiple viewing windows, and the -l flag designates a supplemental library file to be searched for Define'd types referenced in the netlist. The default library name is sim.lib.

The list of filenames is also optional, and, if absent, input will be taken from the standard input. The standard input mechanism allows piping sim input from other programs. Standard input can also follow anytime after the first filename by using the specifier "-".

An intermediate file, sim.tmp, will be created in the local directory and used for expansion of any instances of Define's uses.

Any window parameters specified on the command line use the following forms:

Flag	Arguments	Attributes
-Wb	red green blue	background color
-Wf	red green blue	foreground color
-Wp	x y	window position (pixels)
-Ws	x y	window size (pixels)

These parameters may be omitted and the default values will be used. The defaults will be satisfactory although the default blank-on-white on color monitors can be improved upon by giving harmonious colors.

The red, green and blue color components are given as unsigned, 8-bit intensities. The screen rendition will be the additive result of the primaries. The following table lists some common combinations (there are 16 million colors possible):

COLOR COMBINATIONS		
(r)[BLACK]=0	(g)[BLACK]=0	(b)[BLACK]=0
(r)[RED]=255	(g)[RED]=0	(b)[RED]=0
(r)[ORANGE]=192	(g)[ORANGE]=64	(b)[ORANGE]=0
(r)[YELLOW]=128	(g)[YELLOW]=128	(b)[YELLOW]=0
(r)[GREEN]=0	(g)[GREEN]=255	(b)[GREEN]=0
(r)[CYAN]=0	(g)[CYAN]=128	(b)[CYAN]=128
(r)[BLUE]=0	(g)[BLUE]=0	(b)[BLUE]=255
(r)[MAGENTA]=128	(g)[MAGENTA]=0	(b)[MAGENTA]=128
(r)[WHITE]=255	(g)[WHITE]=255	(b)[WHITE]=255

The rule for indigo is B > R & SQRT(B**2+R**2) < .5 where 0.0<=B|R<=1.0). Trying R=.25 and B=.3.

(r)[INDIGO] = 64;   (g)[INDIGO] = 0;   (b)[INDIGO] = 76;

The rule for violet is R > B & SQRT(B**2+R**2) > .5 where 0.0<=B|R<=1.0). Trying R=.5 and B=.7.

(r)[VIOLET] = 128;   (g)[VIOLET] = 0;   (b)[VIOLET] = 178;

The -g flag specifies the graphics adapter if a PC is being used. If no adapter is specified, sim will search for the one of greatest resolution.

PC Graphics			
Flag String	Adapter	Resolution	Colors
MRES256COLOR	VGA	320x200	256 color
VRES16COLOR	VGA	640x480	16 color
VRES2COLOR	VGA	640x480	BW
ERESCOLOR	EGA	640x350	4or16 color
ERESNOCOLOR	EGA	640x350	BW
HRES16COLOR	EGA	640x200	16 color
MRES16COLOR	EGA	320x200	16 color
HRESBW	CGA	640x200	BW
MRESNOCOLOR	CGA	320x200	4 grey
MRES4COLOR	CGA	320x200	4 color

2.14. Experimental Techniques

It is not practical to give exhaustive descriptions of individual component's characteristics. For the most part, the components are closely related to standard component types, and further supporting information can be found in your digital design textbook.

A valuable approach is to experiment with individual components separately if any of their characteristics is in doubt. Usually this merely amounts to connecting the component to Switch inputs and Probe outputs, and putting the component through a revealing set of excitation patterns.

In more complicated designs it is common to experience timing difficulties. To investigate these problems: the "throbs" may be turned on, Probes may be added, the simulation may be slowed, the timed signals may be displayed on a() Scope(s), and/or Clocks may temporarily be replaced with Switches or Pulsers so a more controlled investigation can be conducted. Time-probe's are powerful investigative elements, but they're hard to use; these are a last resort that should rarely be needed.

2.15. Function

As networks are composed in an hierarchical manner, it may neither be feasible nor desirable to define networks down to the primitive component level. Instead, it may be desirable to express some components functionally, i.e. without any further decomposition. Functional specification can be useful in the following situations:

1. The gate or primitive-level components are not adequate models, perhaps in a multiple-input, asynchronous network.
2. The internal specifications of a module are still in development, but it is desirable to proceed using a functional form in the meantime.
3. The primitive subnetwork already exists and it is desirable to develop a functional model for descriptive purposes.
4. The functional-level model will enhance the speed of simulation by avoiding the evaluation of numerous primitive elements.
5. It may be desirable to access some special hardware or operating-system characteristics of the simulation machine.

To accomplish this functional specification, the characteristics of a component can be modeled in a programming language--most likely C.

The netlist specification of a Function use is

    Function "file-name" <screen-cell> <inputs> | <outputs> ;

where the "file-name" string is the name of the file in the current directory or pathname of a file that has the executable object code for the functional specification. Variable numbers of inputs and outputs are allowed. There is currently a limitation of 80 outputs maximum, but no limitation on the number of inputs allowed.

The same file-name can be referenced multiple times in the netlist, and only one copy of the code from that file will be loaded, and each netlist reference will be to that copy. Other Function's can reference other object-code files. There is no provision for accessing more than one entry point from any object file. The entry point is the first byte of the text segment in the file.

This Function facility is provided in the late-binding mode, i.e. the linkage editor cannot be employed to link anything between the "file-name" file and the simulator code or within the object code file. To do otherwise would require too much time linking between simulator invocations, and encourage too much file-space usage with incrementally linked object files.

Functions within the object module can be used if they are totally linked. Also, any data used will have to be within the text segment. Since the object module will be placed in the heap, it will be possible to write any data in the text segment.

The code in the "file-name" file should be constructed so there is only a text segment. Other segments can be there but they will be ignored. The component corresponding to Function will be evaluated whenever any of its inputs changes.

The following is an example of the C-language source for a function that provides a variable-width inverting-buffer component.

```
#define ZERO        1
#define HI_Z        2
#define UNKNOWN     3
#define ONE         5
one ( state, nr_inputs, input_indices, global_signals, nr_outputs, output_vector)
 char * state, * global_signals, * output_vector;
 short nr_inputs, nr_outputs;
 short *input_indices;
 {
  int i;
  for ( i = 0; i < nr_inputs; i++ )   {
    switch ( global_signals [ input_indices [ i ] ] )   {
      case ZERO:    output_vector[i] = ONE; break;
      case UNKNOWN: output_vector[i] = UNKNOWN; break;
      case HI_Z:    output_vector[i] = UNKNOWN; break;
      case ONE:     output_vector[i] = ZERO; break;
      default:      output_vector[i] = UNKNOWN;
     }
  }
 return ( 1 );
 }
```

The name used for the function, "one" above, is immaterial. The interface to the function is always the same and must adhere to the sizes and indirections indicated.

Do not use the direct formal-parameter typing that ANSI C allows since the calling sequence (in the simulator) uses the default, actual-parameter promotion rules.

There is no control of the timing of these functionally-modeled components. There is a fixed-delay model of the times at which output changes will become effective. Also, the outputs will be made "continuous" regardless of what the Function code tries to do. An output of ZERO followed next by an output of ONE will have an intermediate UNKNOWN value inserted.

The state value is -1 for the initial call of the function and for the first call after each reset (R) of the simulation. This called function is in control of the state for other times.

A non-zero value returned by the function indicates that the outputs have changed. A zero value returned indicates that they have not changed. Not only can the zero return value speed processing, but it also keeps the function from otherwise having to "know" or "remember" what the previous outputs were.

The following section of code illustrates how local space can be allocated for use of a Function component when using a Sun-4 workstation. The scheme is a little awkward since there is no PC-relative addressing mode in the SPARC architecture, and relocatable code is required.

```
  .text
  .proc 1
_where:    ! Declare in C as:  char * where();
  save %sp,-112,%sp
_L_X:   call _d_ummy,0
  mov %o0,%o0
  add %o0,_S_pace-_L_X,%i0
  ret
  restore
  .proc 1
_d_ummy:
  save %sp,-112,%sp
  mov %i7,%i0
  ret
  restore
_S_pace:   ! Non-volatile area follows
  .byte 9
  .byte 5
  .byte 6
  .byte 3
  .half 31
  .word 23
  .skip 1016
```

A More Compact Solution --

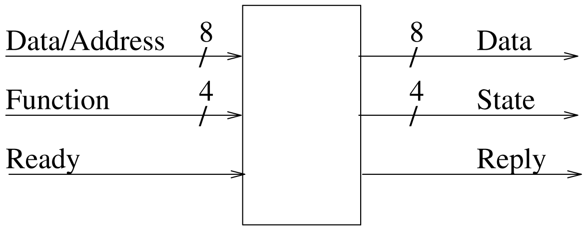
```
.text
        .align 4
.global _where  ! Declare in C as:  char * where();
        .proc 1
_where:
        save %sp,-112,%sp
1: call 2f
        mov %o0,%o0
2: add %o7,_S_pace-1b,%i0
        ret
        restore
        .align 4
.global _S_pace
_S_pace:        ! Non-volatile storage area:
        .byte 9
        .byte 5
        .byte 6
        .byte 3
        .half 31
        .word 23
        .skip 1016
```

2.16. Disk-unit

The Disk-unit is an asynchronous device that provides 262,144 bytes of disk space. The Disk-unit allows random access among the tracks and sectors provided, and sequential access within sectors.

Disk Unit	
Tracks	64
Sectors	32
Bytes/Sector	128

The Disk-unit has 13 inputs and 13 outputs, and accommodates up to five, tape-like, external files.



Disk Unit Interface

Example usage:

Disk-unit "file_0,file_1" 2b In[7-0] Fun[3-0] Ready Out[7-0] State[3-0] Reply;

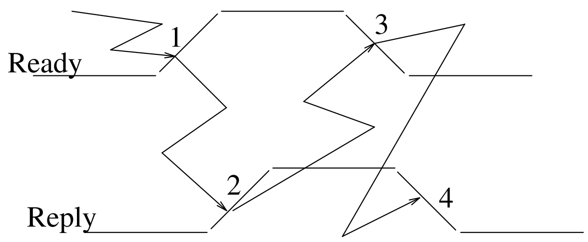
The Function inputs specify an operation or record a selection. The following table lists the functions and their corresponding codes.

Disk-unit Functions		
Function	Code	Comments
Spares	0-4	
Track Seek	5	Invalidates Sector Selection
Sector Seek	6	Invalidates Read/Write Selection
Write Select	7	Invalidates Previous Contents of Sector
Read Select	8	Circularly in Sector
Move Byte	9	Read or Write as Selected
Select External File	10	
External to Sector	11	Data_Out = # of Bytes-Read
Sector to External	12	
Rewind External	13	Include This!
Dump Disk to External	14	Full Disk Dump
Spare	15	

A selection persists until it is displaced by a different selection. For example, if external file zero is selected, that selection remains until a different external file is selected, even though there might be intervening actions not involving external files.

If an illegal operation is attempted, such as Move Byte without previous Track, Sector and Read/Write selections, the Disk-unit will go to an error state. The error state will be reported on the four State output lines, and their coding will correspond to the Function-in-error attempted.

The protocol for all functions and actions is shown in the following:



Ready/Reply Sequencing

where the numbered transitions indicate:

1. Data/Function Inputs are Ready.
2. Disk has Completed its Action. The initiator needs to wait for this transition before lowering Ready or perturbing the Data/Address or Function values.
3. Initiator Acknowledges Disk Reply.
4. Disk Returns to Wait for Next Request. The initiator needs to wait for this Reply transition before raising Ready again.

In addition to the interface presented at the simulation interconnection-level, the Disk-unit allows references to external, tape-like files. Information can be transferred between these external files and the Disk-unit. The external files can be created and post-mortem'ed in the UNIX environment.

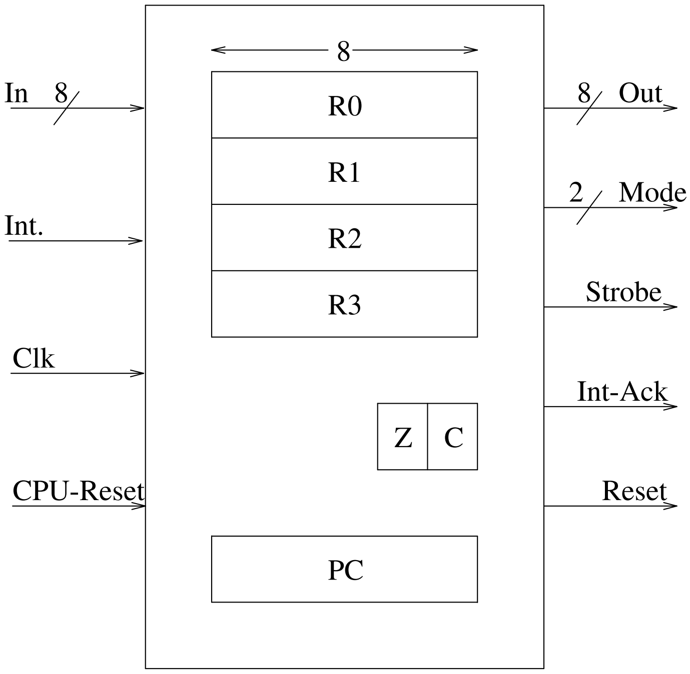
This disk is volatile and is empty at the start of a simulation. The serial files are provided to allow the disk to be initialized. In a computer system, File_0 could contain the operating system, and would be loaded by the Rom start-up program, to form the boot record on the disk. File_1 could be a user-program. File_2 & 3 could be user-program or data-I/O. File_4 could be used as a disk dump.

Some disk-like characteristics are missing from this implementation. These include:

1. The Disk-unit cannot initiate the data transfer. In reality, this is where the critical time originates--a disk byte needs to be filled NOW!
2. There is no provision for an interrupt generation. Usually, the operations that take a long time, track-seek for example, are launched and cause an interrupt when they complete. In this unit, these long times are not modeled.
3. There is no hardware reset connection to the unit. This would typically be obtained as the system master reset and would fix the initial state of the Disk-unit.

2.17. The 4x8 Microprocessor

The 4x8 microprocessor has four, eight-bit registers, and was derived from the Motorola M680x0. While the 4x8 is much smaller and simpler, the M680x0 mnemonic codes and address designators as used by the Sun Assembler for the M680x0 have been used. Since the 4x8 uses mostly the same mnemonics and operations, as well as a subset of the M680x0 addressing modes, further helpful information can be found in the Motorola Reference Manual.



4x8 Microprocessor

The microprocessor has two eight-bit buses, one input and one output. There is an eight-bit address space, and the supplemental RAM, ROM and I/O must be mapped into this space.

The four registers are completely equivalent in all their operations.

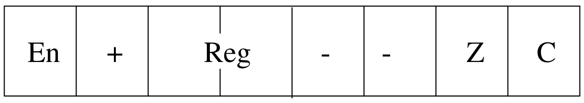
An automatic stack operation is only available for handling interrupts, other stack operations must be implemented in fine detail. That is, by separate instructions for adjusting the stack pointer (a chosen register) and storing and retrieving using the register-indirect addressing mode.

There is a conditions code register with a zero (Z) and a carry (C) flag. These conditions are both set as a result of all arithmetic operations and the increment and decrement instructions. The carry flag can be set or cleared directly by instructions and can participate in shifting operations. The conditional jump instructions test these flags.

A four-bit internal condition is maintained to record whether the interrupt system is enabled, which register is to be used as the interrupt stack pointer and whether this register is to be used in the pre-decrement or post-increment mode when pushing elements onto the stack.

When the interrupt system is enabled a register is designated as a stack pointer for the stack to be used to capture information about the state of the main program that will be interrupted. This state is restored by executing the return from interrupt (rti) instruction.

The internal condition plus the condition code form a Program Status Word (PSW).



PSW

The PSW can be displayed, along with other registers, by the simulator.

If the interrupt system is enabled, a low signal on the interrupt line will cause the program counter to be pushed onto the stack and the PSW will be pushed after it. An interrupt disables the interrupt during service. The return from interrupt (rti) will restore the interrupt enable when the PSW is retrieved from the stack. The interrupt service routine starting address is fetched from location zero.

An interrupt-acknowledge signal is supplied by the microprocessor. This signal is normally high, and goes low when the interrupt is recognized (at the beginning of a Fetch phase). Interrupts must remain low until this acknowledging signal responds. The external system should have removed the activating interrupt signal before the rti completes execution or another interrupt will immediately occur. The interrupt-acknowledge signal goes back high at the beginning of the execution of the rti instruction.

The high-going edge of the reset signal initializes the micro by setting the program counter to one, and setting the interrupt system to disabled. Location zero contains the address of the interrupt servicing routine.

There are four addressing modes associated with 4x8 operands:

1. Register-Direct--The designated register is the source and/or destination.
2. Register-Indirect--The location pointed to by the register is the source or destination.
3. Immediate--The byte immediately following the instruction is the data source.
4. Absolute--The byte immediately following the instruction is an address. This address can be the destination of a transfer of control, or can be the source (destination) of a data transfer to (from) a register.

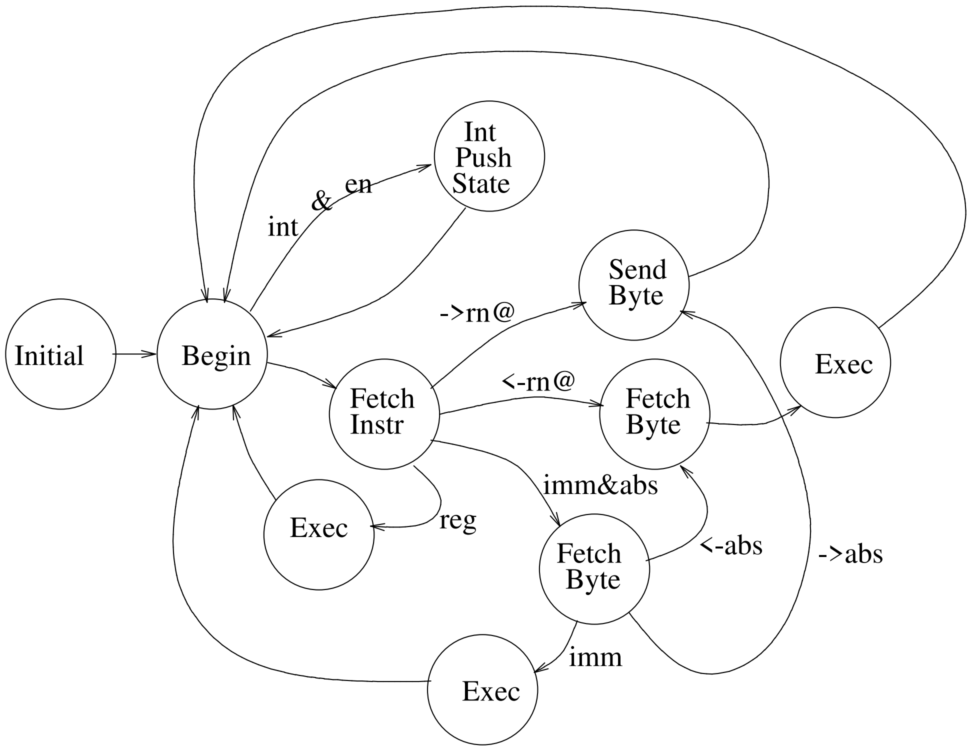
The jsr instructions use a register designation for capturing the current program counter value, and an absolute address for the new program control point. The rts similarly designates a register for the restoration of the program counter. This usage is said to designate the register as a "link" register.

The above addressing features for jsr, rts, en, and rti are unusual, but they are used in this machine to keep the addressing modes as simple as possible except when the interrupt system is being used. Further, the elaboration of these instructions over the register set, as opposed to having special-purpose registers such as stack pointers, is used to keep the registers equivalent in all their operations.

The microprocessor furnishes two mode lines and a strobe signal for interfacing to external components. The mode signals are coded as: (M1,M0): (0,0)--Read; (0,1)--Address For Write; (1,0)--Data For Write. When the Strobe goes low, the Mode signals will indicate the desired operation.

Hex codes with no entry in the table represent illegal instructions and they should not be used.

The following state diagram for the 4x8 control does not

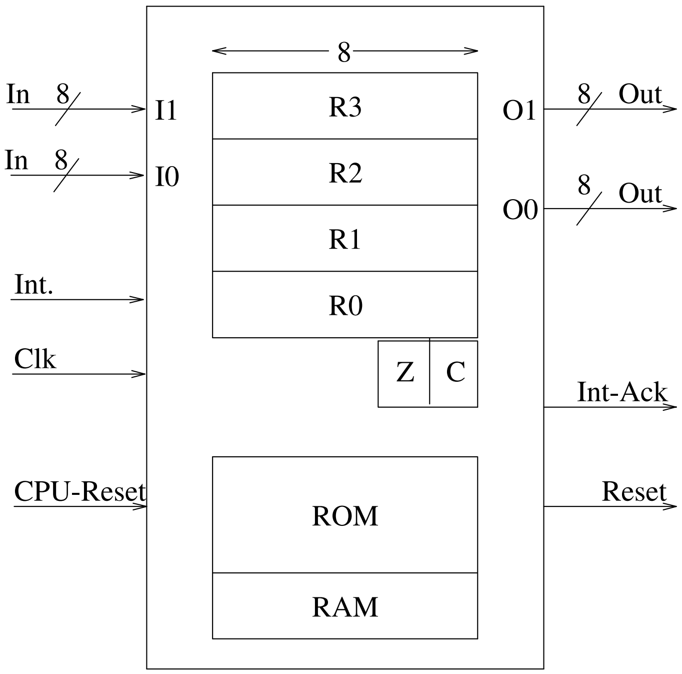


4x8 State Diagram

show the fine-detail states associated with generating the Mode and Strobe outputs. However, it does give a high-level overview suitable as a starting point for microcode development.

2.18. The 4x8 Microcomputer

The 4x8 microcomputer is a combination of a 4x8 microprocessor and a memory I/O unit.



4x8 Microcomputer

The microcomputer has two eight-bit input ports, I0 and I1, at memory-mapped positions 252 and 253, respectively. There are also two eight-bit output ports, O0 and O1, at memory-mapped positions 254 and 255, respectively. Ram memory is located at addresses 224 through 251, and Rom is at locations 0 through 223.

This microcomputer version of the 4x8 is intended to provide high-speed functional simulation. This is possible because the asynchronous communication between the microprocessor and the memory-I/O unit can be eliminated when the units are combined.

The microcomputer and microprocessor have some notable differences:

1. The microcomputer executes an instruction on each clock cycle.
2. The microcomputer has fixed memory mapping and a fixed number of input/output ports.
3. The microcomputer initializes its Ram from the constant list provided in the instantiation.
4. The internal state showing on the register display will differ between the microcomputer and microprocessor. The microprocessor has many more internal states.

The remaining characteristics should be the same as those given in the 4x8 microprocessor description.

Hexadecimal Codes for 4x8 Instructions																
	LSB															
MSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	clr r0	clr r1	clr r2	clr r3	not r0	not r1	not r2	not r3	inc r0	inc r1	inc r2	inc r3	dec r0	dec r1	dec r2	dec r3
1	rol r0	rol r1	rol r2	rol r3	rocl r0	rocl r1	rocl r2	rocl r3	ror r0	ror r1	ror r2	ror r3	rocr r0	rocr r1	rocr r2	rocr r3
2	and r0,r0	and r0,r1	and r0,r2	and r0,r3	and r1,r0	and r1,r1	and r1,r2	and r1,r3	and r2,r0	and r2,r1	and r2,r2	and r2,r3	and r3,r0	and r3,r1	and r3,r2	and r3,r3
3	or r0,r0	or r0,r1	or r0,r2	or r0,r3	or r1,r0	or r1,r1	or r1,r2	or r1,r3	or r2,r0	or r2,r1	or r2,r2	or r2,r3	or r3,r0	or r3,r1	or r3,r2	or r3,r3
4	eor r0,r0	eor r0,r1	eor r0,r2	eor r0,r3	eor r1,r0	eor r1,r1	eor r1,r2	eor r1,r3	eor r2,r0	eor r2,r1	eor r2,r2	eor r2,r3	eor r3,r0	eor r3,r1	eor r3,r2	eor r3,r3
5	add r0,r0	add r0,r1	add r0,r2	add r0,r3	add r1,r0	add r1,r1	add r1,r2	add r1,r3	add r2,r0	add r2,r1	add r2,r2	add r2,r3	add r3,r0	add r3,r1	add r3,r2	add r3,r3
6	addc r0,r0	addc r0,r1	addc r0,r2	addc r0,r3	addc r1,r0	addc r1,r1	addc r1,r2	addc r1,r3	addc r2,r0	addc r2,r1	addc r2,r2	addc r2,r3	addc r3,r0	addc r3,r1	addc r3,r2	addc r3,r3
7	sub r0,r0	sub r0,r1	sub r0,r2	sub r0,r3	sub r1,r0	sub r1,r1	sub r1,r2	sub r1,r3	sub r2,r0	sub r2,r1	sub r2,r2	sub r2,r3	sub r3,r0	sub r3,r1	sub r3,r2	sub r3,r3
8	subc r0,r0	subc r0,r1	subc r0,r2	subc r0,r3	subc r1,r0	subc r1,r1	subc r1,r2	subc r1,r3	subc r2,r0	subc r2,r1	subc r2,r2	subc r2,r3	subc r3,r0	subc r3,r1	subc r3,r2	subc r3,r3
9	mov r0,r0	mov r0,r1	mov r0,r2	mov r0,r3	mov r1,r0	mov r1,r1	mov r1,r2	mov r1,r3	mov r2,r0	mov r2,r1	mov r2,r2	mov r2,r3	mov r3,r0	mov r3,r1	mov r3,r2	mov r3,r3
A	mov cc,r0	mov cc,r1	mov cc,r2	mov cc,r3	mov r0,cc	mov r1,cc	mov r2,cc	mov r3,cc	reset	clr c	set c	-	-	-	-	-
B	mov r0@,r0	mov r0@,r1	mov r0@,r2	mov r0@,r3	mov r1@,r0	mov r1@,r1	mov r1@,r2	mov r1@,r3	mov r2@,r0	mov r2@,r1	mov r2@,r2	mov r2@,r3	mov r3@,r0	mov r3@,r1	mov r3@,r2	mov r3@,r3
C	mov r0,r0@	mov r0,r1@	mov r0,r2@	mov r0,r3@	mov r1,r0@	mov r1,r1@	mov r1,r2@	mov r1,r3@	mov r2,r0@	mov r2,r1@	mov r2,r2@	mov r2,r3@	mov r3,r0@	mov r3,r1@	mov r3,r2@	mov r3,r3@
D	mov #imm,r0	mov #imm,r1	mov #imm,r2	mov #imm,r3	-	-	-	jmp abs	jcc abs	jcs abs	jne abs	jeq abs	jsr r0,abs	jsr r1,abs	jsr r2,abs	jsr r3,abs
E	mov abs,r0	mov abs,r1	mov abs,r2	mov abs,r3	mov r0,abs	mov r1,abs	mov r2,abs	mov r3,abs	-	-	-	-	-	-	-	-
F	en r0@-	en r1@-	en r2@-	en r3@-	en r0@+	en r1@+	en r2@+	en r3@+	dis	rti	stop	nop	rts r0	rts r1	rts r2	rts r3
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

2.19. Description of 4x8 Mnemonics

ADD Rx,Ry	Addition: Rx + Ry -> Ry. C and Z flags modified
ADDC Rx,Ry	Addition using Carry: Rx + Ry + C -> Ry. C and Z flags modified
AND Rx,Ry	Bitwise And: Rx & Ry -> Ry. C and Z flags modified.
CLR C	Clear the Carry flag.
CLR Rn	Clear register Rn, C and Z flags modified.
DEC Rn	Decrement register Rn. C and Z flags modified
DIS	Disable the interrupt system.
EN Rn@-	Enable the interrupt system and designate register Rn to be used as the stack pointer in a pre-decrement mode when pushing.

EN Rn@+	Enable the interrupt system and designate register Rn to be used as the stack pointer in a post-increment mode when pushing.
EOR Rx,Ry	Bitwise Exclusive-or: Rx ^ yx -> Ry. C and Z flags modified.
INC Rn	Increment register Rn. C and Z flags modified
JMP abs	Unconditional jump to the absolute address.
JCC abs	Carry clear conditional jump to the absolute address.
JCS abs	Carry set conditional jump to the absolute address.
JNE abs	Zero-flag clear conditional jump to the absolute address.
JEQ abs	Zero-flag set conditional jump to the absolute address.
JSR Rn,abs	Jump to subroutine at the absolute address, saving the program counter in register Rn.
MOV Rx,Ry	Move: Rx -> Ry
MOV CC,Rn	Move condition codes into Rn
MOV Rn,C	Move register into condition codes
MOV Rx@,Ry	Move the contents of the location pointed to by Rx into Ry.
MOV Rx,Ry@	Move Rx into the location pointed to by Ry.
MOV #imm,Rn	Move the immediate value into register Rn.
MOV abs,Rn	Move the contents of the absolute address into Rn.
MOV Rn,abs	Move Rn into the absolute address given.
NOP	No operation.
NOT Rn	Bitwise complement of register Rn
OR Rx,Ry	Bitwise Or: Rx | Ry -> Ry. C and Z flags modified.
RESET	Lower the microprocessor output line Reset for a little while.
ROL Rn	Rotate left register Rn
ROCL Rn	Rotate with carry left, register Rn. C flag modified
ROR Rn	Rotate right register Rn
ROCR Rm	Rotate with carry right, register Rn. C flag modified
RTI	Return from interrupt.
RTS Rn	Return from subroutine and restore the program counter from Rn.
SET C	Set the Carry flag.
STOP	Avoid this, it probably doesn't work anyway. It's relation to an enabled interrupt system is not decided yet.
SUB Rx,Ry	Subtraction: Ry - Rx -> Ry. C and Z flags modified
SUBC Rx,Ry	Subtraction using Carry: Ry - Rx - C -> Ry. C and Z flags modified

2.20. The 4x8 Assembler

The assembler accepts symbolic input and produces the corresponding hex codes for 4x8 programs. Command line arguments include -s that will dump the symbol table on stderr between passes, and -l that will produce the file asm.lst containing the original source code supplemented with the hex translations. The optional asm.lst file is produced during the second pass. The source code is echoed to stderr during the first pass. The second pass will not be attempted if there are errors during the first pass.

The standard output receives only the hexadecimal bytes of the translation. Thus, by redirecting the standard output, it is possible to obtain a hex codes listing suitable for merging into a netlist file as ROM contents.

The syntax for assembly language statements is:

{<label>:}{<operation>}{{#}<operand>{@}{,<operand>{@}}}{;<comment>}'\n'

where {...} are optional fields.

The assembler is generally case neutral, and upper and lower case can be mixed.

The operation field can be either a mnemonic operation code for the 4x8 or a pseudo operation. The pseudo operations include:

```
ORG <location>
DB <byte(s)_specification(s)>
BSS <integer>
```

Absolute addresses can be either integers or symbolic addresses. Symbolic addresses must be defined by their occurrence as the label for some statement. Symbolic addresses can be composed according to the regular expression:

```
[A-Za-z][A-Za-z0-9]*
```

If an operand field represents an immediate mode data byte, that field will allow an integer, a symbolic address or a character in single quotes. The immediate mode data byte must be preceded by a pounds sign (#) to show the mode.

The ORG statement allows the specification of the assembly-time location counter. It is initialized to zero. The assembly language program must proceed through the address space from location zero upward. When an ORG is given, the location designated must be beyond the current assembly-time location counter. Intervening bytes will be padded by the assembler in the output file.

The DB statement allows assembling single bytes as symbolic address values, integers or single quoted characters, or multiple bytes for quoted strings. A zero byte is assembled as the terminator for strings.
For example:

```
Here:  DB Here
       DB 127
       DB 'x'
       DB "ab cde f"
```

The BSS (block start symbol) statement would usually be labeled, and provides an integer specification of the number of bytes. The bytes are filled in, in the output file.

2.21. 4x8 Example

Assume we will have a 4x8 microprocessor that will provide an eight-bit output that will count in binary under program control. There will be two modes, count up, and count down. An interrupt signal will toggle between these two modes.

A suitable assembly language program would be:

```
          db    intserv      ;interrupt service address
Start:    clr   r1
          mov   r1, cntr     ;initialize cntr in RAM
          mov   #1, r2       ;set initial mode: 0--down; 1--up
          mov   #stack, r0   ;set the stack pointer
          en    r0@-         ;enable interrupt
count:    mov   cntr, r1     ;get value for update
          clr   r3           ;checking the mode
          add   r2,r3
          jeq   decrement
          inc   r1
          jmp   send
```

```
decrement:  dec   r1
send:       mov   r1, cntr      ;back to RAM
            mov   r1, out0      ;to the output port
            clr   r1            ;to check that RAM really works
            jmp   count

intserv:    mov   #1, r3        ;toggle mode
            sub   r2,r3         ;note that r3 is clobbered by
            mov   r3, r2        ; interrupt--maybe harmlessly here.
            rti

            org   224
cntr:       bss   1

            org   230
stack:

            org   252
out0:       bss   1
out1:       bss   2
            org   256
```

If this source file is saved under the filename "example", then the assembly is carried out by invoking the assembler as:

```
asm4  <example  >hexcodes
```

This will leave the assembled code on the "hexcodes" file for merging with the netlist.

A suitable netlist for this example is:

```
Switch 2a Int ONE;
Clock 2a Clk ONE 200 50 400;
Power-on 2a CPU-Reset ZERO 300;

Microprocessor 1b-2d
  Data_Out[7..0]  Int  Clk  CPU-Reset
  Data/Addr_In[7..0] Mode[1..0] Strobe Int-Ack Reset ;

Byte-probe 2g Out_Port_0[7-0];
Probe 3g Int-Ack;

Mem-IO 1e-3f
  In_Port_1[7-0] In_Port_0[7-0]
  Data/Addr_In[7-0]
  Mode[1-0] Strobe  ONE |
  Data_Out[7-0] Out_Port_1[7-0] Out_Port_0[7-0];
```

Some of the above signals are undefined, but except for the diagnostics they will trigger during the SIM input phase, they will be harmless. The definition file for the Mem-IO component is not shown above, but it would need to be included.

2.22. Command Line Flags

If the optional flag -f is used on the command line, a hexadecimal value follows the flag. The value is entered without the leading 0x and without the trailing L and is composed as the union of the desired bits from the following:

```
#define Time_Disp_flag          0x1L
                /* Turns on the time display. */
#define Spare_One               0x2L
                /* Not currently used. */
#define Interconnect_flag       0x4L
                /* Turns off the drawing of interconnections. */
#define No_Pause_Flag           0x8L
                /* Inhibits the Pause after the Netlist read. */
#define Throb_flag              0x10L
                /* Turns on a Probe in each component icon. */
#define Gate_Decal              0x20L
                /* Enables the labeling of primitive gates. */
#define Micro_Regs_Flag         0x40L
                /* Enables the display of the micro's registers */
#define Spare_Two               0x80L
                /* Not currently used. */
#define Mem_Part_flag           0x100L
                /* Gives the PC memory-segment assignments. */
#define Echo_flag               0x200L
                /* Enables keyboard-to-screen echoing. */
#define Function_Flag           0x400L
                /* Not currently used. */
#define Vector_flag             0x800L
                /* Dumps the coordinates of vectors drawn. */
#define Ext_Ref_flag            0x1000L
                /* Tells of externals (Define's) processing. */
#define CompIO_flag             0x2000L
                /* Provides commentary on the input. */
#define A_lists_flag            0x4000L
                /* Lists the activation sets. */
#define Audit_Tab_flag          0x8000L
                /* Enables auditing of the interconnections. */
#define Sub_Cell_flag           0x10000L
                /* Displays the graphic packing of screen cells. */
#define Manhattan_flag          0x20000L  /* Not currently active */
                /* Enables Manhattan-type interconnections. */
#define Scrn_Cell_flag          0x40000L
                /* Dump the Screen Structure -- world level. */
#define Sig_Q_flag              0x80000L
                /* Displays the length of the event queue. */
                                /* Currently, always on when Time is displayed */
#define M_Audit_flag            0x100000L
                /* Enables auditing of memory allocation. */
#define Symbol_flag             0x200000L  /* Not currently active */
                /* Shows symbol table operations. */
#define Node_label_flag         0x400000L  /* Not currently active */
                /* Disables the labeling of interconnections. */
#define Dynamic_alloc_flag      0x800000L
                /* Enables comments on heap actions. */
#define Stack_flag              0x1000000L
                /* Enables comments on stack allocation. */
#define Q_Warp                  0x2000000L  /* Not currently active */
                /* Provides diagnostics for Q-Warp. */
#define Hash_flag               0x4000000L  /* Not currently active */
```

```
                /* Dumps the hash table. */
#define  Dump_Q_Flag         0x8000000L
                /* Dumps the event queue as it is changed. */
#define DEFFLAGS   0x103L
```

2.23. Interactive Control

During simulation the network may be logically stimulated or the display may be controlled. The following single-key actions are provided:

Key	Action
Ctrl-a	Cycle the mode of arrow key actions among: ZOOM-IN, ZOOM-OUT and PAN.
Ctrl-b	Change the mode of Switch activations.
D	Toggle the auditing of dynamic (HEAP) allocation.
E	Toggle echo of key to screen.
G	Toggle primitive gate labeling. Redraw the screen, "r", to see the effects.
H	Toggle the "Halt" of the simulation.
I	Toggle the drawing of interconnections. Redraw the screen, "r", to see the effects.
K	Toggle the auditing of stack usage.
l	Toggle the the option of including node labels. Redraw the screen, "r", to see the effects.
m	Set the Module display hierarchy to the next higher level. Redraw the screen, "r", to see the effects.
n	Set the Module display hierarchy to the next lower level. Redraw the screen, "r", to see the effects.
R	Reset the simulation to time zero.
r	Redraw the network in the current window.
T	Toggle time display.
t	Toggle "throb" display. This gives a probe display within the icon of each component. This pane reports a component's output as soon as it is delivered to the queue, i.e. sooner than it can affect anything it's connected to.
u	Toggle the display of the microprocessor and microcomputer registers.
U	Moves the scope traces together.
V	Moves the scope traces apart.
w	Wipe the current window to the background color.
x	Exit SIM.
X	Toggle event-queue dumping.
y	Decrease the delay loop.
z	Increase the delay loop, i. e. slow down the screen presentation so you can watch the changes.
Arrow	Window modification depending on the mode set with ctrl-a. The cropping lines are shown when Zoom-in is in effect. A redraw, "r", command must be given to see the modified screen.
0..9:;<=>?	Change the state of the switch with the corresponding screen label. Mode set with ctrl-b.
+	Double the Zoom-in/out increment -- the increment used with the arrow keys.
-	Halve the Zoom-in/out increment.
a..h	Activate the pulser with the corresponding label.

****

The following single-key actions currently have BUGS in their operation:

A	Cycle through foreground colors.
C	Cycle through background colors.
i	Increment the number of color possibilities (modulo MAX_COLORS).
J	Cycle the Logic_On color.
j	Cycle the Logic_Off color.
M	Toggle Manhattan/point-to-point interconnections. Redraw the screen, "r", to see the effects.
Q	Toggle Q-Warp (Q entries in reverse time sequence) reporting.
q	Toggle auditing (error checking) of event queue.
S	Toggle print of diagnostics of signal queueing.
s	Toggle print of diagnostics of cell divisions.

2.24. Simulator Implementation

The major portion of the simulator is written in C. About 8500 lines of C were originated by hand, and an additional 1600 lines of C were produced by Lex from lexical specifications, and Yacc from grammatical specifications. Substantial support for windowing and graphics comes from library functions.

There are no arrays of fixed size used for the network description, hence there is no precise limit that can be given for the size of networks that may be simulated. All space is allocated from the heap including space for component and node-label strings. The event-queue also obtains space from the heap during simulation, and the length to which the queue must grow is also a determining factor of network size. As heap items are no longer needed, the space is returned for reallocation.

If large networks are being developed, you should monitor the use of heap storage by turning on the appropriate flag. As the dynamic allocations reported are made from smaller and smaller regions of memory, you know you are reaching the limit.

If you are using a system that utilizes virtual memory, such as Sun Workstations, it is unlikely you will be short of memory.

2.25. Extensions

In principle it is possible to extend your use of the simulator to include many of the types of components you might desire. If you need a new digital behavior beyond that provided by the primitive components and their hierarchical use, the Function components lets you model the behavior using the C language.

Introducing new components that have your own graphical presentation is somewhat more involved. Connect Pipe-in and Pipe-out components within your simulator usage and connect these Pipes through the operating system to other windows where you have created the desired graphical presentation driven by the pipes. Then, sim runs as one leg of the forked processes, and the Pipes allow the necessary communications. Your graphics runs in the other leg(s).

2.26. Bugs

Here is list of some known bugs and limitations in sim and allied software. The lack of correction stems from sporadic problems, sketchy reports of exact circumstances, low priority, or errors embedded deeply within sim's concept of how things inter-relate and/or should work. If you encounter any of these, your TA or instructor can probably suggest a reasonable work-around.

1. Comments are limited to 1024 characters (Operating System Default Buffer Size), or whatever the default size is for file buffers on the system you're using.
2. Power-on components may not reset (R) properly if they occur AFTER the microprocessor in the netlist.
3. The Disk-unit requires an initial Ready signal cycle before it falls into the proper Ready/Reply protocol.
4. Zooming in the screen display to zero width or height causes sim to abort. Zooming in to very small apertures prevents zooming back out correctly.

5. A Function component allows a maximum of 80 outputs.
6. Sim does not treat reset ("R") correctly in some situations. Mechanicals such as the Droid and Hexobot may not be returned to their initial conditions.
7. If you have embedded #include's, don't have subsequent #include's as the first line of the following files. Insert a blank line if necessary to avoid a sim error that exists.
8. Most, if not all, of these architecture software tools require a newline character at the end of every line, including the last line of a file. These usually are provided automatically without special attention; however, the emacs editor can allow you to create files that don't end with a newline after the last line unless you specifically type one at that position.
9. Macro references are processed slowly if there are many levels or large components in a Define. This is especially noticeable if very large Rom's are used at low levels of organization.
10. The Droid's moves are only displayed on the screen if there is a probe, clock, or something else which otherwise updates the screen.
11. Manhattan drawing of interconnect lines will not go around screen regions that are larger than single width and height.
12. The file "sim.tmp" is created and REMAINS in the current directory by a sim invocation.
13. Resizing a window can leave remnants of the old display, use the keyboard redraw "r" command to bring the display up to date.

Chapter 3: SPARC Architecture and Assembly Language Programming

3. Introduction

This chapter is an abbreviated description of the SPARC architecture, and are intended to support learning about the architecture, understanding and doing simple assembly-language programming, and implementing a gate and register-level simulation model of the SPARC microprocessor.

There is no consideration here of the traps and interrupts, the tagged-data formats, the memory-management unit, the floating-point processor nor the co-processors.

Assembly language programming has a diminished role for computer scientists these days. In times past, many programmers were engaged in assembly language programming. In the very early days of computing, until 1957 or so, almost all programmers had to be proficient with this primitive method of communicating with computers. Even after this time, when high-level language compilers began to appear, there was plenty of assembly language programming to be done. Early compilers produced code that hand-coders could easily surpass in execution-time performance. Also, many early compilers translated high-level languages to assembly language, and then proceeded to assemble that result into the machine form of the code. This intermediate pass through assembly language isn't very common any more since the two-tiered approach requires reenacting some of the very same steps at each stage of the translation. So, for overall speed of translation, many compilers integrate both functions, and translate directly from high-level languages to machine code. This is not to say there is no notion of any intermediate language in the compilation process. While any intermediate language(s) is(are) not assembly language, tree-form, regular-expression, symbolic-expression, and other intermediate languages are in common usage.

An intermediate, assembly-level version of the high-level language program can be produced by many compilers, but that version is produced only on specific request, and is not part of the normal translation process.

There are two views of an assembler. From the high-level perspective, the assembler is the translator that allows the programmer to reach the primitive capabilities of the hardware. For any given machine, not every hardware feature would have a counterpart in the C or C++ language. To reach some of these features, one would resort to assembly-language programming. Examples of hardware features that C might not provide access to include: Block Move Instructions, Bit Test and Set Instructions, and Multiplication and Division allowing extended precision.

Sometimes, even though there are no particular machine instructions that one wants to reach, extracting the ultimate performance in solving some particular problem may warrant assembly-language programming. While the optimization that modern C compilers provide gives respectable performance across the broad range of possible programs, there are a few cases where the programmer's special knowledge of how the algorithm proceeds would allow a deployment of the machine registers and memory more effectively than a generalized optimizer can. These performance issues are the usual reasons given for the use of assembly-language programming.

The low-level view of the assembler shows the features machine-level programming that relieve the programmer of having to deal with the binary patterns of instructions and the management of memory allocation for instructions and data. Each instruction type has a mnemonic code that is translated to the instruction binary pattern by the assembler. The mnemonic itself usually does not specify the complete pattern, and portions of the pattern are completed by examining the data type and address(s) given along with the mnemonic code.

While the mnemonic code translation is the most visible action of the assembler, the most important action is probably the provision for handling symbolic names for memory locations. Symbolic names can be provided for labeling both instructions and data sections of the code. The symbolic names created as labels can be referenced as parts of addresses in composing the operands for instructions.

Some of the symbolic names used within an assembly-language program do not occur as labels anywhere within the program. These correspond to external references that are expected to be satisfied by consulting the computer's supporting library. The supporting program code from the library is loaded along with the execution module to form the complete code.

3.1. Reduced Instruction Set Computers

Throughout the rapid development of computer hardware and software during the 1950's, 60's and 70', computer instruction sets became more complicated. Many of the complications resulted from the design of the hardware counterpart of the provisions in high-level languages for defining and referencing elements within complex data structures--for example: increment a variable found in a given field of a particular record that's an element of an array of such records. Most of the computer architectures were these Complex Instruction Set Computers (CISC) until the 1980's.

A few computer designers suspected that providing for these complex instruction types resulted in computer designs that were slowed in their execution of the simplest instructions, and furthermore, that these simple instructions were the dominant factor in determining how real programs actually performed. Studies of actual code being executed verified that simple things were by far the most common. Constants encountered had small values, expressions rarely had more than two terms, division was rare, multiplication was infrequent, functions were most frequently called with none or one argument, single subscripts were dominant in array usage, etc.

These simplicity notions gave rise to the Reduced Instruction Set Computers (RISC). The basic hardware premise for RISC implementation was: instructions included in the design must be executable at the rate of one instruction per clock cycle. This execution rate was to be provided using pipelining as shown in Figure 1.

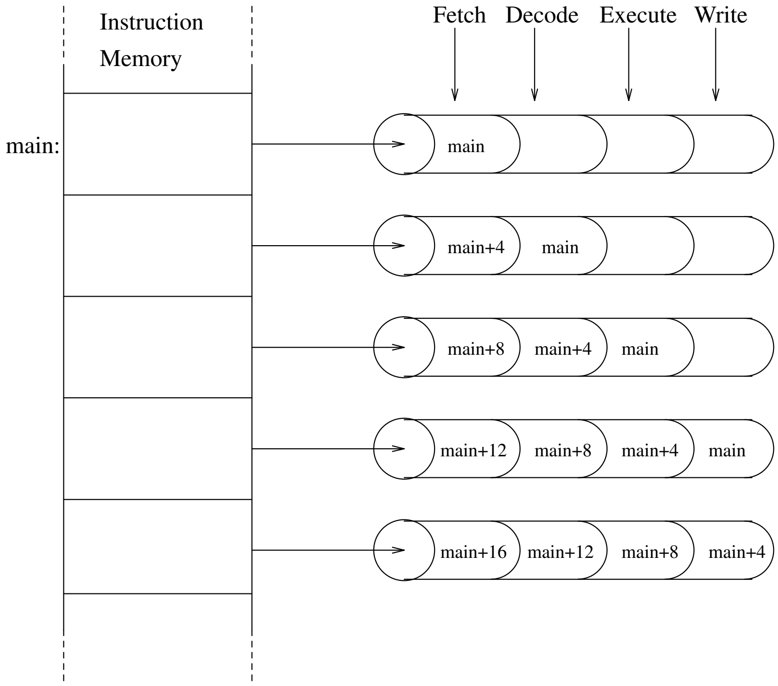


Figure 1. Instruction Pipeline.

This shows a pipeline that has four segments. There is just a single pipeline; however, the Figure shows this single pipeline, and its contents, at five successive times. At each clock tick an instruction moves one-quarter of the way through its complete execution, and at any time there are four instructions partially completed. One instruction enters the pipeline and another instruction completes its pass through the pipeline on each clock cycle. While each instruction takes four clock cycles to complete (for a four-segment pipeline), the rate of instruction completion is one instruction every clock cycle.

In some instances, one or more of the pipeline segments may be effectively empty. They would actually contain a no-operation (nop) instruction. This happens at start-up as shown in the Figure above, but it can happen for other reasons also. Effective design and programming strive to eliminate as many of these "holes" as possible.

3.2. SPARC Architecture

The SPARC (Scalable Processor Architecture) is a derivation from the RISC architecture. SPARC implements a particular instruction set (extendable in various implementations), and a particular windowing scheme of central processor registers. SPARC processors and other basic components are supplied by a half-dozen or so semiconductor manufacturers. These manufacturers adhere to a set of standards that makes them all binary-compatible. That is they will all execute the same binary code, although each manufacturer might have design variations for improved performance that are invisible to the users. For example, some SPARC integer units are built using a five-segment pipeline as opposed to the four-segment shown above. On the SPARC every instruction is 32 bits (a word) long. The instruction formats used are shown in Figure 2.

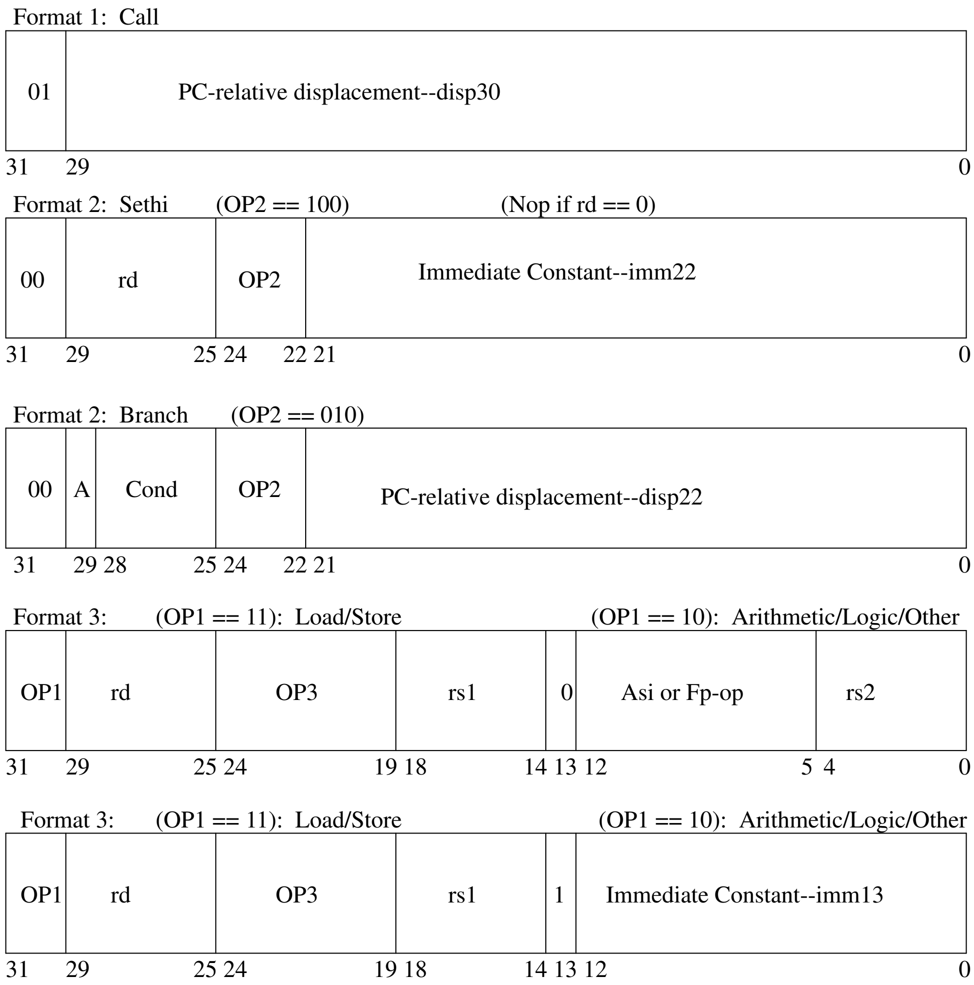


Figure 2. Instruction Formats.

All of these instruction types do not complete their operation in time for the following instruction to make use of their results. This delay in result availability can result either from latency of memory reference or from multi-cycle instructions such as multiply. Memory references are load/store instructions and branches, the former for data, the latter for the next instruction. Loads and stores are delayed one clock in their completion, but the hardware is interlocked so the programmer can ignore this delay and incur some performance penalty. To avoid this penalty arrange the code so an operand loaded from memory to a register is not used in the next instruction.

A hardware feature that allows avoiding the delay penalty when branching cannot be overlooked by the programmer. All branches, conditional, unconditional, calls and returns, are delayed by one clock cycle. That is, one more instruction will be executed at the next sequential location of the program counter before the branch takes effect. An executable instruction must be placed after every branch instruction. Often there is a reasonable computation that can be done in that slot, but a nop instruction can be used otherwise.

In addition to limiting the complexity of instructions SPARC provides a special mechanism, register windowing, is used to speed up function calling and returning. The model for interfunction communications protocol uses a stack for recording the activation record of a function. The stack LIFO protocol provides the correct sequencing for capturing return address points, and for recording actual parameters and local variables. Some RISC architectures provide the top several layers of the stack using hardware registers. This mechanism provides multiple sets of registers, and a new set can be designated rapidly when arriving at a newly called function level. This also relieves the need to save registers on behalf of your caller or callee.

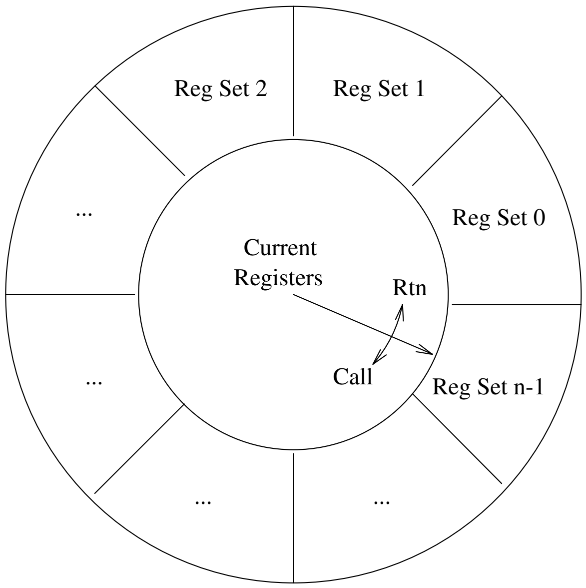


Figure 3. Register Sets.

These register sets are used in a circular manner so they can support arbitrary depths of function calls. Any overflow is backed up to the stack. The registers are allocated from the top down within the register file.

To facilitate the communication of parameters to called functions, the register sets are made to overlap as shown in Figure 4.

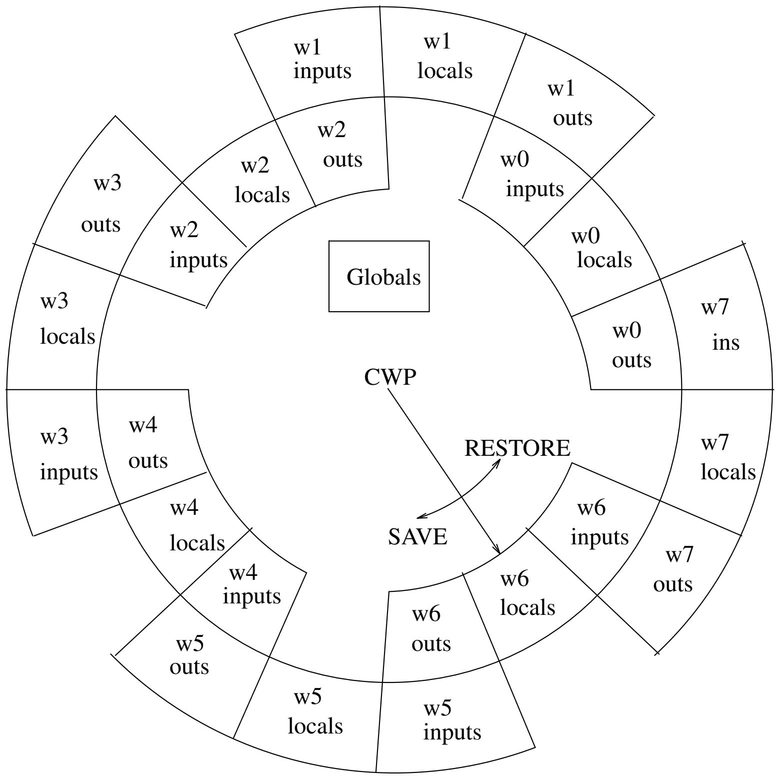


Figure 4. Overlapping Register Windows.

With this arrangement, a function receives its arguments in its input register set. The arguments have been placed there by the caller. The caller knew these same registers as its output register set. Any overflow arguments, beyond the registers available, are placed on the stack. A Current Window Pointer (CWP) holds the index to the current register set. The CALL and RET are separated from the rolling of the windows, and separate instructions, SAVE and RESTORE, move to the adjacent window. This allows avoiding changing the register set when desired. It is often desirable to avoid this change, and the possible windows-overflow overhead. This avoidance is often feasible when in a leaf (of the call graph) procedure.

The various assignments and aliases for the window registers are shown in Figure 5.

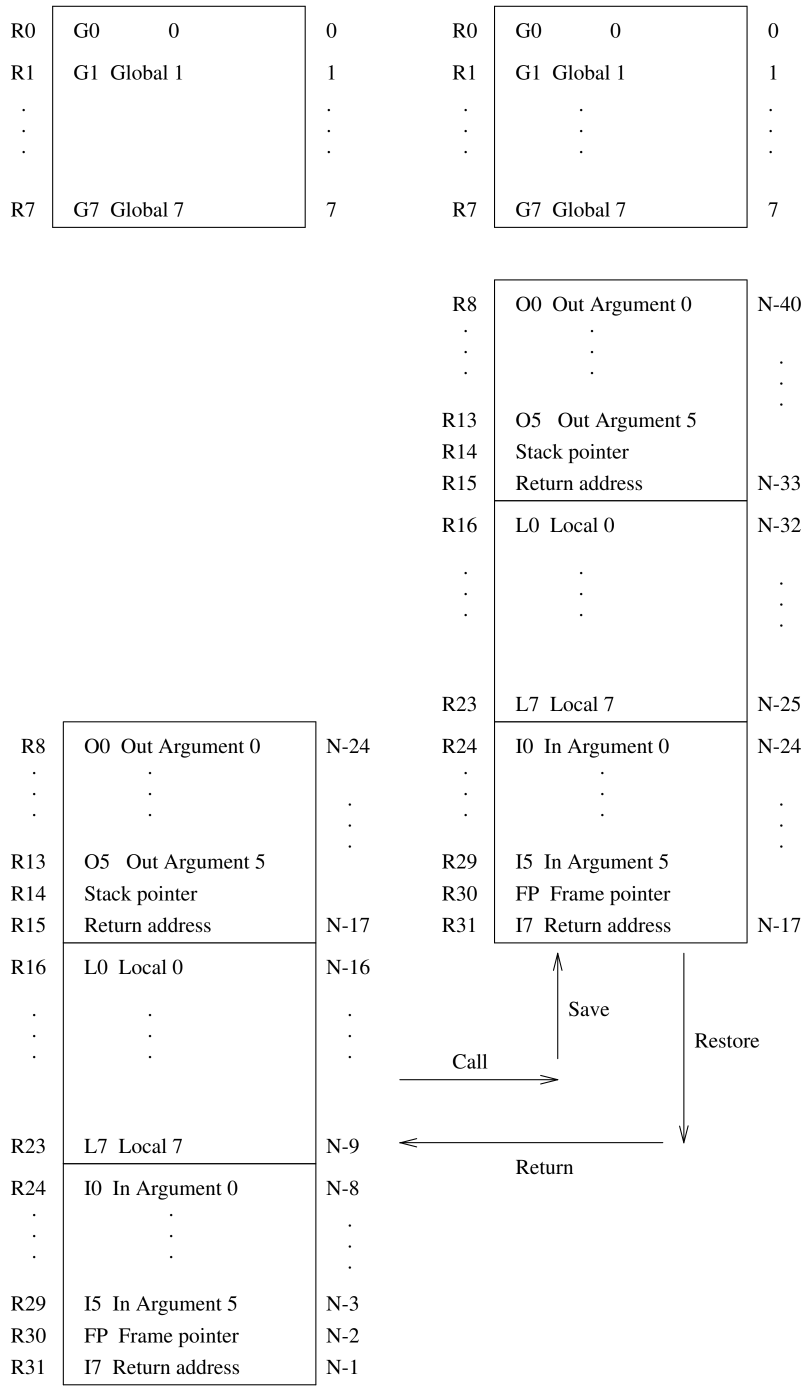


Figure 5. Window Register Assignments.

3.3. Assembly Language Versions of C++ Programs

Many compilers provide the option of creating an assembly language version of their generated code. Assume we have a file, one.c, containing:

```
main () {
 printf ( "Example main().\\n" );
 }
```

Using g++, the assembly version can be produced using:

```
> g++  -S  one.c
```

The assembler version will appear as the corresponding dot-s file, one.s, containing:

```
gcc_compiled.:
.text
LC0:
        .ascii "Example main().\\12\\0"
        .align 4
.global _main
        .proc 1
_main:
        !#PROLOGUE# 0
        save %sp,-112,%sp
        !#PROLOGUE# 1
        call ___main,0
        nop
        mov %o0,%o0
        sethi %hi(LC0),%o0
        or %lo(LC0),%o0,%o0
        call _printf,0
        nop
        mov 0,%i0
        b L1
        nop
L1:
        ret
        restore
```

Consider the following example, two.c, containing:

```
int dbl ( int );

main() {
 printf( "Doubled = %d\\n", dbl ( 13) );
 }

int dbl ( int x )  {
 return 2 * x;
 }
```

This gives the dot-s for the dbl() function as:

```
.text
        .align 4
.global _dbl__Fi
        .proc 1
_dbl__Fi:
        !#PROLOGUE# 0
        save %sp,-112,%sp
        !#PROLOGUE# 1
        st %i0,[%fp+68]
        ld [%fp+68],%o0
        sll %o0,1,%o0
        mov %o0,%i0
        b L1
        nop
L1:
        ret
        restore
```

Here it is possible to implement the dbl() function as a leaf function as (some optimizing compilers can do this):

```
.text
        .align 4
.global _dbl__Fi
        .proc 1
_dbl__Fi:
        retl
        sll %o0,1,%o0
```

Parameters

If parameters are to be communicated from caller to callee, the registers %o0 ... %o5 are used for the first six parameters. If there are any aggregate types, they are passed by reference with the reference address in the register. If there are more than six parameters, the overflow are on the stack. The return value is placed in %i0.

Aggregate return values are handled differently by different compilers. In each treatment an address of the region that is to receive the aggregate return value is communicated to the callee. The protocol for communicating this address varies--it might be in a register, on the stack or in the instruction word that is just after the delay slot in the calling sequence.

The register usage conventions are shown in the following table.

Register Usage			
in		%i7 (%r31)	return address - 8†
	%fp	%i6 (%r30)	frame pointer†
		%i5 (%r29)	incoming parameter 6†
		%i4 (%r28)	incoming parameter 5†
		%i3 (%r27)	incoming parameter 4†
		%i2 (%r26)	incoming parameter 3†
		%i1 (%r25)	incoming parameter 2†
		%i0 (%r24)	incoming parameter 1†
local		%l7 (%r23)	local 7†
		%l6 (%r22)	local 6†
		%l5 (%r21)	local 5†
		%l4 (%r20)	local 4†
		%l3 (%r19)	local 3†
		%l2 (%r18	local 2†
		%l1 (%r17)	local 1†
		%l0 (%r16)	local 0†
out		%o7 (%r15)	temporary/address of CALL instruction‡
	%sp	%o6 (%r14)	stack pointer†
		%o5 (%r13)	outgoing parameter 6‡
		%o4 (%r12)	outgoing parameter 5‡
		%o3 (%r11)	outgoing parameter 4‡
		%o2 (%r10)	outgoing parameter 3‡
		%o1 (%r9)	outgoing parameter 2‡
		%o0 (%r8)	outgoing parameter 1/return value‡
global		%g7 (%r7)	global 7/reserved
		%g6 (%r6)	global 6/reserved
		%g5 (%r5)	global 5/reserved
		%g4 (%r4)	global 4/reserved
		%g3 (%r3)	global 3/reserved
		%g2 (%r2	global 2/reserved
		%g1 (%r1)	temporary‡
		%g0 (%r0)	0

† Assumed by the caller to be preserved across function calls.

‡ Not assumed by the caller to be preserved across function calls.

Further function communication protocol is associated with the stack usage as shown by the following:

Stack Frame

		Previous Stack Frame
%fp (old %sp) -> %fp-offset ->	Space (if needed) for automatic arrays, aggregates, and addressable scalar automatics.	
%sp+offset ->	Space (if needed) for compiler temporaries.	
%sp+offset ->	Outgoing parameters beyond the sixth.	

%sp+offset ->	6 words for callee to store registers.	
%sp+offset ->	One-word hidden parameter. Address for aggregate return.	
%sp+offset -> %sp ->	16 words to save register window (in and local registers).	
	↓ Stack Growth	Next Stack Frame

Function Name Encoding

The name space wherein calling function parameters are checked for a match with the callee parameters requires an elaborate set of conventions. Each function name is translated ("mangled") according to the types of its arguments. From an example above:

_dbl__Fi:

the function name dbl is translated to:

<underline> dbl <underline> <underline> F <arg_type_list>

where "F" is the Function designator and "i" indicates the single integer argument.

Other argument types are encoded as:

Type	Encoding
void	v
char	c
short	s
int	i
long	l
long long	x
float	f
double	d
long double	r
...	e
class Name	4Name

Modifier	Encoding
unsigned	U
const	C
volatile	V
signed	S

Type	Notation	Encoding
pointer	*	P
reference	&	R
array	[n]	An_
function	()	F
pointer to member	S::*	M1S

Operator overloading encodes the operator names as:

Operator	Encoding	Operator	Encoding	Operator	Encoding
*	__ml	/	__dv	%	__md
+	__pl	-	__mi	<<	__ls
>>	__rs	==	__eq	!=	__ne
<	__lt	>	__gt	<=	__le
>=	__ge	&	__ad	|	__or
ˆ	__er	&&	__aa	||	__oo
!	__nt	˜	__co	++	__pp
--	__mm	=	__as	->	__rf
+=	__apl	-=	__ami	*=	__amu
/=	__adv	%=	__amd	<<=	__als
>>=	__ars	&=	__aad	|=	__aor
ˆ=	__aer	,	__cm	->*	__rm

Operator	Encoding
() (Call)	__cl
[] (Subscript)	__vc
constructor	__ct
destructor	__dt
operator new()	__nw
operator delete()	__dl

3.4. Assembly Language

Assembly language has the syntax:

<statement> ::= {<label>:} {<instruction>}

<comment> ::= "!" <anything-till-end-of-line> | <C-style-comment>

<line> ::= <statement> { ";" <statement> }* {<comment>}

where { } are optional fields. Labels are:

<label> ::= [a-zA-Z_$.][a-zA-Z_$.0-9]+

Contents Of

Brackets surrounding an expression in an instruction indicate a reference (contents of) as:

ld [%i3],%g1

load memory, as pointed to by register i3, into register g1.

Numbers

Numbers in instruction arguments are written as in C. Additionally, real pseudo-values are represented as:

0r3.5 -or- 0R3.5

3.5. Instructions

add reg_{rs1}, reg_{rs2}_or_imm, reg_{rd} Add

addcc	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Add and set condition codes
addx	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Add Extended
addxcc	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Add Extended and Set Condition Codes
and	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	And
andcc	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	And and set condition codes
andn	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	And-Not
andncc	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Nand and Set Condition Codes
ba{,a}	<label>	Branch Always
be{,a}	<label>	Branch on Equal
bgeu{,a}	<label>	Branch on Less or Equal Unsigned
bge{,a}	<label>	Branch on Greater or Equal
bgu{,a}	<label>	Branch on Greater Unsigned
bg{,a}	<label>	Branch on Greater
bleu{,a}	<label>	Branch on Less or Equal Unsigned
ble{,a}	<label>	Branch on Less or Equal
blu{,a}	<label>	Branch on Less Than, Unsigned
bl{,a}	<label>	Branch on Less
bneg{,a}	<label>	Branch on Negative
bne{,a}	<label>	Branch on Not Equal
bn{,a}	<label>	Branch Never
bpos{,a}	<label>	Branch on Positive
bvc{,a}	<label>	Branch on Overflow Clear
bvs{,a}	<label>	Branch on Overflow Set
call	<label>	r(15) <- PC; PC = PC + 4 * disp30
jmpl	$address, reg_{rd}$	Jump and Link; reg_{rd} <- PC
ld	$[address], reg_{rd}$	Load word
ldd	$[address], reg_{rd}$	Load double-word
ldsb	$[address], reg_{rd}$	Load signed byte
ldsh	$[address], reg_{rd}$	Load signed half-word
ldub	$[address], reg_{rd}$	Load unsigned byte
lduh	$[address], reg_{rd}$	Load unsigned half-word
mulscc	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Multiply Signed and Set CC
nop		No operation
or	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Inclusive Or
orcc	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Or and set condition codes
orn	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Or-Not
orncc	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Nor and Set Condition Codes
restore	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Backup Window Registers
		restore acts like a normal add instruction except
		$rs1$ and $rs2$ are read from the OLD window, and
		the sum is written into rd of the NEW window.
save	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Index Window Registers
		save acts like a normal add instruction except
		$rs1$ and $rs2$ are read from the OLD window, and
		the sum is written into rd of the NEW window.
sethi	$imm22, reg_{rd}$	Set the upper 22 bits, and clear the low 10
sll	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Shift Left Logical
sra	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Shift Right Arithmetic (Sign Extend)
srl	$reg_{rs1}, reg_{rs2}\_or\_imm, reg_{rd}$	Shift Right Logical
st	$reg_{rd}, [address]$	Store word
stb	$reg_{rd}, [address]$	Store byte
std	$reg_{rd}, [address]$	Store double-word
sth	$reg_{rd}, [address]$	Store half-word

sub	reg_{rs1}, $reg_{rs2}\_or\_imm$, reg_{rd}	Subtract
subx	reg_{rs1}, $reg_{rs2}\_or\_imm$, reg_{rd}	Subtract Extended
subxcc	reg_{rs1}, $reg_{rs2}\_or\_imm$, reg_{rd}	Subtract Extended and Set CC
xnor	reg_{rs1}, $reg_{rs2}\_or\_imm$, reg_{rd}	Exclusive Nor
xnorcc	reg_{rs1}, $reg_{rs2}\_or\_imm$, reg_{rd}	Exclusive Nor and Set Condition Codes
xor	reg_{rs1}, $reg_{rs2}\_or\_imm$, reg_{rd}	Exclusive Or
xorcc	reg_{rs1}, $reg_{rs2}\_or\_imm$, reg_{rd}	Exclusive Or and set Condition Codes

Where *address* is $reg_{rs1} + reg_{rs2}$ or $reg_{rs1} + sign\_ext\ imm13$.

The optional annul-bit "{,a}" on the branch instructions above allows a delay-slot-fill instruction to be annulled if a conditional branch instruction is not taken.

3.6. Pseudo Instructions

Pseudo-Instruction		SPARC Instruction(s)		Description
cmp	rs1,reg_or_imm	subcc	rs1,reg_or_imm,%g0	compare
jmp	address	jmpl	address,%g0	
call	address	jmpl	address,%o7	
tst	rs2	orcc	%g0,rs2,%g0	test
ret		jmpl	%i7+8,%g0	return
retl		jmpl	%o7+8,%g0	return from leaf
clr	rd	or	%g0,%g0,rd	clear
mov	reg_or_imm,rd	or	%g0,reg_or_imm,rd	

3.7. Pseudo-Operations

Pseudo-Op	Argument List	Description
.ascii	"string" {,"string"}*	Loads array of characters.
.asciz	"string" {,"string"}*	As above plus null byte for each string.
.skip	n	Increment the locations counter by n.
.align	boundary	1, 2, 4, or 8-byte boundary alignment.
.byte	val {,val}*	Generates sequence of initialized bytes.
.half	val {,val}*	Generates sequence of initialized halfwords.
.word	val {,val}*	Generates sequence of initialized words.
.single	val {,val}*	Generates sequence of initialized
		single-precision floating point values.
.double	val {,val}*	Generates sequence of initialized
		double-precision floating point values.
.global	label {,label}*	Declares label(s) to be external symbols.

3.8. Branch Condition Codes

Conditional branches depend on the setting of the four condition codes.

N	Z	V	C

Figure 6. Condition Codes.

These codes are kept in four bits of the program state register (PSR).

These flags are changed as a result of using an arithmetic/logic instruction that has the "cc" postfix. The flags retain their value until the next such postfixed instruction causes their replacement. Any time after the condition codes have been set and before they are modified explicitly, they can be used to determine the action of a conditional branch.

The conditional branch instruction mnemonics, the instruction-field binary pattern and the dependence on the condition codes are given in the following table.

Mnemonic	Cond	Description	cc test
bn	0000	Branch Never	0
be	0001	Branch on Equal	Z
ble	0010	Branch on Less or Equal	Z or (N xor V)
bl	0011	Branch on Less	N xor V
bleu	0100	Branch on Less or Equal Unsigned	(C or Z)
blu	0101	Branch on Less Than, Unsigned	C
bneg	0110	Branch on Negative	N
bvs	0111	Branch on Overflow Set	V
ba	1000	Branch Always	1
bne	1001	Branch on Not Equal	not Z
bg	1010	Branch on Greater	not(Z or (N xor V))
bge	1011	Branch on Greater or Equal	not (N xor V)
bgu	1100	Branch on Greater Unsigned	not (C or Z)
bgeu	1101	Branch on Less or Equal Unsigned	(C or Z)
bpos	1110	Branch on Positive	not N
bvc	1111	Branch on Overflow Clear	not V

3.9. Opcode Tables

The additional encodings for the instruction fields OP2 and OP3 and given in the following tables.

OP2[2..0]				
000	010	100	110	111
UNIMP	Bicc	SETHI	FBfcc	CBccc
		NOP		

OP3 Coding for Arith/Logic Instructions				
	OP3[5..4]			
OP3[3..0]	00	01	10	11
0000	ADD	ADDcc	TADDcc	WRY
0001	AND	ANDcc	TSUBcc	WRPSR
0010	OR	ORcc	TADDccTV	WRWIM
0011	XOR	XORcc	TSUBccTV	WRTBR
0100	SUB	SUBcc	MULScc	FPop1
0101	ANDN	ANDNcc	SLL	FPop2
0110	ORN	ORNcc	SRL	CPop1
0111	XNOR	XNORcc	SRA	CPop2
1000	ADDX	ADDXcc	RDY	JMPL
1001			RDPSR	RETT
1010	UMUL	UMULcc	RDWIM	Ticc
1011	SMUL	SMULcc	RDTBR	FLUSH
1100	SUBX	SUBXcc		SAVE
1101				RESTORE
1110	UDIV	UDIVcc		
1111	SDIV	SDIVcc		

OP3 Coding for Load/Store Instructions				
	OP3[5..4]			
OP3[3..0]	00	01	10	11
0000	LD	LDA	LDF	LDC
0001	LDUB	LDUBA	LDFSR	LDCSR
0010	LDUH	LDUHA		
0011	LDD	LDDA	LDDF	LDDC
0100	ST	STA	STF	STC
0101	STB	STBA	STF	STC
0110	STH	STHA	STDFQ	STDCQ
0111	STD	STDA	STDF	STDC
1000				
1001	LDSB	LDSBA		
1010	LDSH	LDSHA		
1011				
1100				
1101	LDSTUB	LDSTUBA		
1110				
1111	SWAP	SWAPA		

3.10. Machine Binary

The binary machine instructions can be obtained from a debugger operating on the executable code, or by hand from the various tables. As an example consider the translation of the instruction:

mov %o0,%i0

The mov is a pseudo-instruction that gives the translation:

or %g0,%o0,%i0

An *or* instruction uses Format 3 with the assignments:

Field	Entry	Description
OP1	10	Format 3
rd	11000	%i0--input register 0 is register 24
OP3	000010	Or
rs1	00000	%g0
rs2	01000	%o0--output register 0 is register 8
bit13	0	Register/register format
Asi	00000000	Alternate Space 0

Composing these fields gives the machine instruction:

<OP1>	<rd>	<OP3>	<rs1>	<bit13>	<Asi>	<rs2>
10	11000	000010	00000	0	00000000	01000

or

0xB0100008

Other translations can be found as:

Assembly	Machine Form
mov 0,%i0	0xB0102000
ld [%fp+68],%o0	0xD007A044
sll %o0,1,%o0	0x912A2001
b L1; nop; L1:	0x10800002
nop	0x01000000
ret	0x81C7E008

3.11. ADB

In addition to providing debugging with breakpointed, controlled execution with register and memory examination, the debugger adb allows listing the machine (hexadecimal) form of assembly language instructions. For example, consider the C program:

```
main() {
 int x = 13;
 printf("%d\n", x);
 }
```

If this is compiled using:

```
> g++ above.c
```

The resulting a.out file can be examined with adb using:

```
> adb
```

At the prompt, supply:

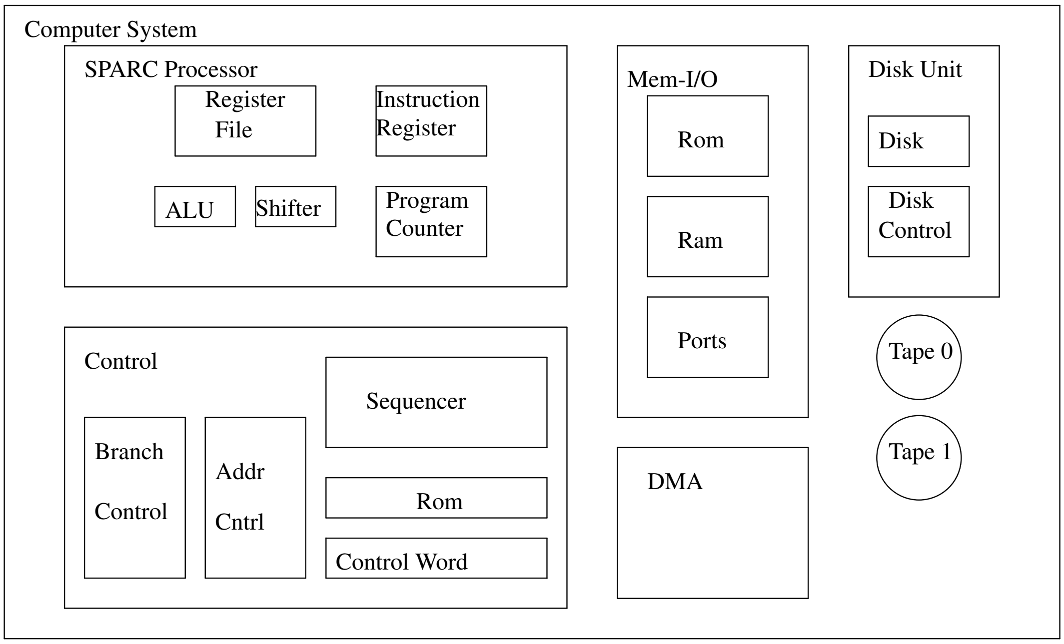
```
adb> main?X^i
```

giving the word (X) hexadecimal value, backing the location counter over that hex output (^), and giving the instruction (i) at that same location. Successive return's will move the display through the rest of the text section of a.out using the same hex-instruction format.

Address	Hexadecimal	Assembly
_main:	9de3bf88	save %sp, -0x78, %sp
_main+4:	4000000f	call ___main
_main+8:	01000000	nop
_main+0xc:	90100008	mov %o0, %o0
_main+0x10:	9410200d	mov 0xd, %o2
_main+0x14:	d427bfec	st %o2, [%fp - 0x14]
_main+0x18:	11000008	sethi %hi(0x2000), %o0
_main+0x1c:	90122290	or %o0, 0x290, %o0 ! start + 0x270
_main+0x20:	d207bfec	ld [%fp - 0x14], %o1
_main+0x24:	4000006d	call _printf
_main+0x28:	01000000	nop
_main+0x2c:	b0102000	clr %i0
_main+0x30:	10800002	ba _main + 0x38
_main+0x34:	01000000	nop
_main+0x38:	81c7e008	ret
_main+0x3c:	81e80000	restore
___main:		
___main:	9de3bf90	save %sp, -0x70, %sp

3.12. SPARC Design Notes

A model of a SPARC computer system as will be built in the Architecture Laboratory is shown below.



SPARC Computer System

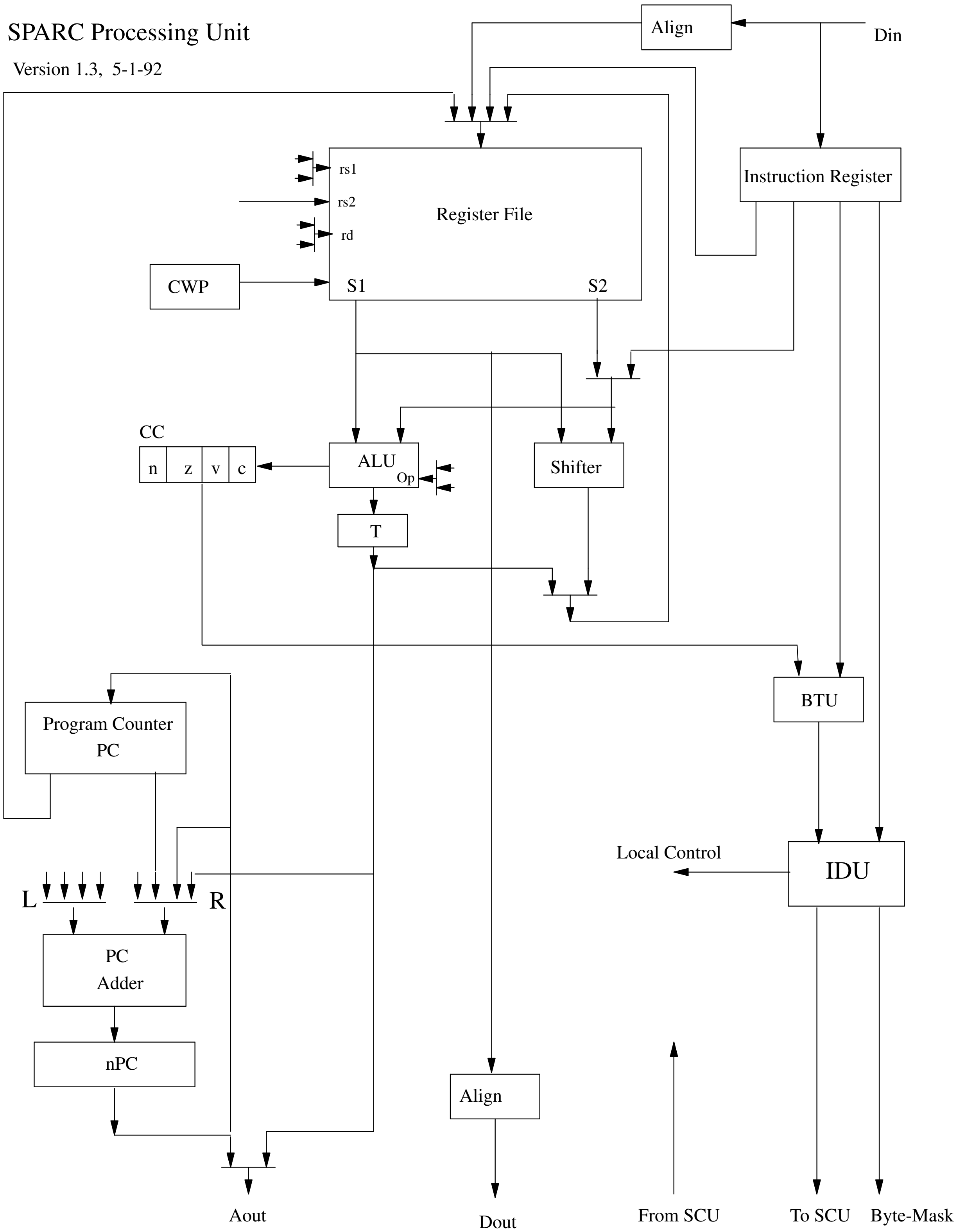
The notable differences between this model and the real machines include:

1. This model uses microprogramming for the control unit. This makes the control unit for the lab machines easier to develop, construct, change, and debug, although the time/performance penalty would probably be too great to use microprogramming for actual machines.

2. This model will not use instruction pipelining. This will help reduce the complexity of these projects.
3. Features to mask memory latency (caches) and provide the task management options that use virtual memory (Memory Management Units) are omitted for simplicity.

Sparc-processor

The registers, multiplexers and data paths of the SPARC integer unit are shown below.



SPARC Processor

This diagram shows the major data pathways required by the integer unit. These pathways and operation units are deduced from a general knowledge of the architecture the programmer knows plus the data routing options required to support the instruction set of the machine.

Much of the detailed information is still missing from the above diagram, however. The remaining details necessary to complete the design of the CPU include:

1. The additional multiplexers required to provide the addresses for the register file.
2. The sources for the multiplexers' addresses.
3. The details of the alignment units to provide byte and half-word reads and writes.
4. The combinational circuitry necessary to transform the condition codes to the forms needed for conditional branching.
5. The registers' write signals.

Any additional hardware units and interconnections can be inferred from the machine's instruction list:

alu	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
alucc	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
alux	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
aluxcc	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
branch{,a}	$disp22$
call	$disp30$
jmpl	$reg_{rs1} + reg_{rs2}\_or\_simm13$, reg_{rd}
ld	$[reg_{rs1} + reg_{rs2}\_or\_simm13]$, reg_{rd}
ldd	$[reg_{rs1} + reg_{rs2}\_or\_simm13]$, reg_{rd}
ldsb	$[reg_{rs1} + reg_{rs2}\_or\_simm13]$, reg_{rd}
ldsh	$[reg_{rs1} + reg_{rs2}\_or\_simm13]$, reg_{rd}
ldub	$[reg_{rs1} + reg_{rs2}\_or\_simm13]$, reg_{rd}
lduh	$[reg_{rs1} + reg_{rs2}\_or\_simm13]$, reg_{rd}
nop	
restore	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
save	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
sethi	$imm22$
sll	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
sra	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
srl	reg_{rs1}, $reg_{rs2}\_or\_simm13$, reg_{rd}
st	reg_{rd},$[reg_{rs1} + reg_{rs2}\_or\_simm13]$
stb	reg_{rd},$[reg_{rs1} + reg_{rs2}\_or\_simm13]$
std	reg_{rd},$[reg_{rs1} + reg_{rs2}\_or\_simm13]$
sth	reg_{rd},$[reg_{rs1} + reg_{rs2}\_or\_simm13]$

The major components used in the SPARC processor are the:

- Register-file
- Alu
- Sparc-shifter
- Adder
- Word Multiplexers

These are primitive Sim components and are described in the Components section of these notes.

Appendix I: Simulator Components

It is not practical to give exhaustive descriptions of individual component's characteristics. For the most part, the components are closely related to standard component types, and further supporting information can be found in your digital design textbook.

A valuable approach is to experiment with individual components separately if any of their characteristics is in doubt. Usually this merely amounts to connecting the component to Switch inputs and Probe outputs, and putting the component through a revealing set of excitation patterns.

In more complicated designs it is common to experience timing difficulties. To investigate these problems: the "throbs" may be turned on, Probes may be added, the simulation may be slowed, the timed signals may be displayed on a() Scope(s), and/or Clocks may temporarily be replaced with Switches or Pulsers so a more controlled investigation can be conducted. Time-probe's are powerful investigative elements, but they're hard to use; these are a last resort that should rarely be needed.

The classes of components directly available are shown in the functional listing below.

Functional Index	
Category	**Elements**
Gates	And Nand Nor Not Or Phaser
	Schmitt-trigger-inverter T-gate Tri-state-buffer Xnor Xor
Multiplexers	Mux2 Mux4 Mux8 Mux16
	Mux2x4 Mux4x4 Mux16x4 Mux2x8 Mux4x8
	Mux-mxn
Decoders	Dec2x4 Dec3x8 Dec4x16
Flip-flops	Dff Jkff
Counters & Registers	Counter Latch Presettable-counter Register
	Shift-register Up-down-counter
Ram's	Ram16x4 Ram16x8 Ram32x4 Ram32x8 Ram64x4 Ram64x8
	Ram256x8 Ram1024x8 Ram4096x8 Ram16384x8
Rom's	Rom16x4 Rom16x8 Rom16x12 Rom16x16
	Rom32x4 Rom32x8 Rom32x12 Rom32x16
	Rom64x4 Rom64x8 Rom64x12 Rom64x16
	Rom64x24 Rom64x32 Rom64x40 Rom64x48
	Rom128x24 Rom128x32 Rom128x40 Rom128x48
	Rom256x4 Rom256x8 Rom256x12 Rom256x16
	Rom256x24 Rom256x32 Rom256x40 Rom256x48
	Rom256x56 Rom256x64
	Rom1024x32 Rom4096x32 Rom1024x64 Rom4096x64
Timing	Cable Clock Gate Meta-control One-shot Power-on Sync
Manual Inputs	Pulser Switch
Input/Output	Disk-unit Printer
	Byte-in Byte-out File-in File-out
	Pipe-in Pipe-in-async Pipe-out Pipe-out-async
	Standard-out Tape Sound
Instrumentation	Byte-probe Byte-probe-h Hex-probe Hex-probe-h
	Probe Logic-analyzer Scope Seven-segment Time-probe
Micros	Microprocessor Microcomputer
SPARC Components	Register-file Alu
	Shift-unit Adder
Animation	Cubix Disk Droid Gantry-robot Gemini Hexobot
	Links Taurus X-stepper Y-stepper
Displays	Cube Die
Pseudo	Alias Define End Endef Function Module

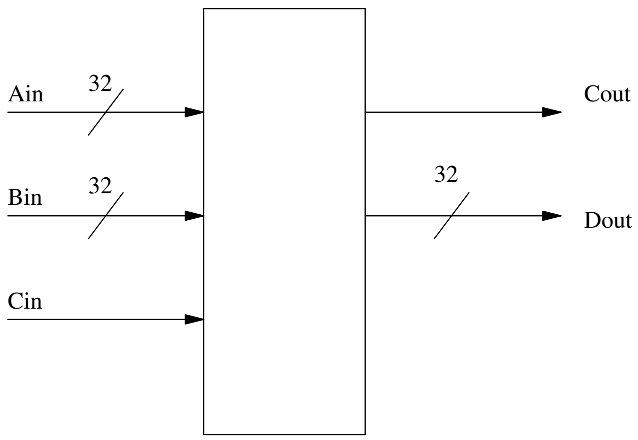
A range for the component delays is given in the descriptions below. The smaller end of the range shows the most rapid propagation of input effects to the output for the given type of device. The higher end of the range represents the maximum delay the given type will exhibit.

The examples below use explicit function names for signal lines in some cases, such as "Enable"; however, when a component requires a vector of labels for component inputs or outputs, an abbreviated symbol is used. The following associations will be found: i or I for data-input; a or A for address; c or C for control, m or M for mode, o or O for output. A suffix modifier will give the positional significance, e.g. a three-bit address could have the labels: A2 A1 A0.

The numeric parameters may be given in decimal, or in octal if a leading zero is used, or in hex if the prefix 0x (zero x) is used. The maximum value a numeric parameter may have is 32767, 077777 or 0x7FFF.

Adder

The Adder component has 65 inputs and 33 outputs when used as a Sparc Program-Counter Adder. Other widths are allowable by using the appropriate size of input and output paths.



Adder (For Sparc Program-Counter Arithmetic)

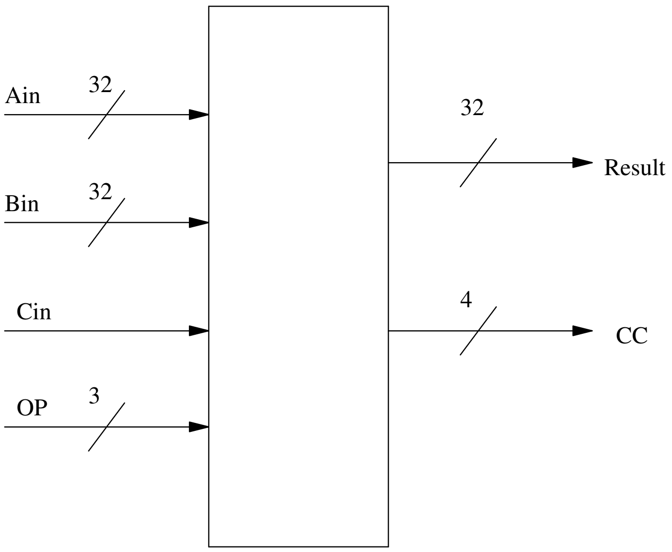
Example:
Adder 1b-3c Ain[31-0] Bin[31-0] cin | cout out[31-0];

Alias

Declaration that two names represent the same signal. May be used repeatedly to declare three or more equivalent names. Accepts two lists, with the corresponding names in each being equivalent. Either name may be used before an alias declaration, but both may not be or they will already have been made independent. There is no difference between a name appearing before or after the separator.
Example: With associations: (A[2],x), (A[1],y).
Alias A[2..1] | x y;

Alu

The Alu component used as a Sparc-alu has 68 inputs and 36 outputs. The Alu can be used with data paths that are other than 32 bits wide by adjusting the number of input and output connections accordingly.



SPARC Arithmetic/Logic Unit

The operation of the ALU is controlled by the OP code as given in the following table:

ALU Function Codes	
OP	Operation
000	Add
001	And
010	Or
011	Xor
100	Sub
101	Nand (AndN)
110	Nor (OrN)
111	Xnor (XorN)

Example:

Alu 1b-4c Ain[31-0] Bin[31-0] Cin OP[2-0] | CC[3-0] Out[31-0];

And
Variable number of inputs.
Delay: 15-22
Example: (With five inputs)
And 2K a b c d e Out;

Byte-in
(PC version only) One input, eight outputs and one numeric parameter. This component provides an interface to the real (non-simulated) world of the computer's I/O bus. Inputs from the port are the outputs of this component. The input port is read on the leading edge of the component's clocking signal. The numeric parameter gives the port address.

```
************************************************
* This component is potentially hazardous since *
* disks, controllers, virtually all the I/O can *
* be accessed without restriction or protection.*
************************************************
```

See the computer hardware description of the I/O address space for further information.
Delay: 10-15
Example:
  Byte-in 4B clock out[7-0] 0x03F9 ;

<u>Byte-out</u>
(PC version only) Nine inputs, zero outputs and one numeric parameter. This component provides an interface to the real world through the computer's I/O bus. Inputs to this component are sent out through the port on the leading edge of the component's clocking signal. The numeric parameter gives the port address.

```
************************************************
* This component is potentially hazardous since *
* disks, controllers, virtually all the I/O can *
* be accessed without restriction or protection.*
************************************************
```

See the computer hardware description of the I/O address space for further information.
Delay: 10-15
Example:
  Byte-out 4B data[7-0] clock 0x03AE ;

<u>Byte-probe</u>
Eight-input logic display.
Example:
  Byte-probe 1A A[3..0] B[2..0] C;

<u>Byte-probe-h</u>
Eight-input logic display with horizontal orientation.
Example:
  Byte-probe-h 1A A[3..0] B[2..0] C;

<u>Cable</u>
One input and one output. The numeric parameter gives the nominal time required for a signal change to propagate the length of the cable. The use of this component is not encouraged, and usage may require justification -- ask your TA. The maximum length is 32767.
Delay: Length-(Length+100)
Example:
  Cable 1A in out 10000;

<u>Clock</u>
One output, one symbolic parameter, three integer parameters. The symbolic parameter specifies the quiescent value of the output. The integer parameters are: "on" duration, offset from t = 0, and period.
Delay: 10-15
Example:
  Clock 4W-5X Clk ONE 200 100 500;

<u>Counter</u>
Two inputs, variable number of outputs.
A synchronous counter with active-low reset signal. The count occurs on the falling edge.
Delay: 15-22
Example: (With five outputs)

Counter aB Reset Count | O4 O3 O2 O1 O0;

Cube

No inputs nor outputs. This presents a 3-d-like image of a cube on the screen.
Example:
  Cube 3W;

Cubix

Seven inputs and no outputs. This presents a 3-d-like image of a cube on the screen. The first six inputs are three groups of two lines each. Each pair controls rotation about one of the axes according to the values:
Ix1 Ix0: 0,0 - Hold; 0,1 - Clockwise; 1,0 - Counterclockwise; 1,1 - Home position.
The last input is the drive signal, as it goes low the cube position is adjusted according to the input selections. Cube rotations are in 22.5 degree steps.
Example:
  Cubix 3a Ix1 Ix0 Iy1 Iy0 Iz1 Iz0 drive;

Dec2x4

Two inputs to be decoded, one active-low input for enabling. There are four active-low outputs.
Delay: 15-22
Example:
  Dec2x4 2A I1 I0 Enable O3 O2 O1 O0;

Dec3x8

Three inputs to be decoded, one active-low input for enabling. There are eight active-low outputs.
Delay: 15-22
Example:
  Dec3x8 4X I2 I1 I0 Enable O7 O6 O5 O4 O3 O2 O1 O0;

Dec4x16

Four inputs to be decoded, one active-low input for enabling. There are sixteen active-low outputs.
Delay: 15-22
Example:
  Dec4x16 4X I[3..0] Enable O[15..0];

Define

Introduces a macro definition that can be referenced (earlier or later) by name. The reference will supply the proper interface parameters. The body continues through the corresponding Endef statement, and may include other macro references. Defines are expanded with Module encapsulation automatically, and the internal coordinate system and nodes will be localized.
Example: (Could extend over multiple lines.)
  Define Dbl-Not a | b; Not 1a a c; Not 1b c b; Endef;

Dff

Edge-triggered D flip-flop with asynchronous, active-low Set and Clear. The D input is captured on the rising edge of the clock. This is a functionally modeled device.
Delay: 10-15
Example:
  Dff aa Set D Clk Clr Q Q';
If gate behavior is desired, a "Define" using six Nands can be used.

```
Define DFF preset din clock clear | q q';
  Nand 3Y preset LOC10 LOC8 LOC7;
  Nand 3Y LOC7 clock clear LOC8;
  Nand 3Y LOC8 clock LOC10 LOC9;
  Nand 3Y LOC9 din clear LOC10;
  Nand 3Y preset LOC8 q' q;
  Nand 3Y q LOC9 clear q';
Endef;
```

Die

Seven LED-like spots in a Die configuration.

Spot configuration		
a		e
b	d	f
c		g

Example:

```
Die 9x a b c d e f g;
```

Disk

There are two inputs and no outputs. The first input controls the direction of spin, one for counterclockwise, and the second input activates the spin when high. Only the graphical presentation is currently implemented.

Example:

```
Disk 2b ccw spin;
```

Disk-unit

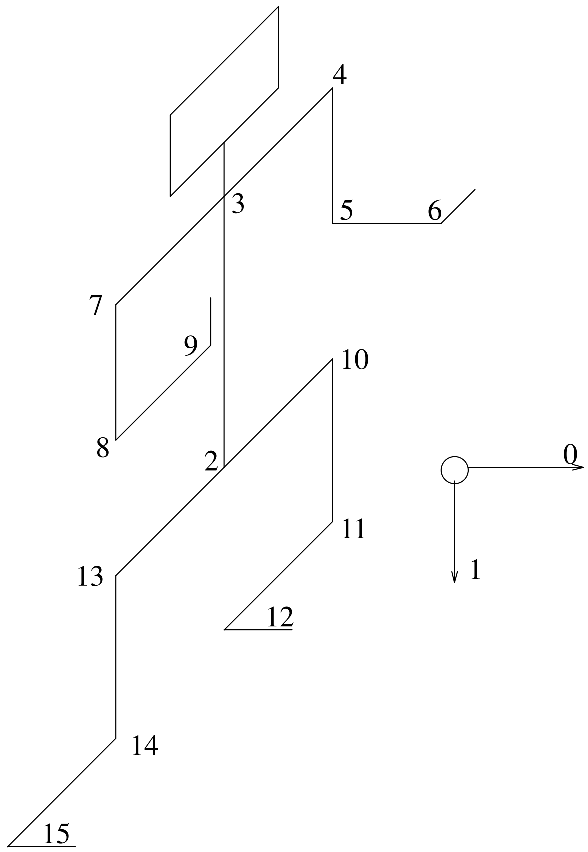
The Disk-unit has 13 inputs and 13 outputs, and accommodates up to five, tape-like, external files. See the Disk-unit description section elsewhere in these notes.

Example:

```
Disk-unit "file_0,file_1" 2b In[7-0] Fun[3-0] Ready Out[7-0] State[3-0] Reply;
```

Droid

There are ten inputs and no outputs. There are 36 numeric parameters required for initialization. The inputs consist of four address lines, two data lines, two mode lines, and a strobe and a reset line, in that order. The reset line is active-low and causes all internal registers to be cleared. When the strobe goes low, the mode lines determine the operation. The Droid has two banks of internal registers each containing 16 registers. One bank contains the actual position of rectilinear motion or joint rotations, and the other bank contains the present incremental-action specification for the corresponding motion or joint rotation.



Droid with Numbered Joints

Angular displacements are measured counterclockwise. Joints 2, 10 and 13 assume zero degrees is horizontal and to the right. Joints 3, 4 and 7 have the line 2-3 as their zero-degree reference. The remaining joints, n, have the line from n-1 to n as their zero-degree reference.

Although the Droid is shown as an oblique view, there is only a two-dimensional figure and all motion is in two-dimensions.

The Mode lines determine one of four actions as:

Mode[1..0]:
0,0 - No action
0,1 - Clear all the registers
1,0 - Load according to the Data Lines
1,1 - Execute one composite movement

When the load mode is selected, the values on the data lines determine the actions on the addressed register in the following way:

Data[1..0]: 0,0 - Zero the incremental register
0,1 - Set the incremental register to +1 (10 degrees)
1,1 - Set the incremental register to -1 (-10 degrees)
1,0 - Zero the accumulator register

The numeric parameters determine the initialization as follows:

0 - Droid width in percent of screen-cell width.
1 - Droid height in percent of screen-cell height.
2 - Droid x-increment step in percent of screen-cell width.
3 - Droid y-increment step in percent of screen-cell height.
4..19 Accumulator registers initial contents in degrees.
20..35 Incremental registers initial contents in degrees.

Although initialization values are given in degrees, the internal registers contain only the number of ten-degree steps involved.

Example: (Showing initialization to upright position shown above)

```
Droid 2b Addr[3..0] Data[1..0]
      Mode[1..0] strobe reset
      10 10 2 3
/* 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 */
   0 0 90 0 180 90 70 180 130 70 270 330 120 270 330 120
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   ;
```

<u>End</u>
Terminates the scope of the corresponding Module.
Example:
  End ;

<u>Endef</u>
Terminates the scope of the corresponding Define.
Example:
  Endef ;

<u>File-in</u>
One input and nine outputs. The specified file provides the outputs for this component a byte at a time. The next file byte is read on the leading edge of the component's clocking signal. The ninth output gives the status, ONE if a valid character is read, ZERO otherwise -- probably EOF.
Delay: 10-15
Example:
  File-in "FILENAME" 4B clock out[7-0] valid ;

<u>File-out</u>
Nine inputs and one output. Inputs to this component are sent to the designated file on the leading edge of the component's clocking signal. The output gives the status, ONE if a valid transfer, ZERO otherwise.
Delay: 10-15
Example:
  File-out "FILENAME" 4B data[7-0] clock valid ;

<u>Function</u>
Variable numbers of inputs and outputs, 80 outputs maximum. This component must be supported by supplemental object-code given in an external file.
Example:
  Function "file-name" 5e In[5-0] | Out[2-0];

<u>Gantry-robot</u>
Four inputs, six outputs.
The falling edge of the drive signal moves the arm according to:
c1,c0: (0,0) (0,1) (1,0) (1,1) :: (up) (right) (left) (down)
C1 and C0 must not be in transition when the drive makes its transition.
Horizontal and vertical 3-bit outputs give the current position.
The output values are coded as: (down) > (up) and (right) > (left)
The outputs are indeterminate initially--before the first active drive signal transition.
The reset signal is required but has no effect.

Delay: 20-30
Example:
  Gantry-robot 2b c1 c0 drive reset h2 h1 h0 v2 v1 v0;

Gate

Two inputs and one output. A clock signal is gated to the output under control of a gating signal. Only complete, positive clock pulses appear at the output regardless of the timing of the gate signal. Gate is high-active.
Delay: 10-15
Example:
  Gate 2b clk2 gate g-clock;

Gemini

Six inputs and no outputs. Two blocks sliding and colliding. Adjustable masses, frictional forces and impulse drivers. The inputs: Menu direction, menu move, adjustment direction, adjustment, left-block impulse drive, right-block impulse drive. The menu move, adjustment, and impulses occur on the rising edge of their input signals.
This component probably is a little shaky! Ask the Prof.
Delay: 10-15
Example:
  Gemini 2b rt-left menu rt-left adjust imp-left imp-right;

Hex-probe

Four-input logic display.
Example:
  Hex-probe 1A A[3] B[2..1] C;

Hex-probe-h

Four-input logic display with horizontal orientation.
Example:
  Hex-probe-h 1A A[3..0] ;

Hexobot

Seven inputs and six outputs.
The rising edge of the Drive signal causes a move as:
C1,C0 : (0,0) (0,1) (1,0) (1,1) :: (BK) (CW) (CCW) (FWD)
C1 and C0 must not be in transition when the Drive makes its active transition. Two inputs, S1 and S0, control the step size the hexobot takes when moving forward or backward:.
S1 S0 : (00,01,10,11) : (0, 1, 2, 3)--when 3, the hexobot moves one percent of the cell width for each step; correspondingly smaller steps for the other values.
The Erase input, when a ONE, causes the hexobot to wipe out the lines encountered as it travels. If this input is a ZERO, the hexobot might restore objects it encounters. The Trail input, when a ONE, causes the hexobot to leave a trail of its travels.
The six outputs are activated by switches on the hexobot's skirt. These switches close when the hexobot's skirt extends beyond its playing field.
The turn increment is fixed at 10 degrees.
Delay: 20-30
Example:
  Hexobot 2b C1 C0 S1 S0 Erase Trail Drive o5 o4 o3 o2 o1 o0;

Jkff

Master/slave flip-flop with asynchronous, active-low Set and Clear. The clock input is normally low. The master rank accepts inputs when the clock is high, and the falling edge of the clock gates the master to the slave rank.
Delay: 10-15
Example:
  Jkff aa Set J Clk K Clr Q Q';
This is a functionally modeled device; if gate behavior is desired, a "Define" using eight Nand gates can be used.

```
Define JKFF preset jin clock kin clear | q q';
  Nand 2X preset clock kin q LOC2;
  Nand 2X jin clock clear q' LOC1;
  Nand 2X LOC2 clear LOC3 LOC4;
  Nand 2X preset LOC1 LOC4 LOC3;
  Nand 2X LOC4 LOC2 LOC6;
  Nand 2X LOC3 LOC1 LOC5;
  Nand 2X LOC6 clear q q';
  Nand 2X preset LOC5 q' q;
Endef;
```

Latch

This is a positive-edge-triggered, enabled, variable-width data latch. The enable is active-low. The inputs are captured on the positive going edge of the load signal.
Delay: 10-15
Example: (With a width of three bits)

```
Latch 5R I2 I1 I0 Load Enable | O2 O1 O0;
```

Links

Seven inputs and no outputs. Three numeric parameters. This presents a 3-bar linkage image on the screen. The three numeric parameters give the lengths of the three bars of the linkage. The first six inputs are three groups of two lines each. Each pair controls rotation about one pin of each link according to the values:
Ia1 Ia0: 0,0 - Hold; 0,1 - Clockwise; 1,0 Counterclockwise; 1,1 - 0 degrees.
The last input is the drive signal, as it goes low the linkage position is adjusted according to the input selections. Joints step in 10 degree increments.
Example:

```
Links 3a Ia1 Ia0 Ib1 Ib0 Ic1 Ic0 drive 600 400 200;
```

Logic-analyzer

Variable number of inputs. The inputs are displayed as functions of time using the character set {0,?,1}. A transition to ONE of the reset line re-initializes the display. The first numeric parameter sets the sampling time. The second numeric parameter gives the character dot-width of each display sample.
Example: (With three inputs to be displayed)

```
Logic-analyzer 4x I2 I1 I0 reset 10 7;
```

Meta-control

A meta-level component controlling the simulation duration.
When the input goes to ONE, the simulation is terminated, as if by
a CNTRL-C from the keyboard.
Example:

```
Meta-control 3D Sig;
```

Microcomputer

A single-chip 8-bit microcomputer, the 4x8, with 256 bytes of combined memory and memory-mapped I/O. There are 224 bytes of ROM, 28 bytes of RAM, two bytes of input, and two bytes of output. See the separate sections describing the 4x8.
Delay: 10-15
Example: (Not all the required ROM values are shown here, but they must be included.)

```
Microcomputer-4x8 2a
 In1[7-0] In0[7-0]
     Int Clk CPU-Reset
       Out1[7-0] Out0[7-0] Int-Ack Reset
         0x00 0x01 0x02 ...
         /* Constants continuing for a total of 256 bytes. */
```

Microprocessor

An eight-bit microprocessor, the 4x8. This microprocessor has four, eight-bit registers. Unlike a microcomputer,

the microprocessor has only a CPU, no memory or I/O built-ins. See the separate sections describing the 4x8.
Delay: 10-15
Example:

```
Microprocessor  2X  In[7-0] Int Clk CPU-Reset
        Out[7-0] Mode[1-0] Strobe Int-Ack Reset;
```

Module

This header statement introduces a Module. The subsequent statements, until the corresponding End, define the components within this Module. Modules may be nested and have variable numbers of inputs and outputs. This is a graphical organization device only without any logical significance. It is usually preferable to have Modules made automatically through the use of a "Define", because those Modules are constructed with localized internal nodes and coordinate system.
Example: (With five inputs and three outputs)

```
Module 4S Ia Ib Ic Id Ie | Ox Oy Oz;
```

Mux2

This multiplexer has two data inputs and an address input. The address selects between the data inputs and routes that value to the single output.
Delay: 20-30
Example:

```
Mux2  2A I1 I0 Addr Out;
```

Mux4

This multiplexer has four data inputs and two address inputs. The address selects the data input and routes that value to the single output.
Delay: 20-30
Example:

```
Mux4  3G I3 I2 I1 I0 a1 a0 Out;
```

Mux8

This multiplexer has eight data inputs and three address inputs. The address selects the data input and routes that value to the single output.
Delay: 20-30
Example:

```
Mux8  23 I7 I6 I5 I4 I3 I2 I1 I0  a2 a1 a0 Out;
```

Mux16

This multiplexer has sixteen data inputs and four address inputs. The address selects the data input and routes that value to the single output.
Delay: 20-30
Example:

```
Mux16  2A  I[15..0]  A[3..0]  Out;
```

Mux2x4

This multiplexer has two input ports of four bits each. The address selects the data input port and routes those values to the single output port.
Delay: 20-30
Example:

```
Mux2x4 2A  I1[3-0] I0[3-0]  Addr  Out[3-0];
```

Mux4x4

This multiplexer has four input ports of four bits each. The address selects the data input port and routes those values to the single output port.
Delay: 20-30
Example:

```
Mux4x4 2A I3[3-0] I2[3-0] I1[3-0] I0[3-0]  A[1-0]  Out[3-0];
```

Mux16x4

This multiplexer has sixteen input ports of four bits each. The address selects the data input port and routes those values to the single output port.

Delay: 20-30
Example:
Mux16x4 2A I15[3-0] Others[59-16]
I3[3-0] I2[3-0] I1[3-0] I0[3-0] Addr[3-0] Out[3-0];

Mux2x8
This multiplexer has two input ports of eight bits each. The address selects the data input port and routes those values to the single output port.
Delay: 20-30
Example:
Mux2x8 2A I1[7-0] I0[7-0] Addr Out[7-0];

Mux4x8
This multiplexer has four input ports of eight bits each. The address selects the data input port and routes those values to the single output port.
Delay: 20-30
Example:
Mux4x8 2A I3[7-0] I2[7-0] I1[7-0] I0[7-0] A[1-0] Out[7-0];

Mux-mxn

The Mux-mxn multiplexer component provides selection among m words, n bits long. The number of words, m, must be a power of two. The input ports must be fully populated, that is m * n port lines total. The input selection must be done using log2(m) signal lines.

Example: (Selection from four, four-bit words.)

Mux-mxn 1a A[3-0] B[3-0] C[3-0] D[3-0] S[1-0] | Out[3-0];

Nand
Variable number of inputs.
Delay: 10-15
Example: (With three inputs)
Nand 4E a b c d;

Nor
Variable number of inputs.
Delay: 10-15
Example: (With five inputs)
Nor 4E a b c d e f;

Not
Delay: 10-15
Example:
Not 1W Sig Sig';

One-shot
This device emits a single pulse for each edge-trigger activation. The first symbolic parameter is the quiescent output, and the second is the triggering direction for the input. The integer parameter is the duration of the emitted pulse. The use of this component is not encouraged, and usage may require justification -- ask your TA.
Delay: 10-15
Example:
One-shot 2x in out ZERO ONE 50;

Or
Variable number of inputs.
Delay: 15-22
Example: (With two inputs)
Or 3J A B Or(A,B);

Phaser

One input and two outputs. The first output follows the input in a Schmitt-trigger-like fashion, and the second output is the complement of the first, but without the further delay that would be caused by an inverter. That is, the two outputs are timed the same, but are complements. This device is convenient for controlling tri-state-buffers.

Delay: 10-15

Example:

```
Phaser 1W Sig Siga Siga';
```

Pipe-in

(UNIX only) One input and nine outputs. The named pipe provides the outputs for this component a byte at a time. The next pipe byte is read on the leading edge of the component's clocking signal. The ninth output gives the status, ONE if a valid character is read, ZERO otherwise -- pipe empty or closed for writing, probably.

Delay: 10-15

Example:

```
Pipe-in "PIPENAME" 4B clock out[7-0] valid ;
```

Pipe-in-async

(UNIX only) No inputs and eight outputs. The named pipe provides the outputs for this component a byte at a time. The next pipe byte is read when it becomes available on the input pipe. A limit of 16 of these components may be included.

Delay: 10-15

Example:

```
Pipe-in-async "PIPENAME" 4B out[7-0] ;
```

Pipe-out

(UNIX only) Nine inputs and one output. Inputs to this component are sent to the designated pipe on the leading edge of the component's clocking signal. The output gives the status, ONE if a valid transfer, ZERO otherwise.

Delay: 10-15

Example:

```
Pipe-out "PIPENAME" 4B data[7-0] clock valid ;
```

Pipe-out-async

(UNIX only) Eight inputs and no outputs. Inputs to this component are sent to the designated pipe whenever there is any change in the ZERO/ONE status of the inputs. There is no output if any of the inputs is other than ZERO or ONE.

Delay: 10-15

Example:

```
Pipe-out-async "PIPENAME" 4B data[7-0] ;
```

Power-on

The single output is normally used for state initialization. The single symbolic parameter is the quiescent value immediately upon start up. The integer parameter gives the time that the output changes to the complement of the quiescent value to remain there for the remainder of the simulation.

Example:

```
Power-on 3Z clr ZERO 100;
```

Presettable-counter

This variable-length counter has asynchronous, active-low Load and Reset signals. The count occurs on the falling edge of the Count signal.

Delay: 15-22

Example: (As a four-bit counter)

```
Presettable-counter aB I3 I2 I1 I0 Load Reset Count |
    O3 O2 O1 O0;
```

Printer

An on-screen printer with 10 inputs and two outputs. The 8 data inputs (ASCII coded) determine the character printed or control function executed. The active-low strobe input times the printer's action. The busy output is an active-low indication of the printer's status. The error output (active-high) indicates a data-overrun, that is, a

strobe occurs while the printer is busy. The reset signal is active-low, and the resetting action occurs on the rising edge of the signal. The following control codes are effective (numeric values are base 10): LF (10), FF (12) and CR (13). The form-feed (FF) clears the Printer's pane and resets printing to the first position in the top row of the pane. The only automatic action the printer has is to print successive characters in successive horizontal positions and to restrict the printing to the Printer's pane. Any other desired action must be initiated with the appropriate control codes.
The screen-cell-size of the printer determines the number of rows and columns that may be printed. The font size is not adjustable.
Delay: 70-100 (?)
Example: Printer H3 D7 D6 D5 D4 D3 D2 D1 D0 strobe reset busy error;

Probe
A logic probe with a single input.
Example:
Probe 3D Sig;

Pulser
A momentary switch. The symbolic parameter is the quiescent value. The integer parameter is the duration of the output. There is a maximum of eight pulsers that may be used. The first occurrence is associated with the "a" key, the second with "b", etc. through "h".
Delay 10-15
Example:
Pulser 2C Pa ZERO 100;

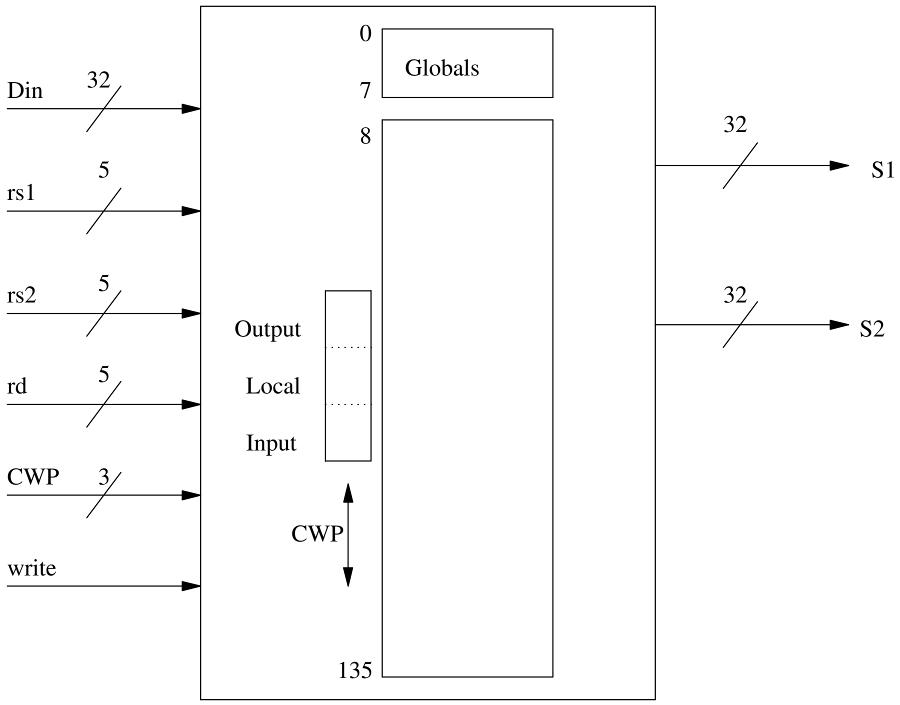
Ram16x4
Random access memory: 16 words by 4 bits. Active-low write and enable. Outputs are zero when not enabled.
Delay: 10-15
Example:
Ram16x4 2B I3 I2 I1 I0 a3 a2 a1 a0 R/W' Enable O3 O2 O1 O0;

Ram16x8 Ram32x4 Ram32x8 Ram64x4 Ram64x8...
Additional Ram's available--see the Functional Index at the beginning of this section. The number of data and address lines vary according to size.

Register
This is a variable-width register. The inputs are captured by an active-low load signal.
Delay: 10-15
Example: (With a width of three bits)
Register 5R I2 I1 I0 Load | O2 O1 O0;

Register-file

The Register-file is a component containing the windowing register set for a SPARC integer unit. This component has 51 inputs and 64 outputs.



SPARC Register File

The component has no knowledge of the SAVE and RESTORE concept. The windowing features are controlled externally, and presented to the Iregs component as the Current Window Pointer (CWP) value. The byte and half-word operations are also implemented externally.

Operation features include:

1. The CWP will normally be started at 111 (7), and be decremented for SAVE's and incremented for RESTORE's.
2. Data is captured in registers by setting rd to address the register, providing the data on the Din lines, and writing with the falling edge of the write signal.
3. Data is obtained for S1 and S2 asynchronously, as addressed by rs1 and rs2 respectively.
4. All registers except for g0 initially have UNKNOWN contents. Once registers take on known values, they do not become UNKNOWN by changes to CWP.
5. UNKNOWN's on Din are captured and retained within any properly addressed register.
6. Writing (write's falling edge) when rd is UNKNOWN or when the CWP is UNKNOWN is disastrous, and will generally result in massive changes to register contents.

Sparc-iregs		
Inputs[51]	Size	Description
Din	32	Data Input
rs1	5	Address of Register Source 1 (S1)
rs2	5	Address of Register Source 2 (S2)
rd	5	Address of the Destination Register
CWP	3	Current Window Pointer
write	1	Write the Data Input in the Destination Register (falling edge)
Outputs[64]		
S1	32	Source Operand 1
S2	32	Source Operand 2

The registers 8 through 135 are treated as a circular configuration; the Input registers for window 7 are the output registers for register 0.

Example:

```
Register-file 0b-9c
  Din[31..0]
  rs1[4..0] rs2[4..0] rd[4..0]
  CWP[2..0] write
  s1[31..0] s2[31..0];
```

<u>Rom16x4</u>
A read-only memory with 16 words by 4 bits. Active-low enable. The outputs are zero when the Rom is not enabled. The contents are listed from low to high address and may be entered in decimal, octal or hex.
Delay: 10-15
Example: (With contents set equal to address)

```
Rom16x4 2B  a3 a2 a1 a0  enable  O3 O2 O1 O0
  0 1 2 3 0x4 5 6 7 010 011 0xA 11 014 13 14 0xf;
```

<u>Rom16x8 Rom16x12 Rom16x16</u>
<u>Rom32x4 Rom32x8 Rom32x12 Rom32x16</u>
<u>Rom64x4 Rom64x8 Rom64x12 Rom64x16</u>
<u>Rom256x4 Rom256x8 Rom256x12 Rom256x16</u>
Additional Rom's available. The numbers of address lines, outputs, and contents vary according to size. A single numeric constant represents each word's value.

<u>Rom256x24</u>
A read-only memory with 256 words of 24 bits each. Active-low enable. The outputs are zero when the Rom is not enabled. The contents are listed from low to high address and may be entered in decimal, octal or hex. Each numeric constants gives the value of one byte. The constant bytes are entered with the most significant byte of the word first. The first eight bytes form the contents of word zero.
Delay: 10-15
Example:

```
Rom256x24  2B  addr[7-0] enable  Out[23-0]
  0 1 2 3 ... /* 768 bytes total */ ;
```

<u>Rom256x32 Rom256x40 Rom256x48 Rom256x56 Rom256x64 ...</u>
Additional Rom's available--see the Functional Index at the beginning of this section. The numbers of address lines, outputs, and contents vary according to size. Each numeric constant gives one byte's value.

<u>Schmitt-trigger-inverter</u>
This inverter is a thresholding device. It maintains its current output level until the input completes its change to a new logical level. Input changes between a logical level and uncertainty are not reflected as changes in the output. This type of component is commonly used to gain noise immunity on input signal lines. In the simulator

this component "cleans" long uncertainty intervals from signals and restores rapid transitions between logical levels. The use of this component is not encouraged, and usage may require justification -- ask your TA.
Delay: 10-15
Example:
  Schmitt-trigger-inverter 1W Sig Sig';

Scope
There may be a variable number of inputs and these will be displayed as functions of time. The last signal input causes a recording sweep to occur on a transition to ONE. The first integer parameter sets the sampling time, and the second gives the trace length (in screen dots) for each display value.
Example: (With three traces)
  Scope 4x I2 I1 I0 trigger 20 3;

Seven-segment
The segments turn on when their corresponding input is a ONE.

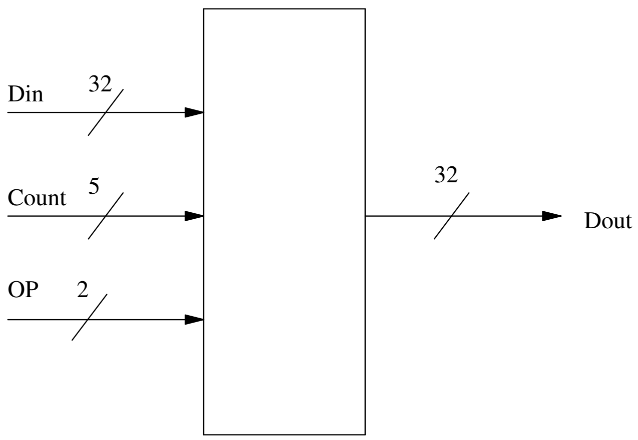
```
  Segment configuration
            a
           ---
   f   |    g    |   b
           ---
   e   |    d    |   c
           ---
```

Example:
  Seven-segment 9w a b c d e f g;

Shift-register
This is a universal synchronous shift register of variable length. Both parallel and serial input and output are provided. The mode-control inputs select the function according to:
 M1,M0: (0,0)-hold; (0,1)-SR; (1,0)-SL; (1,1)-Load.
the mode-selected action occurs on the falling edge of the clock. Reset is active-low and asynchronous.
Delay: 15-22
Example: (With three-bit width)
  Shift-register aB SRin I2 I1 I0 SLin M1 M0 reset clock |
      O2 O1 O0;

Sparc-shifter

The Shifter component has 39 inputs and 32 outputs when used as a Sparc-shifter. Other path widths than 32 bits can be used by adjusting the numbers of input and output lines accordingly. The shift count field must be wide enough to accommodate shifts at least as great as n-1 positions for n-bit wide data paths.



SPARC Shifter

The operation of the Shifter is controlled by the OP code as given in the following table:

Shifter Function Codes	
OP	Operation
00	Nop (In->Out)
01	SLL (Shift left logical--zero fill)
10	SRL (Shift right logical--zero fill)
11	SRA (Shift right arithmetic--sign extend)

Example:
Shift-unit 1b-3c Ain[31-0] Count[4-0] OP[1-0] | Out[31-0];

Sound
(PC version only) The four-bit input selects the pitch of the note played on the micro's speaker. A zero four-bit input selects the pitch of middle C. Successive pitches have a ratio equal to the twelfth-root of two. The rising edge of the gating signal causes the note to play. The enable is low-active.
Example:
Sound 5G I3 I2 I1 I0 Gate Enable;

Standard-out
Nine inputs and no outputs. The eight data bits are written to the standard output file on the leading edge of the clocking signal. The disposition of the output is operating system dependent and there will often be a buffer between the output bit stream and the screen or file (by redirection) where it appears.
Delay: 10-15
Example:
Standard-out 4B data[7-0] clock ;

Switch
Switches have a single output that initially has the value of the given symbolic parameter. The first switch in the source file is assigned to the "0" key of the keyboard, the second to "1", etc. The eleventh through sixteenth to ":;<=>?". There is a limit of 16 switches.
Delay: 10-15
Example:
Switch 3A Sw1 ONE;

Sync
This synchronizing element transfers the data input to the output at the falling edge of the clock. The input may be asynchronous.
Delay: 10-15
Example:
Sync 1b data clk s-data;

T-gate
Transmission gate or switch. Icon shows three inputs. When the middle input is a ONE, the outer two inputs are connected together.
This component probably is a little shaky! Ask the Prof.
Delay: 10-15
Example: (Connects "a" to "c" under control of "b")
T-gate AB a b c;

Tape
Three inputs and one output. The inputs are: Tape-movement-direction, Cell-value-to-be-written, clock. On the rising edge of the clock, the input value is written into the current cell. On the falling edge of the clock, the head is moved, and the newly read value is sent to the output. The tape is 256 bits long, and the initial values are given by 16, 16-bit numeric constants. The 17'th numeric constant specifies the initial head position. Multiple,

coordinated elements can be used to encode the desired symbol set for a Turing machine tape. The addressing is modulo 256, so this element also looks somewhat like a disk. Multiple elements can be used for multiple tracks.
Delay: 10-15
Example:
  Tape AB a b c d /* Tape: */ 0xffff 0x55a5 ... /* Head: */ 127;

Taurus
A single sliding and colliding block.
This component probably is a little shaky! Ask the Prof.
Delay: 10-15
Example:
  Taurus AB a b c;

Time-probe
This component gives a digital readout of the time a given value is reached by the sampled input. The capture must be suitably armed. The single input is the signal to be monitored. The first symbolic parameter is the threshold being monitored for the input signal. The second symbolic parameter is the value of the input signal that will re-arm the capture.
Example:
  Time-probe 2a sigx UNKNOWN ONE;
This example displays the time when sigx comes to the value "UNKNOWN". Re-arming occurs when sigx becomes ONE.

Tri-state-buffer
This component has a variable number of inputs and outputs. There must be exactly one more input than outputs. The inputs are buffered to the output under control of the last input, the control. The control line is active low. When the control line is high, the buffer output will be in the high-impedance state. When the control line is low, the outputs will follow the inputs.
Delay: 20-30 (timing uncertain)
Example:
  Tri-state-buffer 2a Ix Iy Iz control | Ox Oy Oz;

Up-down-counter
This is a positive-edge-triggered, enabled, variable-width, up/down, synchronous, binary counter. The counter counts up when the up/down line is a one. The enable is active-low. Counting occurs on the positive going edge of the clock signal. This counter starts (automatically) in the all-bits-zero state.
Delay: 10-15
Example: (With a width of three bits)
  Up-down-counter 5R up-down clock Enable | O2 O1 O0;

X-stepper
This positioning device has asynchronous control inputs. The rising edge of C0 followed by the rising edge of C1 gives a positive increment to the X-position. The opposite phasing in the changes of C0 and C1 gives a decrease in the position. The current position is given as a four-bit output. Whenever one of the control signals is in transition, the other must be stable. C1 C0 (increasing one step per pair): 00 01 11 10 00 01 ... C1 C0 (decreasing one step per pair): 00 10 11 01 00 10 ... It is permissible to change directions at any time (in any state).
Delay: 20-30
Example:
  X-stepper 7b C1 C0 o3 o2 o1 o0;

Xnor
Exclusive-nor (equivalence) gate. If there are more than two inputs, the function is that of even-parity.
Delay: 15-22
Example: (With three inputs)
  Xnor 5J a b c d;

Xor
Exclusive-or gate. If there are more than two inputs, the function is that of odd-parity.

Delay: 15-22
Example: (With five inputs)
Xor 9Q a b c d e f;

Y-stepper

See the description of the X-stepper.
Delay: 20-30
Example:
Y-stepper 7b c1 c0 o3 o2 o1 o0;