CPS 420--COMPUTER ARCHITECTURE

LABORATORY NOTES

Second Edition--C++ Version

Professor Richard J. Reid

Computer Science Department
Michigan State University
E. Lansing, MI 48824

January 1996

Table of Contents

PREFACE

0. Introduction

The laboratory work in Computer Architecture goes through a series of experiments that build parts of a computer. These parts are then integrated and provide a working digital computer of sufficient power to execute programs generated by standard assemblers and compilers. These notes support this laboratory activity.

A key laboratory element is the digital simulator, *sim,* described in these notes. *Sim* is C++ based, that is, a C++ program is used to describe a model of the computer being simulated. A class and function library is provided to make the expression of this model convenient. C++ is used because it is widely known and used, and because of its convenient extension mechanisms.

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

Sim 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. In your *sim* usage, you probably will not have designs extending much beyond a few hundred components, but even these few components will allow you to implement a functioning computer.

Development of these design tools is continuing. There may never be a low level click, drag and connect tool for the simulator components, however. For very small networks this is an interesting mode in which to build, but in larger designs, such as building complete computers, placing things together in this manner is too detailed and requires too much time.

This simulator has been repaired, modified and extended many times, but alas, will never be bug free. Your TA and/or instructor may be able to confirm your encounter with a bug, and provide you with a work-around and/or deploy the repair crew.

0.1. Recent Changes

Changes in this version of *sim* include:

- Signal vectors are now composed from left-to-right (using the comma operator) for their most-significant to their least-significant elements.
- The Signal member function *sub()* has been added. This supplements the overloaded minus (-) operator and its use is necessary when finding a sub-vector within a composed vector of Signals.
- Previously existing components have been modified:

  Dff, Jkff--the inputs are reordered.

  Mux--the inputs are reordered.

  Ram--the inputs are reordered and the special designation allowing this component to be used as a Register File has been eliminated.

  Iand, Inand, CiOr, CiNor--arguments have been changed to simplify the specification of which inputs are to be complemented.

  Switches, Pulsers, PowerOns and Clocks now have their quiescent value specified by their network-formulation-time connection to the Signals One or Zero.
- RegisterFile--this new (distinct) component has been added. Formerly, it was a special case of a Ram component.
- Source code filenames have been changed to limit them to eight characters in length--to aid in porting to other systems.

Chapter 1: GETTING STARTED WITH SIMULATION

1. Getting Started

Simulation involves working with models. Models are developed which describe the structure and behavior of real and/or abstract entities that exist, or are being contemplated, in some primary domain. The performance of this model is then examined in some secondary domain. In our case here, we will provide a textual description (model) of a digital network using the C++ language. This description will reference a C++ library that is provided, and the resulting executable computer program (secondary domain) *simex* will provide an interactive simulation that can be examined for specific performance properties. For example, if it's a computer model, does it execute instructions correctly?

Time deserves an early (and probably often) mention when dealing with simulation. Invariably, the system being simulated consists of components that are in simultaneous operation. A simulation will have very low fidelity if we model this simultaneity of operation by getting to the computations associated with multiple components *as-soon-as-we-can* depending on the speed of the processor that does the computing associated with the simulation.

To get around the finite time required for the simulation computer to process the actions of each of the components that are actually in simultaneous operation, it is essential that the two times, system model time and the actual computation time, be decoupled. The computer doing the simulation runs in real (wall clock) time, and there is a separate time (non-real) that is kept for the system being simulated.

The system time advances in discrete steps as dictated by the occurrence of events in the system model. Whenever any event occurs in the system (a logical signal changing value, for instance), we freeze the system time at the time of that change until we are able to carry out *all* the computations for components directly affected by that change. After those computations are completed, the system time can be advanced to the next time at which any event occurs. This freeze/compute/advance scheme of time control allows an accurate evaluation of system models.

Before getting to the many details of modeling and simulation, let's work a simple example.

Consider the half-adder shown in Figure 1.1.

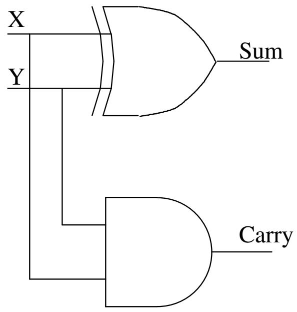

Figure 1.1. Half-Adder Digital Network

This is a network that is commonly developed in an introductory digital logic course and is used frequently throughout computer design. This figure only shows the half-adder as an isolated instance. Any application of the network would always have sources for the excitation signals and destinations for the network outputs. In simulation we will want these inputs and outputs connected in order to examine the network behavior. Figure 1.2 shows the half-adder in a test fixture that will allow validation of the network functionality.

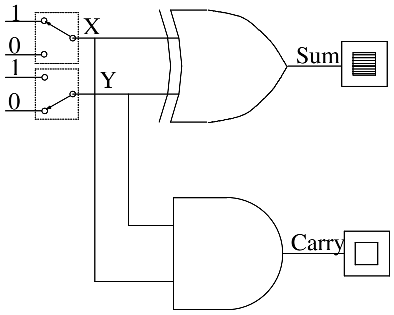

Figure 1.2. Half-Adder in a Test Fixture

Switches are connected to the network inputs to provide excitations of logical ones and zeros. The network outputs are routed to logic-value indicator-probes to allow display of the of the output values. To complete the model of the network for simulation we must further capture and express the graphical schematic information from Figure 1.2.

The following text is a description of the model as a C++ program. This form is called a *netlist.*

```
#include <Sim.h>

void simnet()
{
 Signal X, Y, Sum, Carry;

 Switch ( "1a", X, 'x');   Xor ( "1b", ( X, Y ), Sum );  Probe ( "1c", Sum);

 Switch ( "1a", Y, 'y');   And ( "2b", ( X, Y ), Carry );  Probe ( "2c", Carry);
}
```

Components are instantiated by *calls* to C++ functions of the proper name: Switch, And, Xor, etc. The first argument in the call to a function is the graphical *cell* designation. This two-character ASCII string indicates the component's row and column placement on the schematic. The Switch's are positioned in cell 1a (row "1", column a"), the Xor and And in 1b and 2b respectively and the Probe's in two row positions of the third column, 1c and 2c. The text of the netlist above has been laid out to mimic the placement of the components in Figure 1.2, but the order and placement of the entries in the netlist is generally arbitrary. That is, the netlist:

```
#include <Sim.h>

void simnet()
{
 Signal X, Y, Sum, Carry;

 Probe ( "1c", Sum);
 Probe ( "2c", Carry);

 Switch ( "1a", X, 'x');
 Xor ( "1b", ( Y, X), Sum );

 Switch ( "1a", Y, 'y');
 And ( "2b", ( Y, X), Carry );
}
```

contains the same information and is equivalent. Either netlist above, when processed by *sim* would result in the on-screen display shown in Figure 1.3.

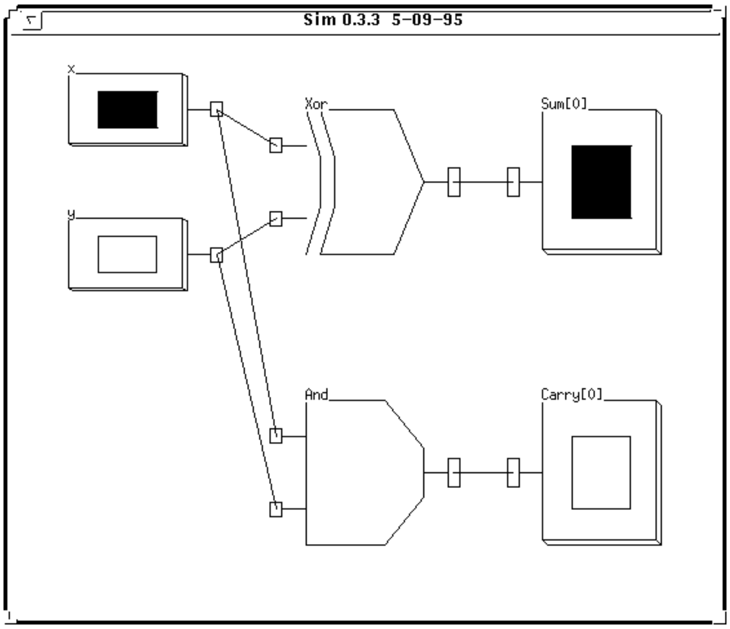

Figure 1.3. Sim Display of Half-Adder Network

The labels in the upper left of the Switch boxes indicate the keyboard key that toggles the Switch values; x and y keyboard keys for the X and Y variables, respectively.

1.1. Basic How To (Do a Simulation)

The simplest steps required to do a simulation are as follows:

1. Extend the network of interest with the addition of the necessary excitation elements: Switches, Pulsers, Clocks, etc.

2. Add the desired instrumentation to display the results: Probes.
3. Organize the network elements along with the added excitation and display elements in the desired schematic relationships.
4. Create a *netlist* as an ordinary C++ function module using a text editor.
5. Use *sim* with the command:

   sim netlist.c

   The naming of the file *netlist.c* is arbitrary, but it must contain the function *simnet().* *Simnet* can call any other functions desired.
6. Execute the resulting program using: *simex*
7. Interact with the network simulation as desired using the keyboard and mouse.

When developing more complicated systems for simulation, a common additional step is the organization of an hierarchy of modules. Modules can be defined in terms of primitive components, or other defined modules, and used in the same way primitive components have been used above. Hierarchical modules are covered in a later section of these notes.

Chapter 2: INTRODUCTION TO SIM

2. Sim

Sim is a collection of classes and global functions that supports the construction and simulation of digital systems.

The overall organization of *sim* is directed nearly exclusively to obtaining a run-time model that executes as rapidly as possible. A secondary goal is to allow the concise and convenient expression of large, hierarchically-organized digital systems. To achieve these goals, the *sim* system of classes and functions is divided into five phases. These phases are:

1. Signal declarations and component interconnection recording. This phase forms a linked list of components and their characteristics from the invocations of the component type functions.
2. Build the runtime-model, component-related, data-structures from the assignments and recordings of Phase 1.

   Convert the linked list formats to contiguous arrays. This conversion is used to provide random access (and its speed) during simulation time.
3. Build the event-driven, simulation-model data-structures.
4. Initialize and run the model.
5. Build the Interface between the running model and the interacting user. Activate the X Window System communications for the display, keyboard and mouse.

The use of #include <Sim.h> in a netlist makes the following interface modules available.

2.1. Sim.h

```
/* Sim.h 7-31-95 */

#ifndef _SIM_H
#define _SIM_H

#include "Signal.h"
#include "Schmtc.h"
#include "Ph1.h"
#include "ssiPh1.h"
#include "msiPh1.h"
#include "lsiPh1.h"
#include "vlsiPh1.h"
#include "sbPh1.h"
#include "lbPh1.h"
#include "Composed.h"
#include "AuxFctns.h"
#include "addStCmp.h"

extern Signal Zero, One;
extern int userRunTimeFlag[];
extern int numberOfUserRunTimeFlags;
#endif /* ifndef _SIM_H */
```

Chapter 3: SIM SCHEMATIC DESCRIPTORS

3. Schematics

In the general format of component specification:

<Component Type> (<Schematic Descriptor> , <Input(s)> , <Output(s)>) ;

the Schematic Descriptor (SD) entry positions the component on a two-dimensional grid in the simulation-time display-window. In their simplest form, descriptors are given as two-character, quoted ASCII strings. This two character sequence specifies the row and column coordinates of a virtual grid overlaying the display window. For example, "1a", "1b", "2c", specify row 1 column a, row 1 column b, and row 2 column c. The grid is virtual in the sense that the maximum extent of placements occurring throughout all the SD's encountered in a system description are mapped to the full extent of the window of the display. Row and column extents in a window 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 could be used for each.

Screen cell positions are composed using the syntax:

```
cell = {C}{C}
     | {C}{C}-{C}{C}
     | cell.cell
```

where {C} is defined to be any single ASCII character, but, for simplicity, usually restricted to be one of:

```
{C} = [a-zA-Z0-9]
```

The cell-designator must be quoted as a C++ string. In addition to a cell-designator being given as a two-character sequence, denoting a window row and column for placement of the component, an extent sequence, such as "1a-3b", can be given for a component occupying an extended screen region.

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 window cell by giving the identical cell designator for each component. For example:

```
Switch ( "1a", x, 'x' );  Switch ( "1a", y, 'y' );
```

In these cases, the components are stacked vertically within the cell, their order of occurrence gives a top-to-bottom screen-presentation order.

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

```
Switch ( "1a.ab", x, 'x' );  Switch ( "1a.ac", y, 'y' );
```

If cell designators are not identical ASCII strings, they can overlay one another, and will, if they actually refer to the same space. For example:

```
Register ( "Mp-Mp", regIn, regOut );
Probe ( "Mp", regOut[3] ); Probe ( "Mp", regOut[1] );
Probe ( "Mp", regOut[2] ); Probe ( "Mp", regOut[0] );
```

places four vertically-stacked probes overlaying the register.

Be cautious in developing complex graphical placements, such as: stacked and overlaid sub-spaces within hierarchically-placed components. It's often advisable to proceed incrementally in developing complicated placements. In actuality, it's rare that one needs or wants placement control beyond the simplest two-character designation.

3.1. Hierarchy

For simulation models including more than a few gates, it is desirable to be able to form hierarchical modules that may be used directly as model components.

As an example consider the following HalfAdder module that is defined in terms of primitive Xor and And gates.

```
void HalfAdder ( const SD & sd, const Signals & in, const Signals & out )
{
 Module ( sd, in, out );              // Module display at current schematic level
   Xor ( SD ( sd, "1a" ), in, out[1] ); // Nested <Schematic Descriptor>
   And ( SD ( sd, "2a" ), in, out[0] ); // Nested <Schematic Descriptor>
}
```

The above definition allows the subsequent use of the module in a manner similar to the use of primitive components, such as:

```
HalfAdder ( "1b", ( x, y ), ( carry, sum ) ); // Module usage
```

3.2. Schmtc.h

```
/* Schematic.h  7-31-95 */

#ifndef _SCHEMATIC_H
#define _SCHEMATIC_H

class SchematicDescriptor { public:

 int graphicLevel;
 char * graphicPosition, * decal;

 ~SchematicDescriptor();

 SchematicDescriptor();

 SchematicDescriptor ( const SchematicDescriptor & );

 SchematicDescriptor( char * graphicPos, char * decal = 0 );

 SchematicDescriptor( const SchematicDescriptor &, char * graphicPos, char * decal = 0 );

 SchematicDescriptor operator , ( char * decal ) const;
};

typedef SchematicDescriptor SD;

#endif /* ifndef  _SCHEMATIC_H */
```

Chapter 4: SIGNALS IN SIM

4. Signals

Signals are defined types used to make connections, and to provide simulation-time communication-values between components. Signals are defined during the network construction phase using C++ type declarations.

All Signal declarations yield vectors. The default vector length is one. A simple declaration such as:

```
Signal x;
```

defines the Signal x, and gives it (via the action of the constructor for Signal), as a side effect of the declaration, a global, simulation-runtime, variable index. It is this index value, that each run-time, evaluation-component receives indirectly through its argument. The evaluation-component uses this index value to access the corresponding run-time variable that will be set (for output connections) or received (for input connections).

Other forms of declarations are also defined, including:

```
Signal w(4);        // A vector of signals with 4 components.
Signal x(3, "x");   // A 3 element vector, with the name "x" assigned.
```

This second form, with the signal name assigned so that it will carry through to simulation time, is quite desirable, because it allows schematic-display labeling of a signal line with a signal's name. A macro Sig is defined to make it more convenient to supply variable names that will be carried forward to execution time.

```
#define Sig(x,n) Signal x(n,#x)
```

As an alternative to the above declaration of x, use:

```
Sig(x,3)     // A 3 element vector, x, with the name "x" assigned.
```

The second argument for a Signal constructor can alternatively be an integer. In this pattern the Signal is a constant, and the second argument is used for initialization. For example:

```
Signal m(6, 0x26);    //m[5] = One, m[4] = Zero, m[3]=Zero, m[2] = One, m[1] = One, m[0] = Zero.
```

The elements of the vector m read left-to-right for most to least-significant element, the same as the bits in the initialization constant.

Declared Signals can be composed as busses for connections to component inputs and/or outputs. Assume we have the declaration of the Signal vector "s" as:

```
Sig(s,5);
```

To support the different situations that arise in gathering Signals together from several sources, the following can be used:

s[2]	// Makes a signal composed only of element #2 of s	s[2]
s[2],s[4]	// Makes a 2 component signal from s	(s[2],s[4])
s[2],s[4],s[1]	// Makes a 3 component signal from s	(s[2],s[4],s[1])
s[2]-s[4]	// Makes a 3 component signal from s	(s[2],s[3],s[4])
s[4]-s[2]	// Makes a 3 component signal from s	(s[4],s[3],s[2])
3*s[0],s[1]-s[2]	// Makes a 5 component signal	(s[0],s[0],s[0],s[1],s[2])

making use of the overloaded comma (,), minus (-), and star (*) operators of the class Signal. These gathering mechanisms can be used in any elaboration for initializing a Signal variable that is to be composed from multiple sources, as, for example:

```
Sig(f,3); Sig(g,4);                   // A 3 and a 4 component signal are created
Signal h = ( f, g[0], g[3]-g[2] );    // Establishes h, which has
                                      // 6 components total, (f[2],f[1],f[0],g[0],g[3],g[2]),
                                      // but no new, simulation-time
                                      // variables are created; this is a new collection of
                                      // variables already in existence.
```

There is *no assignment* from one Signal to another; initialization is what should be used, usually. Be careful not to confuse initialization with assignment. Initialization is part of an object's construction, where the raw bits allocated are being filled in as part of the object's first breaths. Assignment is from one existing object to another existing object, and does the usual member-wise copy from the right-hand object to the left-hand object--probably overwriting and destroying information already in the left-hand object. The default assignment operator "=", which usually exists, has been overridden to yield a diagnostic, since it may have no legitimate usage.

4.1. Connection Ordering

Many primitive components such as And, Or and other gates have no functionally-significant ordering to their connections. However, when the components are rendered schematically, their interconnections have specific positional attachments. For example, assuming all the signals are scalars, the component specification

```
And ("1a", (a,b,c), f);
```

would be functionally equivalent for any permutation of its three inputs. As long as the same inputs were used somewhere, the output, f, would be the same. The graphical rendition of this gate and its interconnections would be different for each input permutation, however. The input connections to components have the following association:

> As the component connection listing goes from left-to-right, the schematic connections are made from the top to the bottom. The above And component specification corresponds to the drawing in Figure 4.1.

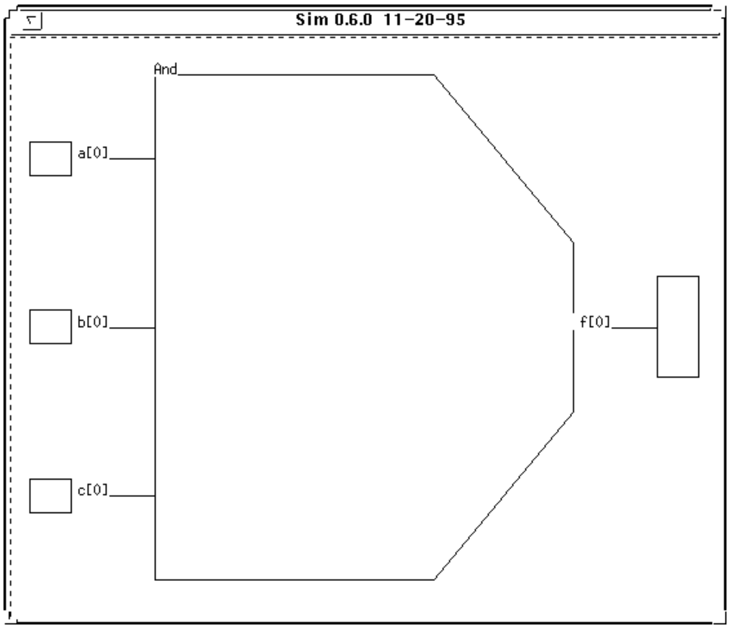

Figure 4.1. Input Connection Ordering

Similarly, for multiple-output components, outputs are drawn on the right of the component going from the top to the bottom as the output connection list is read from left-to-right.

Some components, a Counter, for example, has a notion of a most-significant bit and least-significant bit. Whenever such an ordering is inherent for a component type, *sim* will always have the most-to-least significant bit ordering correspond to left-to-right ordering in the netlist connections, and to top-to-bottom ordering in the schematic rendition.

For some components there can be nested orderings. Consider a multiplexer that has four ports, with each port five-bits wide. A specification for such a multiplexer could be:

```
Signal P0(5), P1(5), P2(5), P3(5), muxAddr(2), Out(5);

Mux ("1a", muxAddr, (P3, P2, P1, P0), Out );
```

Here, there is a further ordering, not just the ordering within the ports from most- to least-significant bit positions, but an ordering among the ports themselves from most to least-significant, and a particular port is selected by the Mux address lines' values in this order.

4.2. Signal.h

```
/* Signal.h  9-7-95 */

#ifndef _SIGNAL_H
#define _SIGNAL_H

enum SignalValue { ZERO, UNINITIALIZED, XXX, HIZ, ONE };

#define Sig(x,n) Signal x(n,#x)

void saySignalValue ( const SignalValue sv );

class Signal { public:

 int length;
 int * serialNumber;
 int constSignal;
 char * nameTag;

 ~Signal();

 Signal( Signal & );

 Signal(unsigned int nr = 1, char * name = "UnNamed");

 Signal(unsigned int nr, unsigned int setMask, char * name = "UnNamed");

 Signal(unsigned int nr, SignalValue initValue,
         int constValue, char * name = "Anon");

 Signal operator = (const Signal & sig) const;

 Signal operator[](int n) const;

 Signal operator - (const Signal & sig) const;

 Signal sub ( int start, int end ) const;

 Signal operator , (const Signal & sig) const;

 void print() const;
};
```

```
typedef Signal Signals;

Signal operator * ( const int reps, const Signal & sig);

class SignalNode { public:
  void print();
  SignalNode * ptr; int signalSerialNumber; char * name; int constSignal;
  SignalValue value;
};

class SignalList { public:
 int numberOfSignals;
 SignalNode * start;
 SignalNode * end;
 SignalList();
 ~SignalList();
 void add ( int serialNumber, SignalValue value, const char * name, int constSignal );
 void print();
};

#endif /* _SIGNAL_H */
```

Chapter 5: HIERARCHICAL MODULES

In building simulation models for digital systems that include more than just a few gates, it is desirable to be able to form hierarchical modules that may be used directly as model components. As a simple example, Figure 5.1 shows a hierarchy with four levels.

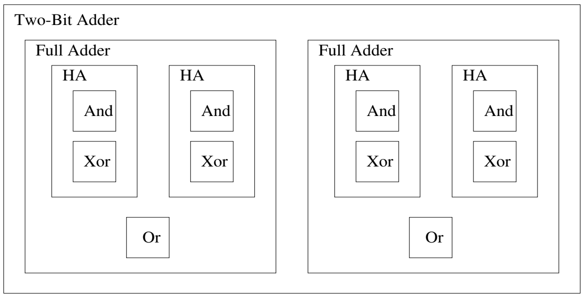

Figure 5.1. Hierarchical Component Organization

Here, we'll just look at two of the levels to illustrate how modules can be formed. Figure 5.2 shows the gate-level implementation of a half-adder,

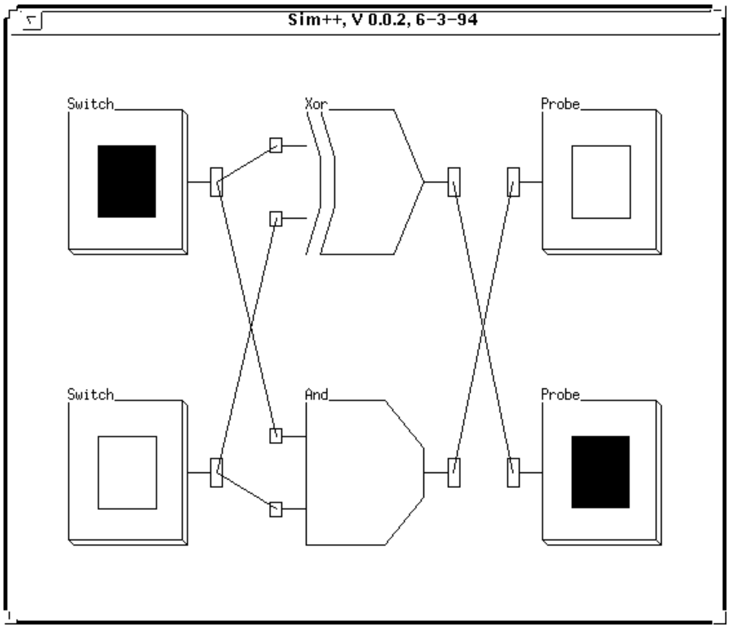

Figure 5.2. Half Adder Internal Gates

and Figure 5.3

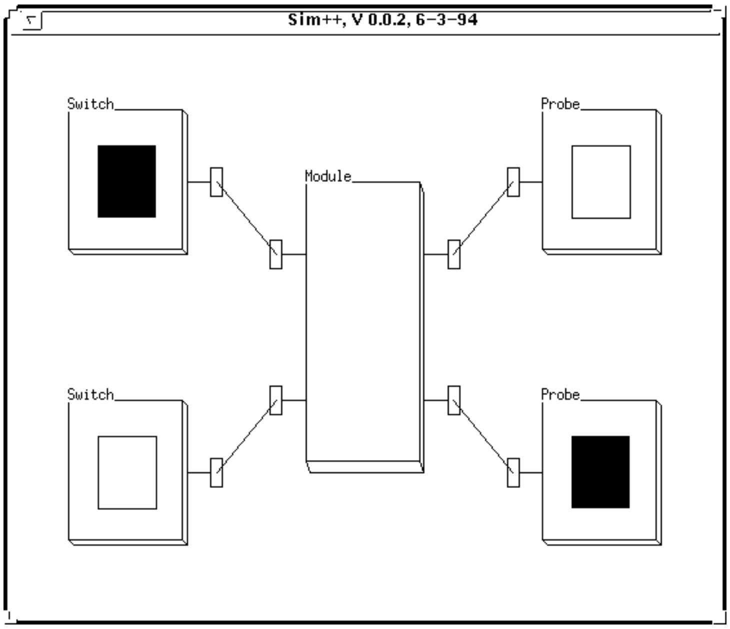

Figure 5.3. Half Adder Module

shows the modularized form of the half adder. This module can be used as a primitive component as illustrated by the following program.

```
#include <Sim.h>

void
HalfAdder(const SD & sd, const Signals & in,
          const Signals & out)
{
 Module ( sd, in, out ); // Module display at current schematic level, sd

   Xor ( SD ( sd, "1a" ), in, out[1] ); // Nested <Schematic Descriptor>

   And ( SD ( sd, "2a" ), in, out[0] ); // Nested <Schematic Descriptor>
}

int simnet ()
{
  Signal x, y, carry, sum;
  Switch ( "1a", x, 'x' );
  Switch ( "1a", y, 'y' );

  // Module usage
  HalfAdder ( "1b", ( x, y ), ( carry, sum ) );

  Probe  ( "1c", carry ); Probe  ( "1c", sum );
}
```

Chapter 6: SIM COMMAND LINE

An executable for sim can be made using the command:

> sim simnet.c

The executable that is produced, *simex,* can be invoked with the following options:

If the current directory has a file named *simrc,* the command line arguments will be taken from that file.

Chapter 7: SIM INTERACTIVE CONTROL

Mouse Button Assignments

(for zooming in on schematics)

Adjust Zoom Box	Zoom In	Reset to Full Zoom-Out

New Windowing Mode (default)

Mark Upper Left	Full Zoom Out	Lower Right & Zoom-In

Old Windowing Mode

The critical regions for mouse capture of the zoom-box adjustments in the new windowing mode are shown below.

```
- - - - - - - - - - - - - - - - - - - - - -
|  2 |          3           |  4 |
| - -                          - - |
|                                  |
|  5            1              6   |
|                                  |
| - -                          - - |
|  7 |          8           |  9 |
- - - - - - - - - - - - - - - - - - - - - -
```

Region(s)	Action
1	Drag
2, 4, 7, 9	Expand/Shrink at Corner
3, 5, 6, 8	Expand/Shrink at Edge

Keypad Assignments

(simulation time control & modularization display control)

7 (H) Home Greatest Modularization	8 (1) ↑ Full Speed	9 (U) PgUp Greater Modularization
4 (0) ← Slower	5 ( ) Redraw	6 (2) → Faster
1 (E) End Least Modularization	2 (3) ↓ Stop/ Continue	3 (D) PgDn Less Modularization

The parenthesized entries give the characters needed for activation of these operations from command-line arguments.

Function Key Assignments

F1 Reset Simulation to Time Zero (restart)

F2 Toggle Interconnecting Lines Drawing

F3 Toggle Component Labels Drawing

F4 Toggle Connection Tabs Drawing

F5 Toggle Interconnecting Lines Labeling

F6 Toggle Time, QueueLength, Slowness Display

F7 Toggle Interconnecting Lines for Mux'es Only

F8 Toggle SuperProbe Display

F9 Toggle Windowing Mode

F10 Exit

Chapter 8: SIM IMPLEMENTATION

8.

The simulator is implemented as a set of class definitions and global functions that are written in C++. There are about 8,500 lines of C++ code that make up the simulator.

The simulator uses the X-Window System, XLib library to support the run-time graphics. No higher-level windowing support is used; no XView nor OpenLook nor Motif. This restriction to low-level graphics usage is intended to increase the portability of the simulator.

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 such things as components and signals. 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.

8.1. 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 *SignalBlock* and *LogicBlock* components let 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).

8.2. Source Availability

The source code for *sim* is available. The porting to other UNIX systems should only require the rationalization of the library files appearing in the *makefile.*

Porting to non-UNIX systems will require changing from the X Window System interface to whatever the graphical user interface is for the target system.

8.3. Bugs

This is a list of known bugs and limitations in *sim.*

1. SuperProbes can only display in connectors that are drawn during the first display. Use a command-line option to get to the lower graphical level *before* the first display.
2. Probes start/stop displaying immediately after the graphical level is changed. That is, before any redraw (5) actually shows/removes the Probes themselves.
3. You can't have reliable edge triggering of devices at time = 0; move those edges later in time, after the original devices' transients.

Chapter 9: SIMULATION RUN-TIME MODEL

During network formulation, linked-lists of Signals and Components record the results of Signal declarations and Component function-evaluations. After all the network has been established by the sub-tree of functions called starting with simnet(), all Signals and all Components being used are known.

At this point, random-access data structures, arrays, are dynamically allocated and replace the linked-list forms. This revised organization of network information is used to allow rapid processing of randomly ordered visits to different Signals and Component evaluation-functions.

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

The random-access data structures support execution of the simulation model shown below.

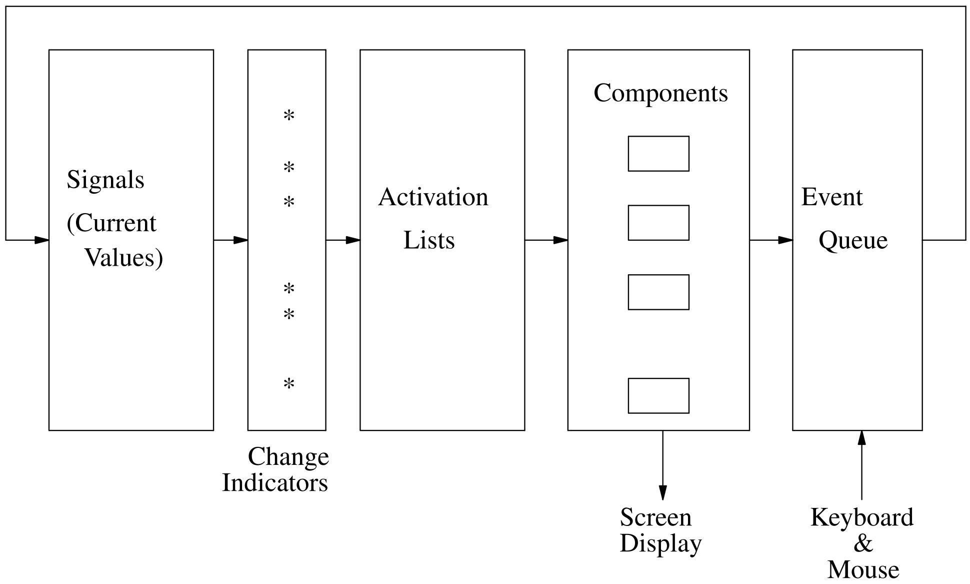

Figure 9.1 Simulation System Model

The design features of this model are:

1. The variable values must maintain their proper time relations, there can be absolutely no time-skew of the signal changes with respect to their timing in the system model. This is an absolute requirement.
2. Consult components as infrequently as possible for evaluation of their outputs based on their current inputs.
3. Limit or prevent iterative searches for any signal value or component.
4. Time order future events, variable-value changes, in the Event Queue in an efficient manner to prevent extensive searches for event placement or retrieval.
5. Prevent unnecessary Event Queue entries that merely repeat the currently scheduled or present values of variables.

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.

The model flows as follows. The variables are updated to their values for the current time by extracting all the "due" events from the Event 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 Event Queue, time-stamped with the future time, based on device delays, when they are to become effective.

There is some adjudication necessary in bringing new variable values from the Event 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. Accommodation of high-impedance Signals is not provided in the current version of *sim.*

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.

Event driven simulation entails a large amount of queue activity. For this reason, the implementation of the queue is extremely important. *Sim* currently uses the Skew-Heap algorithm for the implementation of the event queue. The next event is at the top of the heap, and new entries are made by descending from the top of the heap, down through the binary tree, to the position where the time-label prescribes. The Skew-Heap algorithm also has some features directed to keeping the tree *balanced.* This implementation gives Order (log n), where there are n items in the queue, performance.

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 value of variables during the time between these limits is taken to be indeterminate whenever a logical change of the variable's value 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 or mouse actions. Logic stimulations include changing Switch values and activating Pulsers. Display control includes: zooming-in/out, changing the level of module-hierarchy display, providing/inhibiting component and signal labeling, and changing window size, shape and placement. Other interactions include: resetting, slowing, speeding-up, halting or exiting the simulation, and omitting/including the lines of component interconnection.

Chapter 10: COMBINATIONAL-NETWORK DESIGN TOOLS

10. 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 OpenWindows 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.*

10.1. 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.

10.2. 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.

10.3. 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.

10.3.1. 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

10.3.2. 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, OFF and DC 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, OFF and DC 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. A warning about any overlaps between partitions will be given on the standard output. Note, that if the ON and OFF partitions are interchanged one can easily evaluate the complement function and/or take that solution to be products-of-sums form.

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.

10.3.3. 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.

10.3.4. 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.

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

1) Read the input. Compress the given arrays using consensus.
2) Process the input specifications.
    - ON, OFF and DC--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 of 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.

10.3.6. Troubles

If you encounter problems using cnd, you may want to try the following:

- 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.

10.4. 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
ˆ	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.

10.5. 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.

10.5.1. 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.

10.5.2. 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.

10.6. 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 and net can be piped together as:

```
rd <arb_Bool_exp | net -t
```

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

Chapter 11: SIM COMPONENTS

11. Components

Adder	**Decoder**	**LogicBlock**	**PowerOn**	**SignalBlock**
Alu	**Dff**	**Module**	**Probe**	**Space**
And	**Encoder**	**MultiLevelCompare**	**ProbeH**	**Stderr**
CiNor	**Gate**	**Mux**	**Pulser**	**Stop**
CiOr	**Iand**	**Nand**	**Ram**	**Switch**
Clock	**Inand**	**Nor**	**Register**	**Xnor**
Compare	**Increment**	**Not**	**RegisterFile**	**Xor**
Counter	**Jkff**	**Or**	**Rom**	**loadRom**

11.1. Heading Files

```
void Adder ( const SD &, const Signals & in, const Signal & out);
//Adder ( "1b", ( aPortIn, bPortIn ), sumOut );

void Alu ( const SD & sd, const Signals & dataIn, const Signals & out );
//Combinational device--for MIPS
//Alu ( "1b", ( aIn, bIn, cIn, op ), ( ccOut, dOut ) );
//Widths:       N  N   1   3       4    N
//op -- operation: and=000; or=001; add=010; sub=110;
//              set-on-less-than=111  -- not available yet
//    the following may be changed later (MIPS):
//              xor=011; nand=101; nor=100;
//     not all of the following outputs are necessary for MIPS:
//ccOut -- condition codes (negative,zero,overflow,carryOut)
//                          ^                   ^
//                         MSB                 LSB

void And ( const SD & sd, const Signals & in, const Signals & out );
//Simple single And for a single out.
//A vector of And's if out is a vector, then in.length = N * out.length
// and each And has N inputs.
void And (const SD & sd,
          const Signals & in1, const Signals & in2,
          const Signals & out );
//A vector of out.length And's:
//if in1 & in2 are the same length: each And has two inputs, one from
//       in1 and one from in2.
//in1.length>1 && in2.length==1:the single in2 is an input for
//       each of the Ands.
//in1.length==1 && in2.length>1:the single in1 is an input for
//       each of the Ands.

void CiNor ( const SD &, const Signals & true,
               const Signals & complemented, const Signal & out);
//Complemented-input Nor

void CiOr ( const SD &, const Signals & true,
               const Signals & complemented, const Signal & out);
//Complemented-input Or
```

```
void Clock (const SD &, const Signal &, const int duration,
            const int offset, const int period,
            const Signal & init = Zero );
//duration is ON time, offset is displacement of first ON from t=0,
//period is the time until the next repetition.
//init is the initial output--ON is taken to be the complement of this value.

void Compare ( const SD & sd, const Signals & x, const Signals & y,
              const Signal & match );
//Compare two Signal vectors, x and y, for agreement of their
//binary patterns.  The output, match, is a ONE for agreement.
//This is a composite component, so you can see inside it.

void Counter ( const SD &, const Signals & in, const Signals & out );
//Counter ( "1b", ( reset, clock ), out );

void Decoder ( const SD &, const Signals & enable, const Signals & data,
           const Signals & out );
//N data-in lines are decoded to 2^N out lines
//single active-high enable

void Dff ( const SD &, const Signals & in, const Signals & out );
//in: (set,Din,clock,reset)
//out: Q

void Encoder ( const SD &, const Signals & in,
              const Signals & out, const Signal & valid );
//2^N data-in lines are examined for a ONE
//the most-significant position where there is a ONE is
//encoded as the N out lines' values.
//valid is ONE if there is at least one ONE.

void Gate ( const SD &, const Signals & in, const Signals & out );
//Complete clock pulses are gated to the output.
//Enable must be ONE before the clock starts.
//The clock (gated to the output) will continue for its full
//duration, regardless of what Enable does later.
//in: (enable,clock)
//out: (gatedClock)

void Iand ( const SD &, const Signals & true,
                const Signals & complemented, const Signal & out);
//Inhibit-and

void Inand ( const SD &, const Signals & true,
                const Signals & complemented, const Signal & out);
//Inhibit-nand

void Increment ( const SD &, const Signals & in, const Signals & out );
//Combinational unit. There is no carry out.
```

```
void Jkff ( const SD &, const Signals & in, const Signals & out );
//in: (set,J,clock,K,reset)
//out: (Q,Qprime)

void LogicBlock ( const SD &,
                  LogicBlockEvaluatePointer,
                  const Signals &, const Signals & );
// The second argument is the name of the user-supplied
// function that is to be called whenever any of its
// inputs change.  This user-function computes the outputs
// and returns them in logicOut[].

void Module ( const SD &, const Signals &, const Signals & );

void MultiLevelCompare ( const SD &, const Signals &, const Signal & );
//Compares two inputs for equality, over ALL the
//Signal values: ZERO, UNINITIALIZED, XXX, HIZ, ONE
//Output is ONE for equal, ZERO otherwise.

void Mux ( const SD &, const Signals & control, const Signals & data,
           const Signals & out );
//out.length defines the width of the input ports and the output port.
//data.length/out.length -> numberOfPorts
//control is the port selection
//control.length must be log2(numberOfPorts)
//data[port,bit]order: [P-1,n-1],[P-1,n-2],...[P-1,0],...[0,1],[0,0]
//                     ^top Port,msb                  ^ bottom Port

void Nand ( const SD &, const Signals & in, const Signal & out );

void Nor ( const SD &, const Signals & in, const Signal & out );

void Not ( const SD & sd, const Signals & in, const Signals & out );
//Single inverter or vector of inverters if out.length > 1.

void Or ( const SD & sd, const Signals & in, const Signals & out );
//Simple single Or for a single out.
//A vector of Or's if out is a vector, then in.length = N * out.length
// and each Or has N inputs.
void Or (const SD & sd,
          const Signals & in1, const Signals & in2,
          const Signals & out );
//A vector of out.length Or's:
//if in1 & in2 are the same length: each Or has two inputs, one from
//       in1 and one from in2.
//in1.length>1 && in2.length==1:the single in2 is an input for
//       each of the Ors.
//in1.length==1 && in2.length>1:the single in1 is an input for
//       each of the Ors.
```

```
void PowerOn ( const SD & sd, const Signal & out, const int time,
               const Signal & init = Zero );
//time is when transition will occur.
//init is the initial output--ON is taken to be the complement of this value.

void Probe ( const SD &, const Signals &, const int numberOfPartitions = 0,
          const int * partitionSize = 0 );
//Places a space after each partition to ease the reading of field separations.
//Last partition size repeated, if necessary.
//ZERO->backGroundColor
//ONE->foreGroundColor
//UNINITIALIZED->up-and-to-the-right line
//HIZ->horizontal line
//XXX->crossed lines -- signal in transition

void ProbeH ( const SD &, const Signals & );
//Display arranged horizontally--uses Probe, so labeling is compounded.

void Pulser ( const SD &, const Signal &, const char, const int duration,
               const Signal & init = Zero );
//char is keyboard association, duration is the pulse length.
//init is the initial output--ON is taken to be the complement of this value.

void Ram ( const SD & sd,
    const Signals & controlIn,
      // (enablePort_N-1, writePort_N-1, writePort_N-1_Address, ...
      // ...
      // enablePort_0, writePort_0, writePort_0_Address,
      // readPort_N-1_Address, ... , readPort_0_Address)
      //
      // a port is only written if it's enabled
    const Signals & dataIn,//(dataInWritePort_N-1,...,dataInWritePort_0)
    const Signals & out, // (dataOutReadPort_N-1,...,dataOutReadPort_0)
    const int numberOfWords,
    const int WordWidth, //bits
    const int portAddressSize, //bits
    const int numberOfReadPorts,
    const int numberOfWritePorts
    );

void Register ( const SD & sd, const Signals & enable,
                  const Signal & write, const Signals & dataIn,
              const Signals & out);
//Example:  Register ( "2c", enable, write, in, out );
//Loading occurs on the leading edge of the "write" signal, but
//only if the "enable" signal is ONE at this time.

void RegisterFile ( const SD & sd,
    const Signals & controlIn,
      // (enablePort_N-1, writePort_N-1, writePort_N-1_Address, ...
```

```
    // ...
    // enablePort_0, writePort_0, writePort_0_Address,
    // readPort_N-1_Address, ... , readPort_0_Address)
    //
    // A port is only written if it's enabled (enable == One)
    // Register zero (0) will always read as zero.
   const Signals & dataIn,//(dataInWritePort_N-1,...,dataInWritePort_0)
   const Signals & out, // (dataOutReadPort_N-1,...,dataOutReadPort_0)
   const int numberOfWords,
   const int WordWidth, //bits
   const int portAddressSize, //bits
   const int numberOfReadPorts,
   const int numberOfWritePorts
   );

void Rom ( const SD & sd, const Signals & in, const Signals & out,
      const int words, const int bitsPerWord,
       const unsigned char * romContents );
//unsigned char romContents[] = { 0x01, 0x02, ... };
//Rom ( "1b", RomAddress, RomOut, 16, 12, romContents );
//                    16 words x 12 bits per word
//romContents is by bytes.  Bytes are grouped from left to right until
//there are enough (maybe surplus) to fill a word.  The grouped bytes
//are right justified in the word.
//ONE's in the surplus sections of words (grouped bytes) are immaterial.
//For 16x12, two bytes are needed per word (for 12 bits, with surplus),
//for 32 bytes total.

void SignalBlock ( const SD &,
          SignalBlockEvaluatePointer,
            const Signals &, const Signals & );
// The second argument is the name of the user-supplied
// function that is to be called whenever any of its
// inputs change.  This user-function computes the outputs
// and returns them in signalOut[].

void Space ( const SD & );

void Stderr ( const SD & sd, const Signals & in );
//Stderr ( "1a", (enable,clock,data) );
//writes 8-bit data to stderr on the rising edge of the clock,
//but only if enabled.
//Control characters written as: ^A^B... etc. except
//for 0x0A (^J), which writes the newline character itself on stderr.
//bit[7] (MSB) == 1 writes "|" prefix.

void Stop ( const SD & sd, const Signal & input );
//when the input Signal comes on, sim will exit

void Switch ( const SD &, const Signal &, const char,
```

```
        const Signal & init = Zero );
//char is keyboard association.
//init is the initial output--ON is taken to be the complement of this value.

void Xnor ( const SD &, const Signals &, const Signal & );

void Xor ( const SD &, const Signals &, const Signal & );

void loadRom ( unsigned char * romContents, int size,
        char * filename = "romfile" );
//fill the unsigned char array, up to "size" bytes with
//hex bytes from "romfile"
//"romfile" is a series of hex bytes: 0xA0 0x9C ...
//whitespace is immaterial, but comma separators are NOT allowed.
```