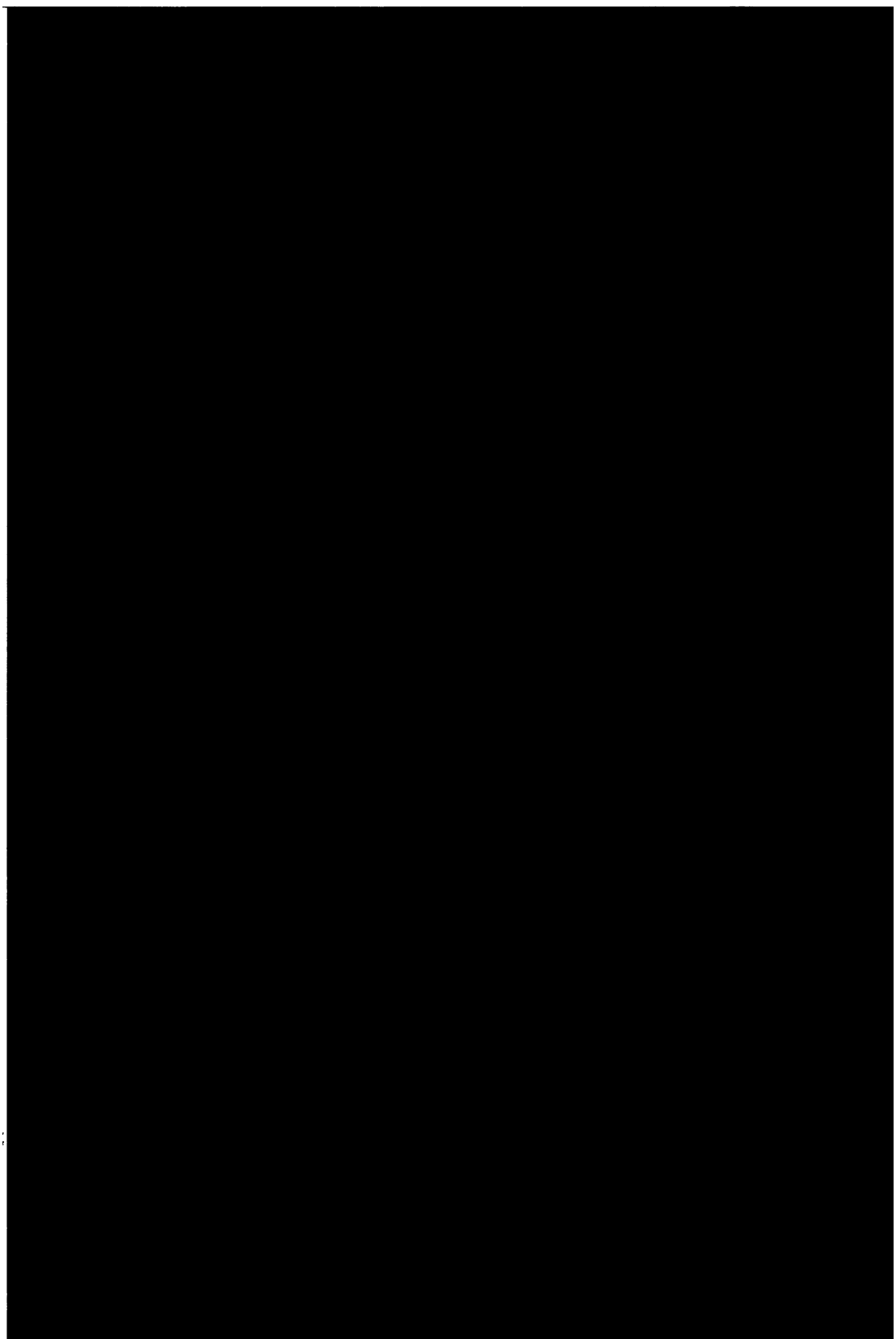


This blank page was inserted to preserve pagination.

MODELS AND DATA STRUCTURES FOR DIGITAL LOGIC SIMULATION

by

DONALD LEIGH SMITH

B.S., University of Kansas City
1962

S.B., Massachusetts Institute of Technology
1962

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1966

Signature of Author..... Donald L. Smith .....
Department of Electrical Engineering
May 20, 1966

Certified by..... Jack B. Dennis .....
Thesis Supervisor

Accepted by..... Thomas S. Huang .....
Chairman, Departmental Committee
on Graduate Students

MODELS AND DATA STRUCTURES FOR DIGITAL LOGIC SIMULATION

by

DONALD LEIGH SMITH

Submitted to the Department of Electrical Engineering on May 20, 1966 in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

A digital logic simulation system is proposed for design verification. Logic to be simulated is specified with a high-level register transfer design language, and the simulation system operates on-line on a large time-shared computer. The problem of selecting adequate circuit and signal models for this purpose is considered. Models are proposed with sufficient timing detail to allow the simulation system to detect timing errors which currently are found by manual checking or prototype debugging.

A data structure for representing idealized circuit and signal models and a matching simulation algorithm is discussed. The data structure is a direct representation of a complete subset of the design language and is organized so that it can be incrementally modified to reflect design changes. The simulation algorithm is very efficient because combinational levels are re-evaluated only if their values are needed and may have changed since last evaluated.

The data structure is expanded to represent detailed circuit and signal models. A method of intermixing idealized and detailed models and efficiently simulating very large designs is discussed. Extensions are proposed to the design language so that it can be used to specify model parameters and serve as the simulation command language.

Thesis Supervisor: Jack B. Dennis
Title: Associate Professor of Electrical Engineering

ACKNOWLEDGMENTS

The author would like to express his gratitude to Professor Jack B. Dennis, whose design language work directed the author's attention to this subject area, and whose advice and suggestions as thesis supervisor have been very helpful. He is also grateful to Professor Eric G. Manning and Messrs. Fred L. Luconi, Nathan R. Melborn and Ashok Malhotra, whose interest in this work has been most encouraging. He would like to express his appreciation to Norma L. Burns, who typed the manuscript, and Irene H. Ziemba, who assisted with the figures. Special thanks are expressed to The MITRE Corporation of Bedford, Massachusetts, who generously financed this work.

TABLE OF CONTENTS

SECTION		PAGE
	ABSTRACT	2
	ACKNOWLEDGMENTS	3
	TABLE OF CONTENTS	4
I	INTRODUCTION	7
II	DESIGN LANGUAGE	11
	Figure 2-1 Four Bit Counter	12
III	CIRCUIT AND SIGNAL MODELS	18
	A Circuit Delay	19
	B Level Circuits	20
	Figure 3-1 Level Signal Values	22
	Figure 3-2 Level Delay Line Model	22
	Figure 3-3 AND Gate	23
	Figure 3-4 Combinational Logic Block Model	23
	Figure 3-5 AND and OR Gate Hazard Values	25
	Figure 3-6 Output Hazard for 2-Input AND Gates	26
	Figure 3-7 Example of AND Gate Model Behavior	28
	Figure 3-8 AND or OR Gate Settling Times	30
	Figure 3-9 Flip-Flop Model	32
	C Register Transfers	32
	D Control Events	33
	Figure 3-10 Example of Signal Spread Fault	35
IV	DATA STRUCTURE FOR IDEALIZED MODELS	37
	A Level Logic	40
	Figure 4-1 Combinational Level String Data Elements	42
	Figure 4-2 Delay and Flip-Flop Level String Data Elements	46
	Figure 4-3 Constant Level and Fixed Memory Data Elements	47

SECTION			PAGE
IV	B	Control Logic	48
		Figure 4-4 Control Logic Data Elements	49
		Figure 4-5 Transfer Effect Table	50
	C	Time Queuing and Miscellaneous Lists	51
		Figure 4-6 Time Queuing Data Structure	52
	D	Simulation Algorithm	53
	E	Discussion of Idealized Model Simulation	57
		Figure 4-7 Unstable Circuit	58
V		DATA STRUCTURE FOR DETAILED MODELS	60
	A	Level Logic	60
		Figure 5-1 Detailed Flip-Flop Level String Data Elements	61
		Figure 5-2 Detailed Combinational Level String Data Elements	62
	B	Control Logic	65
		Figure 5-3 Detailed Control Logic Data Elements	66
	C	Time Queuing Data Structure	67
		Figure 5-4 Time Queuing Data Structure	68
	D	Intermixing Ideal and Detailed Models	70
	E	Data Structure Partitioning	71
	F	Summary of Data Structure Characteristics	75
VI		SIMULATION COMMAND LANGUAGE	76
	A	Design Language Extensions	77
	B	Logic Testing Procedures	88
VII		CONCLUSIONS	90

SECTION		PAGE
APPENDIX A	INTERMIXED SIMULATION DATA STRUCTURE	94
	Figure A-1 Detailed Level String Elements	95
	Figure A-2 Ideal Level String Elements	96
	Figure A-3 Level Output Specification Structure	97
	Figure A-4 Constants, Memories and Stacks	98
	Figure A-5 Control Logic Data Elements	99
	Figure A-6 Activity Queuing Data Structure	100
	Figure A-7 Temporary Storage Lists	101
APPENDIX B	INTERMIXED DATA STRUCTURE SIMULATION ALGORITHM	102
	1. Outline	108
	2. Discussion of Simulator	123
APPENDIX C	COMBINATIONAL LEVEL FORMULAS	125
APPENDIX D	DESIGN LANGUAGE DESCRIPTION	130
	1. Basic Structure	130
	2. Level Logic	132
	Figure D-1 Basic Level Delay Unit	134
	3. Control Event and Transfer Statements	134
	4. Integers	138
	5. Delimiters	138
	6. Iteration Statements	139
	7. Conditional Statements	141
	8. Define Feature	142
	9. Summary	143
BIBLIOGRAPHY		144

I. INTRODUCTION

For more than ten years, engineers have been utilizing general purpose digital computers to aid in the design of new digital systems.[1] The most common form of this aid has been the use of computers to convert descriptions of system logic into wiring lists and other documentation for construction, debugging and maintenance.[2] Included in this conversion process are such things as logic minimization, wiring rule checking, component placement and wire routing. Computers have been used to a lesser extent to simulate designs to verify that the logic yields correct results. The inputs to these systems are generally some special form of Boolean equations specifying flip-flop input and combinational logic block output formulas. Listings of these logic equations have served to supplement or even replace the more traditional logic diagrams.

The large number of logic equations required to specify a digital system makes it difficult for anyone unacquainted with the design to deduce its behavior from listings of the equations. Likewise, it is tedious and time-consuming to formulate designs in such detail. More concise register transfer languages have been developed for design documentation and formulation early in the design process.[3] Experience with design automation systems has led to the development of improved

---

[1] The first paper on computer design automation was given by S. R. Cray and R. N. Kisch at the Western Joint Computer Conference in February, 1956 - Reference (1).

[2] See References (2), (3), (4), (5), (6), (7) and (8).

[3] The first of these register transfer languages was presented by I. S. Reed in References (9) and (10).

register transfer languages for precisely describing digital systems.[4] These languages may describe machines at the register and decoder level, rather than the flip-flop and gate level of the logic equation languages. The great advantage of these new languages is that they describe hardware at a level of detail convenient to logic designers. Such a language has been under development at M.I.T. under the supervision of Professor J. B. Dennis. This design language has been used as a descriptive language for classroom exercises and to design a Microtape Controller for use with a modified PDP-1 computer. It is intended that the design language eventually serve as the input to a design automation and simulation system.

In this thesis we will be concerned with the development of a detailed simulation system for digital logic design verification. Models for logic circuitry are selected, data structures for representing these models are developed, and simulation algorithms are outlined. No simulation program has been written because of coming computer system changeovers at M.I.T., both at Project MAC and the Computation Center.[5]

Most logic simulation programs have been written to simulate a single digital system, either for debugging software to be run on the future system or for detecting logic errors. Of those simulation programs designed for logic checkout, the large majority have been written for synchronous machines and, in all cases known to the author, idealized circuit models have been used. Such models exclude consideration of such

---

[4] See References (11), (12), (13), (14) and (15).

[5] There are advantages to being forced to think about the program before being able to start any coding.

things as circuit delay, pulse width, delay tolerances and fan-out delay variations, and cannot be used to help locate the difficult to detect race conditions and timing errors which are especially important in asynchronous logic checkout. Relatively few simulation systems have been developed which translate special descriptions of designs into programs which simulate them.

Simulation for logic checkout has been widely used in the defense industry, where great emphasis is placed on short design lead time and production runs are short. High competition in the commercial side of the industry is also moving effort away from prototype debugging and toward simulation to detect logic errors as early as possible. With the introduction of third generation computer hardware the cost of constructing a prototype and modifying the design before production is increasing rapidly, again providing economic reasons for using simulation to verify a design before tooling up for prototype construction.

It is intended that this thesis be the basis for a simulation system which uses the Dennis Design Language as input and operates in a large time-shared computer environment. In addition to checking the overall operation of a design, this system would perform special simulations to test for suspected race conditions and logic hazards. It is not proposed that the system automatically check for all possible design errors, but rather that the judgment of the designer be relied upon to select likely problem areas and simulations for testing them. The advantages of this approach are that no additional restrictions need be placed on designs and the simulation system is fully compatible with current design techniques. Circuit and signal models are provided with sufficient timing

detail for race and hazard detection. The internal data structure is designed to facilitate on-line modification to the logic descriptions as errors are detected. These modifications do not require a complete reordering of the data structure; the data structure can be changed incrementally. Simulation efficiency is improved by evaluating only those combinational levels whose values are required and which may have changed since last evaluated.

Section II introduces the design language by way of a brief example and indicates the classifications of circuit and signal types in which a design is described. In the next section we select models for representing the important characteristics of each circuit and signal type. Section IV presents a data structure for representing ideal circuit and signal models and outlines a simulation algorithm based on the structure. The structure is modified to represent detailed models in Section V and a method for intermixing ideal and detailed models in the same structure is discussed. Section VI proposes modifications to the design language so that circuit and signal timing parameters can be declared and the language can be utilized as the simulation command language. Conclusions are presented in Section VII.

II. DESIGN LANGUAGE

In this section we will examine the design language description of a small part of a digital system. The motivation for doing this is to demonstrate the use of some of the features of the design language[6] and to show that three different classes of signals are represented in such a design description. Separate models will be developed for each of these classes of signals and will be included in simulation data structures to be presented later. This will simplify translation from the design language into the data structure, but more importantly, the organization is a natural way of looking at complicated digital networks.

The logic network selected for demonstrating the design language is diagramed in Figure 2-1. The network is a four-bit counter with parity and is realized in common pulse-level hardware. Arrowheads represent pulse inputs and diamonds represent level inputs. Counter parity is generated using carry pulses when it is stepped. Signals are named, and the same names are used in the design language description.

In the following design language description, the counter is specified as a component interfacing with one other component called Maincontrol:

---

[6] A complete syntax was written for the design language by G. J. Burnett in Reference (16). A discussion of the language more consistent with its present state of development is found in Reference (17), written for M.I.T. course 6.535 by H. F. Ledgard. Examples of the use of the language are found in a companion set of notes, Reference (18). Since these references may not be available, a brief outline of the basic design language is included in Appendix D.

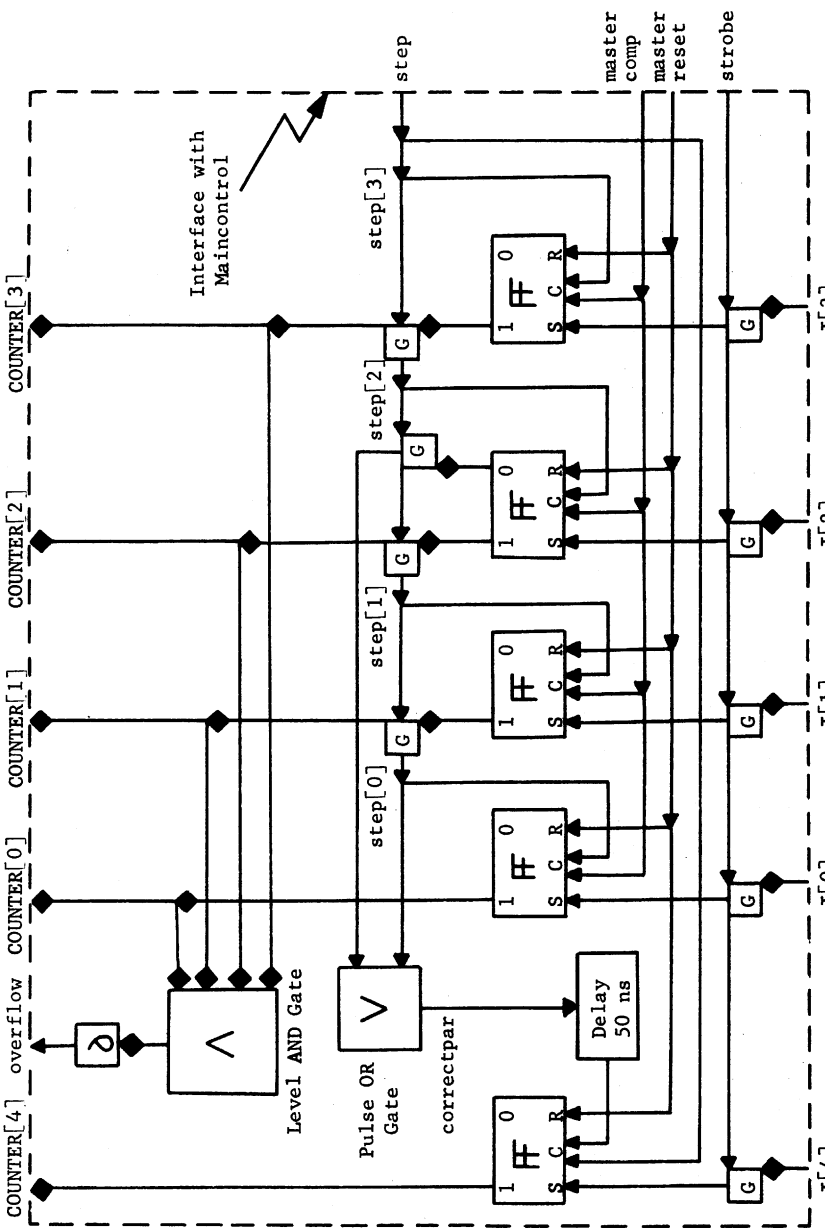


FIGURE 2-1

Four Bit Counter

```
1   component FOURBITCOUNTER;

2   interface MAINCONTROL;
3           input pulse strobe, step, masterreset, mastercomp;
4           input level  I[0:4];
5           output pulse overflow;
6           output level COUNTER[0:4];
7   end interface MAINCONTROL;

8   register COUNTER[0:4];   * Bit 4 is parity bit;

9   strobe:  I → COUNTER;
10  * Step control;
11  step:           step[3];
12                  ↑ COUNTER[4];
13        for i = 1 through 3 do                    begin
14  step[i]:        ↑ COUNTER[i];
15                  if COUNTER[i] then step[i-1];  end;
16  step[0]:        ↑ COUNTER[0];
17                  correctpar;
18  step[2]:        if ¬ COUNTER[2] then correctpar;
19  correctpar:     delay (50 ns);
20                  ↑ COUNTER[4];
```

```
21   * Master Reset and Complement;
22   masterreset:     0 ~ COUNTER[0:4];
23   mastercomp:      ↑ COUNTER[0:3];  * Parity remains correct;
24   * Overflow Detection;
25            MAXCOUNT ≡ COUNTER[0:3] = 15;
26   MAXCOUNT:        overflow;
27   end component FOURBITCOUNTER;
```

The FOURBITCOUNTER component is delimited by lines 1 and 27. (Line numbers are not part of the design language but are used here to aid in the discussion.) Lines 2 - 7 declare the signal interface with the component MAINCONTROL. If the counter interfaced with any other components, the other signal interfaces would also be declared at this point. Next come the register declarations, which in this case consists only of the five flip-flop register COUNTER. Note that the syntax of the design language has much of the flavor of ALGOL syntax. Basic Symbols are underlined when they are more than one character long and semicolons are used to terminate statements. Asterisks are used to precede comment statements, as in lines 8, 10, 21, 23 and 24, rather than the ALGOL symbol comment. Identifiers made up of capital letters and digits are used to name level signals, identifiers made up of small letters and digits are used for pulses. When a pulse identifier appears in a statement label, hardware is to be included in the component to execute all statements down to the next label or delay statement whenever that pulse occurs. Therefore the meaning of lines 11 and 12 is:

whenever the pulse "step" is generated, the machine will generate the pulse "step[3]" and complement flip-flop COUNTER[4]. These actions are taken simultaneously, as opposed to the sequential execution of statements in an ALGOL program. This should not be surprising since the design language is used to specify computer hardware, which is generally highly parallel, while ALGOL is used to specify computer programs, which are at present sequentially executed.

Lines 13 through 15 constitute an iteration statement which describes the propagation of carrys when the counter is stepped. This feature of the design language is provided to save the logic designer the necessity of writing out descriptions for each duplication of the same hardware. Level and pulse names are indexed so that each name is unique. Lines 16 and 17 describe the carry into the last stage of the counter, which is slightly different from the others. Line 18 describes the gate on the false side of COUNTER[2] which is used to complement the parity flip-flop. Lines 19 and 20 indicate that the pulse "correctpar" is delayed 50 ns and used to complement the parity bit. The register reset and complement logic is described in lines 22 and 23.

Line 25 defines the combinational level MAXCOUNT and line 26 indicates that an "overflow" pulse is generated whenever MAXCOUNT changes from a zero to a one. This implies the level differentiator shown in Figure 2-1.

Note that the pulse "step[2]" occurs twice as a label: first, in the expansion of line 14 for i = 2, and a second time in line 18. It is perfectly correct for a pulse identifier to be used as a label any

number of times. The meaning of this is that all statements labeled by the same pulse become active simultaneously whenever the pulse is generated.

From this example it can be noted that three distinct classifications of signals are represented in the design language.

1. Level Signals

   This class consists of register, combinational logic and level delay line output signals. The important characteristic of a level signal is its logical value as a function of time.

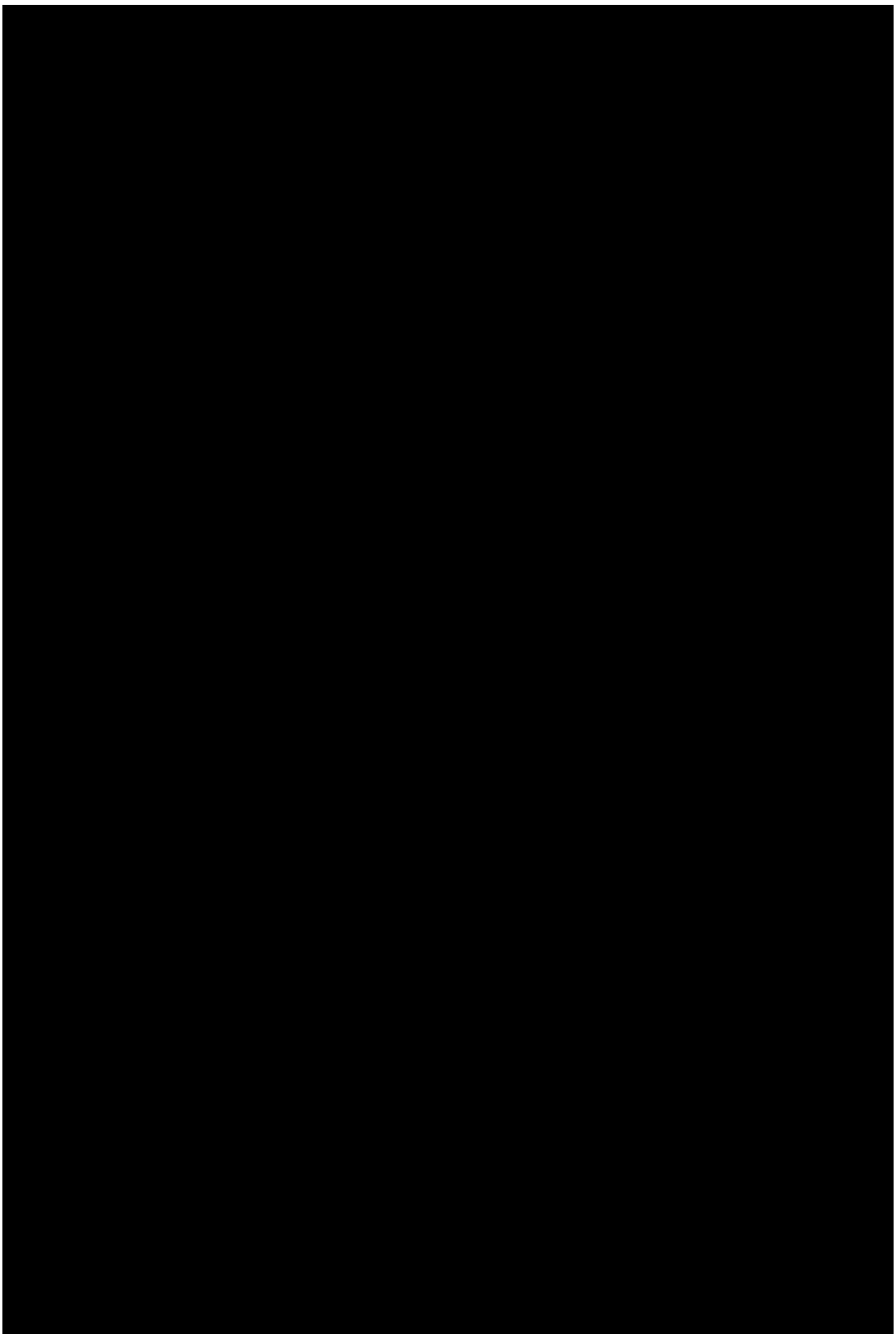
2. Events

   These are the command signals of a digital system. They may be generated by oscillators, other Events or level transitions. The important characteristic of an Event is its occurrence time. In the design language these signals are called pulses because in many digital circuit families they are realized as short duration pulses. This is not always the case - in some systems transitions in regular level signals are used as control events. If a value transition is to be used to generate an Event (line 23 of example) in short duration pulse logic, a differentiator sensitive to that transition must be used. This is not required in all-level circuit families.

3. Transfers

   This group of signals set, reset, complement and jam transfer data into flip-flop registers. They are usually

realized in hardware in the same way as Events. In all-level circuit families, flip-flop inputs have diode-capacitor networks which are only sensitive to 0 to 1 value transitions. Register transfers appear in the design language functionally, as in line 9 of the example, and are unnamed.



e) How often can I step this counter and check its parity without risking the possibility of sampling a level before it is settled?

Provisions must be made for simulation at the circuit block level to answer questions of this type and provide the desired error detection capabilities. The logic designer should be able to formulate his problem by specifying both the depth of simulation and the important circuit and signal parameters for each part of the design. Therefore, the models must be capable of expressing relationships between components modeled at various depths in a compatible manner. Before considering models for the three classes of signals discussed in Section II and the circuit blocks which generate them, let us consider the general problem of modeling circuit delay.

A. Circuit Delay

Logic designers use circuit delay calculations to determine at what times levels are settled and ready to sample and in what time intervals pulses occur. In many instances it is not necessary to make delay calculations because enough time is allowed between each command pulse for all levels to settle; the main reason for the predominance of synchronous logic is that most delay problems are eliminated. Even with synchronous logic there are occasions, such as carry propagation and parity checking, where maximum delay times are needed. In rare instances variables are sampled as input changes are being propagated; in these cases minimum delay times are required.[7]

---

[7] See Reference (19) for discussion of manual techniques for evaluating worst-case delay conditions.

Circuit delays cannot be treated as constants. They vary from one circuit to another of the same type and depend upon environmental factors such as electrical loading and operating temperature. In the signal models developed in this section, signal changes (either pulses or level transitions) will have both a starting time and a spread associated with them. The starting time will be the earliest possible time at which the change could occur and the spread will be the maximum interval of time in which the change could occur. The circuit block models will include minimum delay and ambiguity times, where the ambiguity time is the difference between minimum and maximum delay times through the circuit. Thus the starting time for a change in the output of a circuit block is the starting time of the input change plus the minimum delay of the block. Likewise, the spread of the output is the sum of the spread of the input and the ambiguity time of the circuit block.

B. Level Signals

As mentioned in Section II, level signals are the outputs of flip-flops, combinational logic, and delay lines. These circuits are enough unlike each other to deserve their own individual models. Before going on to discuss those models, let us first develop a model for level signals. From the above discussion we find that whenever the logical value of a level signal changes there is an interval of time, called the signal spread, when the value may be changing. Normally, it is improper to attempt to sample a level during a spread interval, so it is not important what values the signal takes on during that time.

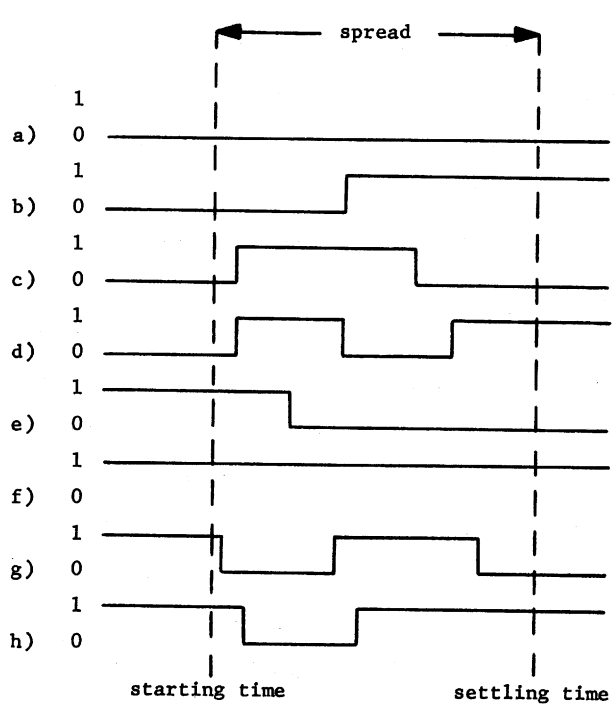
This is not true if the signal enters a circuit which is sensitive to level transitions. Such a circuit is the differentiator, which generates an output pulse whenever its input level changes from 0 to 1. The possibility that a level signal might change values several times before settling is termed a hazard.[8] A hazard occurring on differentiator inputs might produce false outputs. Therefore, it is necessary to be able to determine whether or not such levels have hazards during their transition spreads. Multiple hazards (the possibility of more than one double change in the value of a signal) are no worse than single hazards so there is no need to keep a count of them. It will be shown during the development of the combinational Logic Block that it is necessary that old value, hazard value, and new value be given for each input in order to calculate the hazard value (true if a hazard is present) of the output. Thus the value of a level signal is a three bit quantity - old value, hazard value, and new value. The simplest waveform for each of the eight possible values is shown in Figure 3-1. If a level is changing there is a time, called the settling time, when the change will be completed.

Delay Lines

The model for a level delay line is shown in Figure 3-2. The important characteristics are the minimum delay, τ_D, and the delay ambiguity, τ_A. If level I makes a change beginning at t with spread t_s, the level 0 will make the same change at $t + \tau_D$ with spread $t_s + \tau_A$.

---

[8] See Sections VIII and IX of Reference (20) by D. A. Huffman.



	VALUES		
	Old	Hazard	New
a)	0	0	0
b)	0	0	1
c)	0	1	0
d)	0	1	1
e)	1	0	0
f)	1	0	1
g)	1	1	0
h)	1	1	1

FIGURE 3-1

Level Signal Values

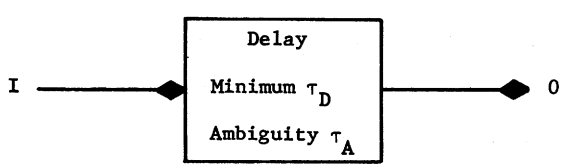
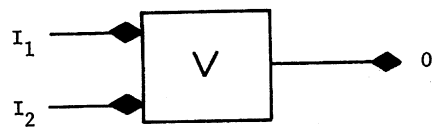


FIGURE 3-2

Level Delay Line Model

Combinational Logic Block

Combinational logic blocks are the standard level logic circuits whose outputs are some Boolean function of their inputs. Examples include level inverters, AND gates, OR gates, NAND gates, etc. Typical of this type of circuit is the 2-input AND gate of Figure 3-3. The truth table for this circuit is shown in Figure 3-3 (b). Such a truth table can always be written for a member of this group and is the most important characteristic of the block.



I_1	I_2	0
0	0	0
0	1	0
1	0	0
1	1	1

(b)

FIGURE 3-3

AND Gate

The detailed model for a combinational logic block is shown in Figure 3-4. It consists of a combinational block with zero minimum delay followed by a delay line with zero delay ambiguity.

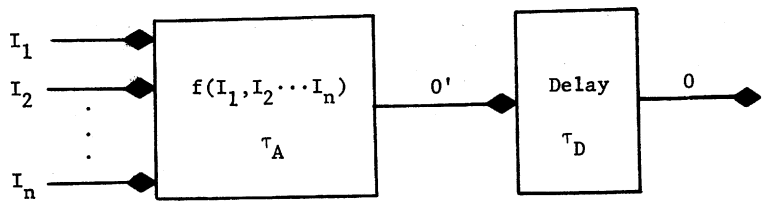


FIGURE 3-4

Combinational Logic Block Model

Note that the delay ambiguity is included in the left part of the model. Thus, in the many cases where minimum delay is unimportant, the left part can be used as the complete model. The function $f(I_1, I_2 \cdots I_n)$ is a mapping of the old, hazard and new values of the inputs into old, hazard and new values of the output, 0'. The old value of 0' is a Boolean function of the old values of the inputs, just as the new value of 0' is the same Boolean function of the new values of the inputs. On the other hand, the hazard value of 0' is a Boolean function of the old, hazard and new values of the inputs.

Karnaugh maps[9] for calculating the hazard value for the outputs of 2-input AND and OR gates are given in Figure 3-5. Again, leftmost bits of the arguments are the old values, the middle bits are the hazard values, and the rightmost are the new values. The method used to determine the entries in these graphs is illustrated in Figure 3-6. Two sample sets of waveforms for the case where the inputs to the AND gate of Figure 3-3 both change at the same time and the same spread are given. The minimum delay time and delay ambiguity of the gate are assumed to be zero.

Figure 3-6 (a) shows that if I_2 goes to "0" before I_2 goes to "1" then no transient pulse is generated. If the timing is reversed, as in Figure 3-6 (b), a transient pulse is generated. Since the only information given is that both signals are changing during the spread interval, either case is possible. Therefore, the hazard value is "1"

[9] See pages 131-142 of Reference (21) by S. H. Caldwell.

$I_1 \wedge I_2$

I_2 \ I_1	000	010	011	001	101	111	110	100
000	0	0	0	0	0	0	0	0
010	0	1	1	1	1	1	1	1
011	0	1	1	1	1	1	1	1
001	0	1	1	0	0	1	1	1
101	0	1	1	0	0	1	1	0
111	0	1	1	1	1	1	1	1
110	0	1	1	1	1	1	1	1
100	0	1	1	1	0	1	1	1

(a)

$I_1 \vee I_2$

I_2 \ I_1	000	010	011	001	101	111	110	100
000	0	1	1	0	0	1	1	0
010	1	1	1	1	0	1	1	1
011	1	1	1	1	0	1	1	1
001	0	1	1	0	0	1	1	1
101	0	0	0	0	0	0	0	0
111	1	1	1	1	0	1	1	1
110	1	1	1	1	0	1	1	1
100	0	1	1	1	0	1	1	0

(b)

FIGURE 3-5

AND and OR Gate Hazard Values

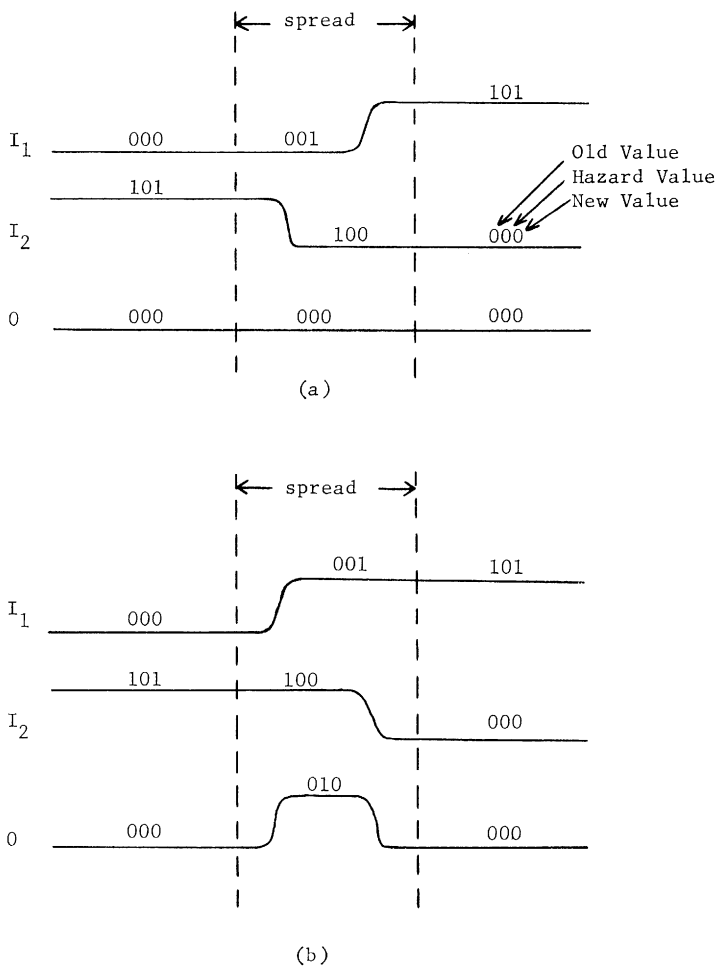


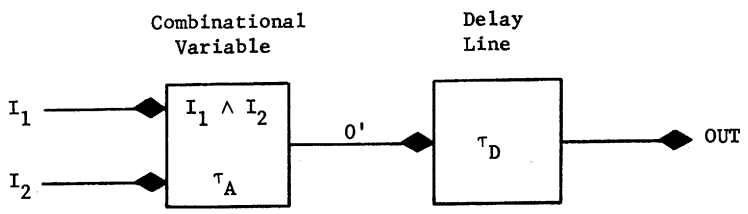
FIGURE 3-6

Output Hazard for 2-Input AND Gate

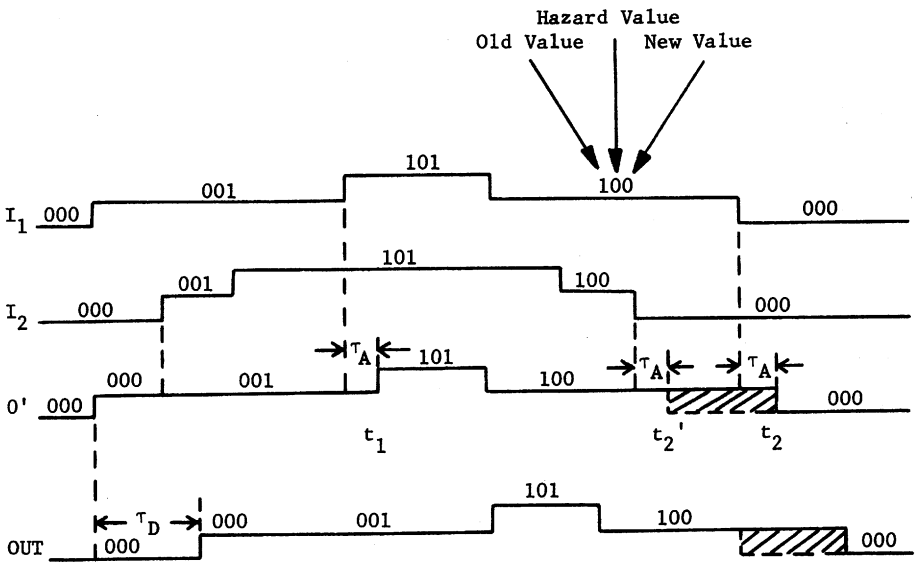
as shown in the shaded square of Figure 3-5. An examination of either part of Figure 3-6 indicates that the functions cannot be simplified to eliminate the need for one of the input value bits. Thus the old, hazard and new values of all inputs to a combinational logic block are required to compute the output hazard value.

Let us now propose and discuss a method for determining the settling time (and thus the spread) of the output level 0' of the combinational logic block model of Figure 3-4. Whenever an input to the combinational logic block changes, the output 0' is re-evaluated. The delay ambiguity, τ_A, is added to the settling time for the changing input level. If this sum is larger than the present settling time for level 0' then it replaces it. When time moves forward to the settling time of 0' the new value replaces the old value and the hazard value becomes zero. Thus, the unsettled level 0' becomes settled.

Figure 3-7 (a) shows the detailed model for a 2-input AND gate. Figure 3-7 (b) illustrates its behavior for the set of given input waveforms. Unsettled levels are indicated in this figure as being half way between a settled "1" and a settled "0". The signal OUT is just signal 0' delayed τ_D seconds. Signal 0' becomes unsettled whenever one of the inputs becomes unsettled and remains so until τ_A seconds after both inputs are settled. Note that level 0' takes on the unsettled value 000. This occurs because the initial change in input I_1 does not change the output until input I_2 also changes. Thus the value of a level may not be changing even though it is unsettled.



(a)



(b)

FIGURE 3-7

Example of AND Gate Model Behavior

A weakness in this method of computing settling times is illustrated in Figure 3-7 (b). The value t_2 for the second settling time for level 0' is incorrect; the correct value should be t_2'. This is because level 0' should be settled τ_A seconds after either of the inputs has settled at "0". The simple approach of selecting the maximum possible settling time would tend to cause correct situations to be flagged as illegal during a simulation. At first this method was considered acceptable because it does not require evaluation of a combinational level's output value to calculate its settling time. In Section V we will find that it is necessary to re-evaluate combinational levels whenever an input changes. Therefore a more accurate method of calculating settling times can be used without great additional cost.

Figure 3-8 indicates how to calculate output settling times for 2-input AND and OR gates. If the output value is changing, the settling time of the change is calculated by adding the gate's ambiguity time, τ_A, to the settling time of one of the inputs, either t_1 or t_2. The figure shows which input settling time to choose for each input value pair - t_1, t_2, the maximum of the two or the minimum of the two. Zero entries indicate that no output changing is occurring and a settling time should not be calculated.

$I_1 \wedge I_2$

I_2 \ I_1	000	010	011	001	101	111	110	100
000	0	0	0	0	0	0	0	0
010	0	Min	t_2	t_2	t_2	t_2	Min	Min
011	0	t_1	Max	Max	t_2	Max	t_1	t_1
001	0	t_1	Max	Max	t_2	Max	t_1	t_1
101	0	t_1	t_1	t_1	0	t_1	t_1	t_1
111	0	t_1	Max	Max	t_2	Max	t_1	t_1
110	0	Min	t_2	t_2	t_2	t_2	Min	Min
100	0	Min	t_2	t_2	t_2	t_2	Min	Min

(a)

$I_1 \vee I_2$

I_2 \ I_1	000	010	011	001	101	111	110	100
000	0	t_1	t_1	t_1	0	t_1	t_1	t_1
010	t_2	Max	t_1	t_1	0	t_1	Max	Max
011	t_2	t_2	Min	Min	0	Min	t_2	t_2
001	t_2	t_2	Min	Min	0	Min	t_2	t_2
101	0	0	0	0	0	0	0	0
111	t_2	t_2	Min	Min	0	Min	t_2	t_2
110	t_2	Max	t_1	t_1	0	t_1	Max	Max
100	t_2	Max	t_1	t_1	0	t_1	Max	Max

(b)

FIGURE 3-8

AND and OR Gate Settling Times

Flip-Flops

Flip-flop output levels are modeled the same way as other level signals. Old and New Values have the same meaning as before. A Hazard Value of "1" indicates the rapid recomplementation of the flip-flop before the previous change has settled. Again minimum delay time and delay ambiguity reflect the spread of possible transition delays whenever the flip-flop's output is changed. The spread of the output change is calculated by adding the flip-flop's ambiguity time, τ_A, to the spread of the transfer which caused the change.

There are special restrictions on flip-flop inputs which should be included in the model. The first of these is that an attempt to simultaneously set and reset a flip-flop with different signals is improper and should flag an error during simulation. In many logic families the normal method for complementing a flip-flop is to use a single signal to both set and reset it. When two separate signals are used the result is ambiguous due to variances in arrival times and amplitudes. Similarly, an attempt to simultaneously complement and either set, reset or complement a flip-flop with different signals is improper. The last special restriction applies only to complement inputs. Attempts to strobe these too soon after the flip-flop output has begun to change can lead to incorrect output values. Therefore, the flip-flop model shown in Figure 3-9 includes a mechanism for detecting attempts to complement the output value before τ_C seconds (minimum complement time) after the output begins to change. τ_C is a constant like τ_A and τ_D provided to the simulator by the logic designer.

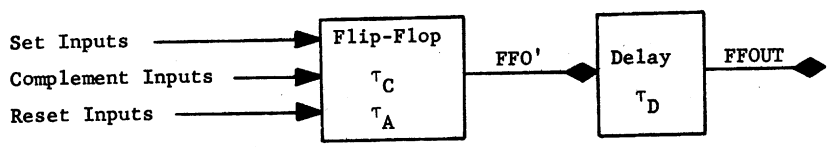


FIGURE 3-9

Flip-Flop Model

Differentiators

These circuits produce output command events whenever their inputs make "0" to "1" transitions. The spread of the output event is the same as the spread of the input level change which generates it. If a differentiator input ever has a hazard value of "1", a simulation error flag is set.

C. Register Transfers

These signals are modeled somewhat differently than the level signals we have been discussing. The important characteristic of this type of signal is its behavior when activated rather than its value at any instant of time. Therefore transfers are modeled in terms of their effects on registers. The complete specification of a register transfer includes:

1. The type of transfer, such as jam transfer, ones transfer, zeros transfer, or complement transfer.
2. The name of the destination register and a specification of the bits affected.

3. The names of the source level signals to be transferred to the destination register. Source levels may be flip-flop (register), combinational logic, or delay line outputs.
4. The delay ambiguity time for the transfer. This figure is used to represent the propagation difference between the minimum and maximum transfer paths when timing might be an important factor in a simulation. The spread of a register transfer is calculated by adding the transfer ambiguity time to the spread of the command event which activates the transfer.
5. If a transfer is ever attempted when one or more of the source levels is changing value, a simulation error is flagged.

D. Control Events

Control events may cause register transfers to occur and, provided that certain level signals have the proper values, may cause other control events to become active immediately or at some later time. The model for a control event includes:

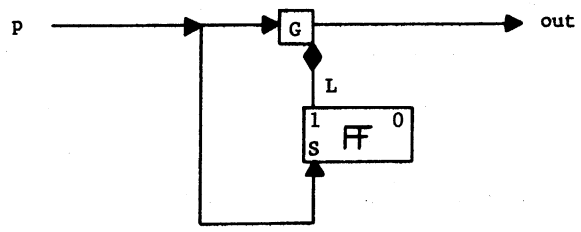
1. A list of transfers which take place whenever the event becomes active.
2. A list of level signals which act as conditions for the events that may be triggered by this event. The value of each of the condition levels is sampled when the event becomes active. Associated with each level is a list of control event - delay time pairs. Those control events

associated with condition levels with correct values are made active after the paired delay time has elapsed. A sampled condition level whose value may be changing causes a simulation alarm to be flagged.

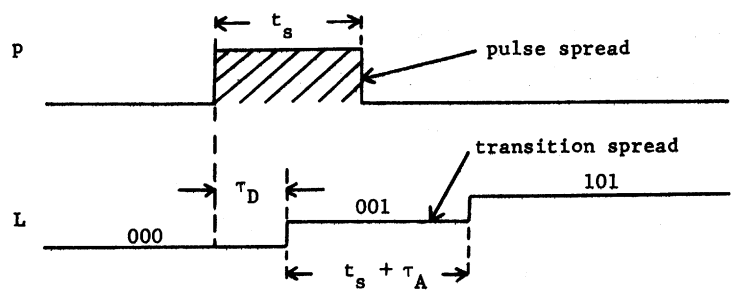
3. Control event minimum delay time.
4. Control event ambiguity time. When modeling timing very precisely this parameter is used to indicate maximum possible differences in arrival times of this signal to the various points it fans out to. The spread of any given activation of a control event is calculated by adding its ambiguity time to the spread of the control event or level transition which activated it.

This completes the development of detailed simulation models for signals and level circuit blocks. Before introducing simulation data structures based on these models, let us point out a serious weakness in the manner in which signal spreads are computed. The signal spread concept was introduced to detect logic design faults, either those involving level hazards feeding differentiators, or those caused by Events sampling Levels which may, depending on circuit variations, have more than one possible value. This second detection problem can be stated as follows. Given that Event p samples Level L at any time t within its spread of occurrence times, can Level L have more than one possible value? Unfortunately, the models do not keep track of the interdependence of signal spreads and can only determine whether or not the spreads overlap. This leads to the detection of logic faults which do not, in fact, exist. A simple example of this is illustrated in

Figure 3-10. During a simulation run pulse p is found to have a spread (t_S) greater than the minimum delay time (τ_D) of the flip-flop. Therefore, the spreads of pulse p and the transition of level L overlap, as shown in Figure 3-10 (b), and a false error detection is made. A situation which can generate a large number of incorrect fault detections is a signal feedback loop such as a delay line ring used as a time pulse distributor. If non-zero delay ambiguity is assigned to any circuit or signal in the loop the amount of spread in the signal increases each time it goes around the loop. Thus signals are generated with increasing spreads and cause more and more incorrect error detections.



(a)



(b)

FIGURE 3-10

Example of Signal Spread Fault

A timing model capable of avoiding these difficulties must retain additional information so that the dependence of signal spreads on the occurrence times of other signals can be calculated. One way of doing this is to keep track of signal histories. When possible sampling faults are detected the histories both the Event and Level are traced back to their common sources, if any. Then the signal spreads are recalculated forward from those points removing ambiguity common to both. Pre-simulation analysis can be used to determine in advance which parts of a signal's history should be retained and thus sharply reduce the amount of storage required. Designs which require extremely long signal histories could be rejected as not simulatable. An alternate approach would be to carry signal histories through only a fixed number of circuit blocks. This technique would not be able to establish remote signal dependencies, but the more common cases, such as the one in Figure 3-10, would be detected. Such modeling would still require longer running time and a greatly expanded data space and is not considered here. Instead, the responsibility for avoiding incorrect fault detection is assigned to the designer using the simulator.

Logic designers generally are aware of the areas of their designs which may contain timing errors. The most fruitful uses of detailed timing simulations are in specially tailored tests of such problem areas rather than exhaustive testing of complete designs. The fault detection problem discussed above is not as severe in these special cases if adequate means are provided for masking out obviously incorrect error detections. It is clear that as simulation becomes less expensive and more desirable, a more adequate method of establishing signal spread interdependences must be provided.

IV. DATA STRUCTURE FOR IDEALIZED MODELS

Two categories of information can be provided by logic simulation based on the models of Section III. The first is used to check the overall behavior of the simulated design against design objectives and includes the values of registers or other level signals at selected times. The second category consists of information about possible logic hazards and timing errors; this aids the logic designer in tracking down and correcting these more difficult to detect design errors. The simulation data structure and matching simulation algorithm presented here are based on idealized circuit models - fan-out, delay and transition times are ignored and level hazard detection is not included. Only information in the first category can be provided by such a simulation. The data structure will be expanded to deal with the complete circuit and signal models in Section V.

This simulation system is designed to be operated on-line by the logic designer and communicates with him via a modified form of the design language. In order of importance, the system design goals are:

a) Any design, synchronous or asynchronous, that can be described in the design language should be simulatable.

b) Timing and parallel operation should be modeled as consistently and accurately as possible. Simulations must be repeatable.

c) It should be possible to make incremental modifications to a simulation as the designer makes changes to his design language description.

d) The simulation system should be densely packed to allow large designs to be simulated with a minimum of memory swapping.

e) The simulation system should be organized to run as fast as possible.

f) The data structure should be organized to ease translation back and forth between it and the design language.

g) The logical complexity of the system should be minimized to reduce the effort needed to program and describe it.

A large number of digital simulations have been based on idealized circuit models similar to the ones used in this section. In some of these cases simulation systems have been provided to translate special forms of a design's logic equations into a program which simulates it;[10] often special simulation programs had to be written for each new design.[11] In each case, special code was included in the program to simulate each logic equation or circuit block of the design. The simulation systems developed here use a data structure, derived from the design language description of the logic to be simulated, as the input to a fixed, table-driven simulation procedure. There is a close correspondence between the design language description and the data structure it represents. This is because the data structure is actually a direct

[10] See References (3), (13), (22) and (23) for examples. In all cases known to the author special descriptions of the designs had to be made for input to the simulation systems.

[11] Examples of such programs and techniques for increasing their efficiency are found in References (24), (25), (26) and (27). An example of macroscopic simulation of digital systems is found in Reference (28).

representation of a subset of the design language. Thus it is fairly easy to translate back and forth between them and make incremental modifications to the data structure. This organization results in a densely packed simulation structure. The price paid for this is slower running speed, but the extra time is more than made up by re-evaluating a combinational level value only when its value is needed and its inputs may have changed since it was last evaluated.

The requirement for incremental changes to the data structure leads to consideration of list structures. Large parts of the data structure are formulated in this manner. It is often much more time-consuming to recover information from list structures than equivalent fixed data blocks. Fixed blocks are used to represent data whose size is not likely to be incrementally altered and in those cases where list structures would be unnecessarily wasteful of space or difficult to quickly access. Lists are used to describe variable length information which the designer may later wish to add to or delete from. This results in a mixture of interrelated lists and fixed blocks of many different lengths.

The total data structure can be divided into three separate interconnecting parts. The first two change size or shape only to reflect changes in the logic design being simulated. The third part of the structure is modified during a simulation run.

1) Level Logic

   This structure describes the output values and interconnections of the registers, combinational logic blocks, level delay lines and differentiators. Special structures are also included to simulate constants and memory interfaces.

2) Control Logic

Control events, register transfers, and their interrelations are described in this part of the data structure. A special Transfer which terminates the simulation is included.

3) Time Queuing and Miscellaneous Lists

This part of the structure is used by the simulation program to queue up future activities and keep track of temporary information such as subroutine arguments and input-output data.

A. Level Logic

Information describing output level signals is included as part of the data for each flip-flop, combinational logic block and level delay line. This information consists of the value of the level,[12] which may be sampled by Events or be a source for a Transfer, and a list of all circuit blocks with outputs dependent on the value or transitions of this level. It is convenient for the simulation program to be able to access this information no matter what the source of the output signal might be. Therefore output signal information is stored in the same format for flip-flops, combinational logic blocks and level delay lines.

Flip-flops are organized into n-length strings in the data structure, where the value of n depends on the machine the simulation is run on. Registers of length n or less are represented as adjacent bits on the same string and larger registers are represented as adjacent bits on two or more flip-flop strings. This allows straightforward register

[12] Only single bit values are used in this Ideal Model structure because level hazard detection is not included.

transfer specifications. To preserve uniformity in accessing output signal information and to allow simplified specification of a group of levels for Transfers, combinational logic blocks and level delay lines are also organized in n-length strings. Levels which are sources for the same multiple-line Transfers, or which are indexed bits of the same level register in the design language description, are ordered together within the same string. This presents an optimization problem because the same level may be a source for several transfers.

Figure 4-1 illustrates the data element representing a string of combinational logic blocks. The conventions used in the figures illustrating data elements are as follows:

a) Small blocks containing information are called cells.

b) Cells outlined in solid lines are part of fixed block made up of adjacent registers of memory.

c) Cells outlined in dotted lines are part of a list made up of generally non-adjacent registers of memory threaded together with pointer addresses.

d) Solid arrows between cells indicate that the source cell contains a pointer to a destination element of the type shown.

e) Dotted arrows are the same as solid ones except that the source cell may contain the null pointer indicating a void destination.

Although all cells are represented in the figures by the same size blocks, this does not imply that all cells need be the same size in the

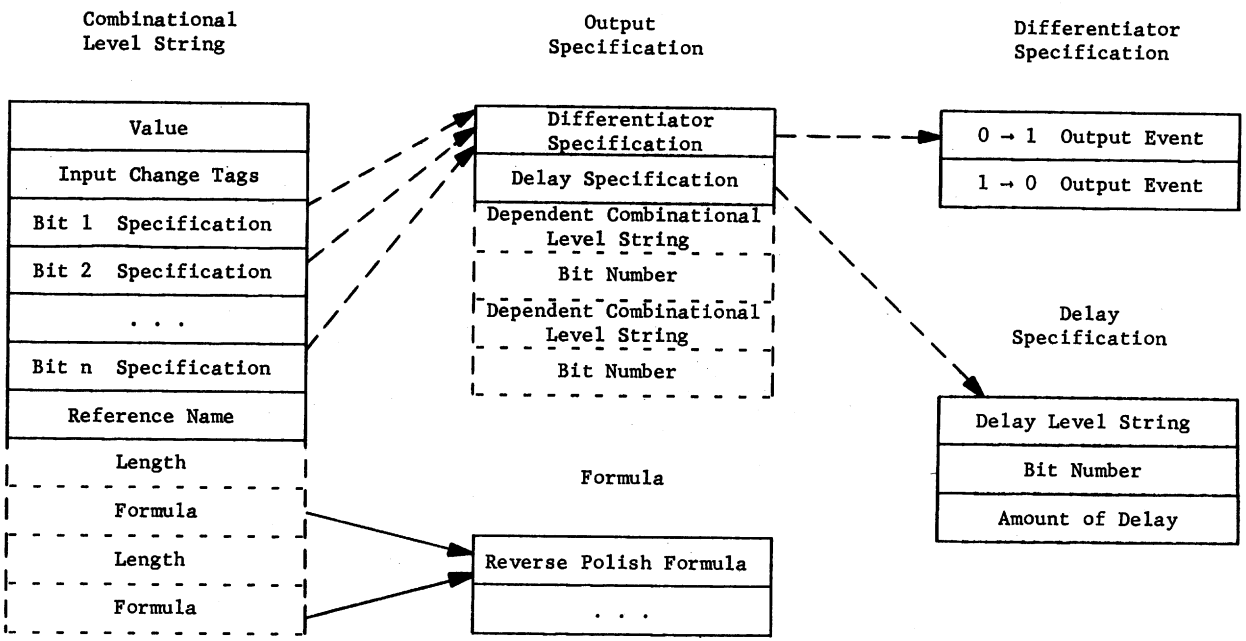


FIGURE 4-1

Combinational Level String Data Elements

data structure. The amount of information stored in a cell varies and there may be circumstances where it is worthwhile having different size cells. A case in point is a two-cell location specification of a combinational level bit. The first cell contains the address of the combinational Level String and the second cell contains the bit number. It is possible to include all of this information in a single word for many computers. Therefore cell is not necessarily synonymous with computer word or address field.

The data element for a combinational Level String consists of an n + 4 cell fixed block followed by a variable length list. The first cell in the element contains the values last calculated for the n output levels. Cell 2 contains an Input Tag bit for each of the output values. A "1" indicates that one of the inputs to that combinational logic block may have changed since the last time the output value was re-evaluated; a "0" indicates the output value is still valid. The next n cells contain pointers to Output Specifications for each of the combinational blocks. The next cell contains a reference name which is used by the routine which translates back and forth between the data structure and the design language.

The last cell in the element contains a pointer to a list specifying formulas to be used to compute the output level values. The first cell on this list contains the number of bits for which the first formula is valid. The second cell has a pointer to a formula for these bits. Next follow pairs of lengths and formulas until the end of the list is reached. If some of the bits on the combinational Level String are unused, the list ends before the sum of the lengths totals up to n.

The formulas themselves are modified Reverse Polish representations of combinational formulas found in the design language description itself.[13] For example, the formula {A[0:4] ∧ (¬ B[3:7] ∨ C[0:4])} would be translated {A', a, B', b, ¬, C', c, ∨, ∧}, where A', B', and C' are pointers to the level strings representing A, B and C, and a, b and c are the bit numbers of A[0], B[3] and C[0] respectively. Each symbol is contained in its own cell on the formula and operators must be distinguishable from pointers to level strings. Figure 4-1 shows that Formulas are represented in fixed blocks. This is done because it is unlikely that a formula would be modified incrementally; more likely it would be completely changed.

The structure for an Output Specification is a three-cell fixed block and a variable length list. If the level feeds a differentiator, the first cell on the block points to a two-cell Differentiator Specification. The first cell on this fixed table points to the Event, if any, triggered by a "0" to "1" transition of the output level. The second cell is for "1" to "0" transitions. If the output level feeds a delay line the second cell of the Output Specification contains a pointer to a Delay Specification. The first two cells of the Delay Specification indicate the delay string and bit number of the delay line and the third cell specifies the amount of delay. There is never a need for a level to feed more than a single delay line because any parallel network of ideal delays and combinational logic can be converted

[13] See Appendix C for a detailed discussion of Formulas.

to an equivalent serial network. The third cell in the Output Specification points to the list of all combinational logic blocks which use the level output as an input. When an Input Change Tag is set for some level during a simulation, these lists are used to propagate input changes to all combinational logic blocks dependent upon that level.

The data elements for Delay Level and Flip-Flop Level Strings are given in Figure 4-2. Note that values, output specification pointers and reference names are arranged as they were for combinational Level Strings. The values of delay lines and flip-flops are always kept up-to-date and their input change tags (second cell) are always zero. Thus it is not necessary for the simulation routines which evaluate combinational level values, test for transfer level values, or require output specification information, to distinguish between the three types of levels.

Level delay line inputs are specified at their source - only values, output specifications and reference names are included in Delay Level String data elements. Information about flip-flop inputs is included in the Transfer data elements. However it is quite possible for one Transfer to be resetting a flip-flop at the same instant another is setting or complementing it. This could be handled by the simulation program by arbitrarily allowing one or the other to have precedence depending on the order they are acted upon. Logic errors of this type are considered to be important enough, even at this level of simulation, to include additional information in the data structure for their detection. Therefore separate cells are included in Flip-Flop Level String elements to accumulate set, reset and complement input activations

during an instant of time. At the end of the instant, this information is used to re-evaluate the flip-flop values and detect flip-flop input timing errors.

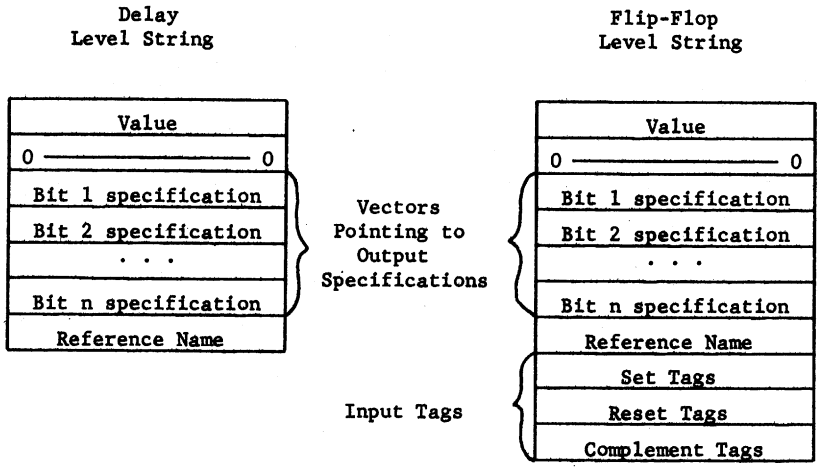


FIGURE 4-2

Delay and Flip-Flop Level String Data Elements

The level elements of Figure 4-3 have been included to model constant levels (wires to power busses) and standard addressable memories. The Memory Block may be used to ease the modeling of interfaces with non-logic devices such as core memories, tape drives and drums. Input change tags are included as part of each constant Level String for compatibility with Combinational Level Strings, although the tags are always reset. The memory model is so different than the other level sources that no attempt was made to make it compatible with them. The first four cells

specify address length, location,[14] maximum value, m, and a reference name for use of the translation routine. The next m + 1 cells represent the simulated memory. If it is necessary to simulate a memory with word length greater than the memory cell size, two or more Memory Blocks with the same address must be used. Memories act as the sources or destinations of a special set of Transfer operations.

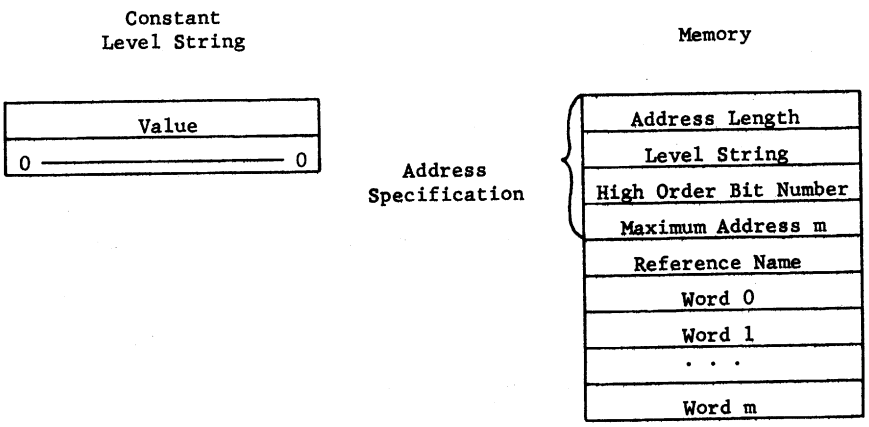


FIGURE 4-3

Constant Level and Fixed Memory Data Elements

[14] This implies that the address field cannot be any longer than n, the number of bits in a level string. This is reasonable because it should not be necessary to simulate large memories.

B. Control Logic

The data elements representing control logic are shown in Figure 4-4. The first two cells of an Event element point to Gate Lists which represent the control gates strobed by the Event. The gates in Gate List 1 are conditioned by levels with value "1" and those on Gate List 0 by levels with value "0". The first two cells of each gate specification list the level string and bit number of the conditioning level. The third cell points to the list of Events triggered if the conditioning level has the right value. Associated with each Event on an Event List is a delay time before activation; pulse delay lines are built in here.

The third cell of an Event element contains a reference name used by the routine which translates back and forth between the data structure and the design language. The last cell points to the list of Transfers which take place when the Event becomes active. Unlike the design language, all Transfers are unconditional. Conditional Transfers are achieved by introducing a new conditional Event which activates the Transfer. This simplifies the data structure by eliminating the need for equivalent of a "Transfer Gate List" without reducing the generality of the structure.

The first cell of a Transfer element specifies its type. Figure 4-5 contains the truth tables for the set of transfers available with standard set-reset-complement flip-flops. A complete set of eight is included to transfer source level strings to destination flip-flop strings. A second set is used to transfer Memory Block contents to flip-flop strings, a third if for level string to memory transfers and a fourth for memory to memory transfers. These sets must be distinguishable because of the format differences between Memory Blocks and level strings. Jam transfers

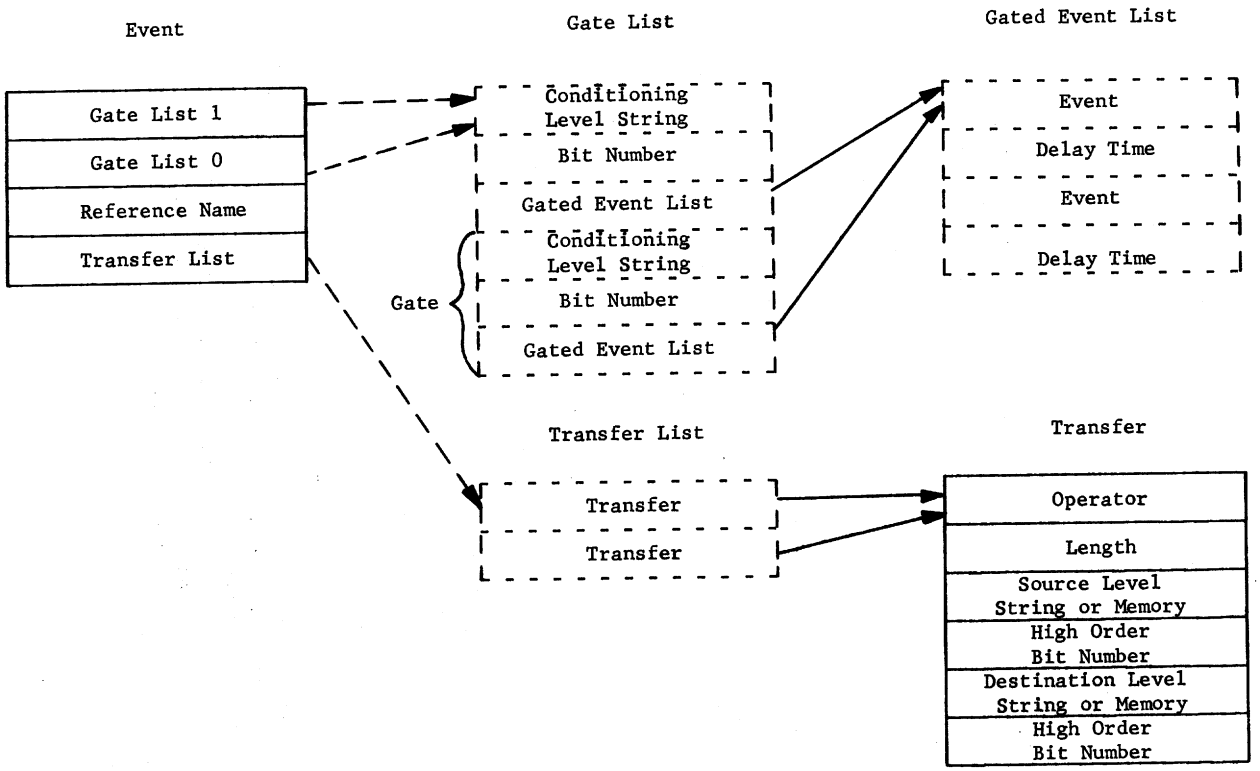


FIGURE 4-4

Control Logic Data Elements

Transfer Name	Symbol	Resultant B Value			
Jam	A ⇒B	0	0	1	1
One's Set	A → B	0	1	1	1
Zero's Reset[15]	A ~ B	0	0	0	1
One's Complement	A ↑ B	0	1	1	0
Negative Jam	¬ A ⇒B	1	1	0	0
Zero's Set	¬ A → B	1	1	0	1
One's Reset[15]	¬ A ~ B	0	1	0	0
Zero's Complement	¬ A ↑ B	1	0	0	1
Values Before Transfer	A	0	0	1	1
	B	0	1	0	1

FIGURE 4-5

Transfer Effect Table

[15] This is consistent with the present formulation of the design language. I would prefer that the meaning of the symbol "~" be changed so that a destination bit is reset if the corresponding source bit is a one rather than a zero, as it is presently defined. This makes the placement of the source gate on the true or false side of a flip-flop consistent with set and complement transfers and eliminates the need for an implied inverter when a combinational level is used as a reset transfer source.

are all that are absolutely necessary for memory reading and writing, but the additional transfers are easily and inexpensively included and may eliminate the need for a memory buffer register to be included on a flip-flop string. A special transfer is included which causes termination of a simulation when it is activated. The number of contiguous bits being transferred is contained in cell two of each Transfer element. Cells three and four specify the source level string or memory and the left-most bit number. Cells five and six do the same for the destination.

C. Time Queuing and Miscellaneous Lists

This part of the data structure satisfies needs for data fields which vary during a simulation. This includes the queuing of future simulation activity, handling recursive subroutine arguments and providing a means for storing input and output data which varies in length during a simulation.

An important part of this structure is shown in Figure 4-6. The Activity Queue, AQ, is a time-ordered list of future Events to be activated and Delay values to be changed. When the simulation program determines that an Event or Delay Level value change is to occur at some time t, an entry is added to the Event List or Delay Value List associated with t. If there is no previous entry on the AQ at time t, one is inserted. The first cell on the AQ contains the present value of simulated time and is called the Clock. The Clock is stepped by deleting the first three cells of the AQ. The second cell points to the Event List which is presently active; this is called the Immediate Event List. Likewise, the third cell points to the Immediate Delay Value List.

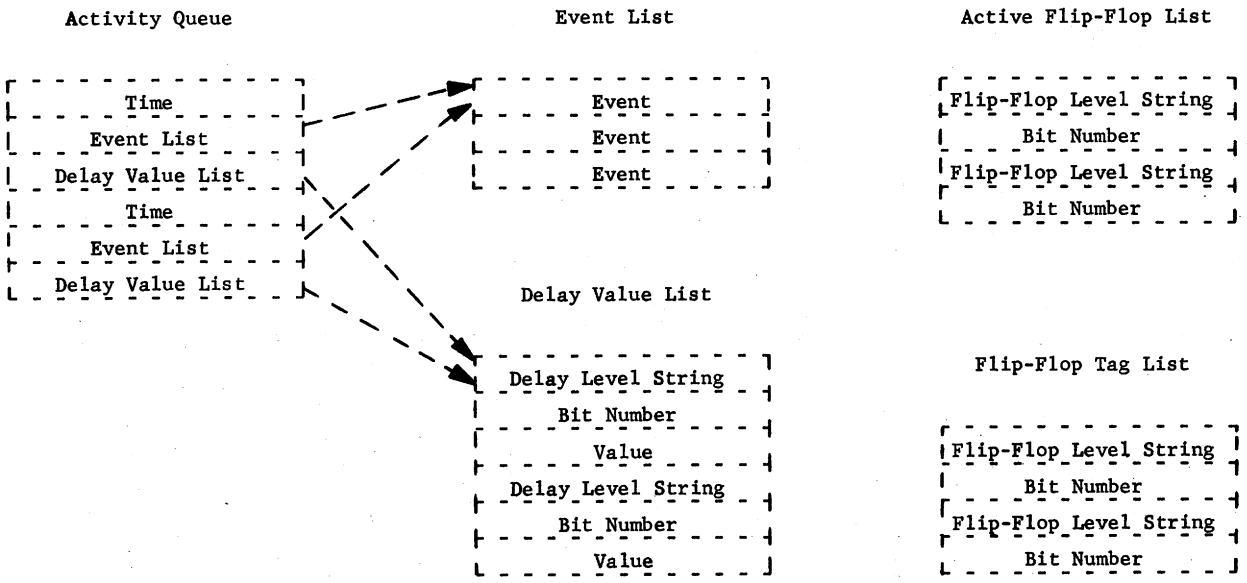


FIGURE 4-6

Time Queuing Data Structure

A number of temporary storage lists are kept by the simulation program. Two of these, the Active Flip-Flip and Flip-Flop Tag Lists, are included in Figure 4-6. They are used to keep track of the flip-flops which are changing value at a given instant of time. Additional lists are used to store recursive subroutine arguments. Data lists are used by the simulator to accumulate output messages. A special Transfer operation is provided to add messages to these lists.

D. Simulation Algorithm

A simulation begins by initializing flip-flop and delay line values, setting all combinational level Input Change tags and setting up the initial AQ. The Active Flip-Flop and Flip-Flop Tag lists are initially empty. The simulation takes off from there and continues until the Halt Transfer is executed or the AQ becomes empty. The algorithm proceeds as follows:

1. The Events on the Immediate Event List are activated one at a time until it is emptied.
   a) When an event is activated it is removed from the list.
   b) All conditioning levels on the Event's Gate Lists are tested and new Events are added to the AQ if the values are correct.
   c) All transfers on the Event's Transfer List are activated one at a time. Whenever one of a flip-flop's input tags is set for the first time, the flip-flop's level string location and bit number are added to the Active Flip-Flop List. If an attempt is made to set a complement tag when

it is already set, the flip-flop's Reference Name and bit number are given to the translation routine so that the user can be informed that a flip-flop input error has been detected. If the special Terminate Transfer is activated the Simulation Terminate Tag is set and the Reference Name of the Event is stored. This causes the simulation to stop at the end of that instant of time.

2. The new output values for the flip-flops listed on the Active Flip-Flop List are computed. If more than one input tag is set for a flip-flop, its value is unchanged. Otherwise the Set Tag causes the output value to become one, the Reset Tag causes the output value to become zero, and the Complement Tag causes the output value to complement.
   a) The Set, Reset, and Complement Tags are <u>not</u> cleared at this time. If more than one input tag is set for the same flip-flop, the flip-flop's Reference Name and bit number are given to the translation routine so that the user can be informed that a flip-flop input error has been detected.
   b) If the output value is unchanged, the flip-flop is removed from the Active Flip-Flop List and added to the Flip-Flop Tag List. This list is used to keep track of all flip-flops whose input tags have been set in a simulated instant of time.

c) The flip-flop's Output Specification block is checked. If the flip-flop output is the input to a "0" to "1" differentiator and it has made this transition, the differentiator's output Event is added to the Immediate Event List. A similar test is made for the "1" to "0" differentiator, if any.

d) If the flip-flop output is the input to a level delay and its value changed, an entry is made to an AQ Delay Value List to cause the delay line output to make the same change after the amount of delay listed on the Delay Specification. Any previous entry for the level delay listed at the same time is deleted. The previous entry was a record of transient behavior and is therefore replaced by an entry with the newer value.

3. The value changes listed on the Immediate Delay Value List are made one at a time.

a) If the new value is the same as the old, the delay line is removed from the Immediate Delay Value List.

b) The delay line's Output Specification block is checked. If the output is an input to a differentiator and its value makes the proper change, the differentiator output Event is added to the Immediate Event List.

c) If the delay output is an input to another delay and its value changed, an entry is added to the proper Delay Value list as in 2(d).

4. Input Change Tags are propagated to all combinational level bits dependent upon flip-flops still listed on the Active Flip-Flop List and delay lines still listed on the Immediate Delay Value List. This is done by using the Dependent Combinational Level list which is part of each Output Specification.
   a) If a combinational level's Input Change Tag is already set, there is no need to propagate tags past that point.
   b) If one of these dependent combinational level bits is an input to a differentiator, its new value must be computed. If the value makes the proper transition the differentiator output Event is added to the Immediate Event List.
   c) If one of the dependent combinational level bits is an input to a delay, its new value must be computed. If the value changes an entry is added to the proper Delay Value List as in 2(d).
   d) If the new value of a combinational level bit is calculated for one of the above tests and the value is found not to change, the Input Change Tags do not have to be set for combinational levels dependent on it.
5. The remaining entries on the Active Flip-Flop List are added to the Flip-Flop Tag List. The Immediate Delay Value and Active Flip-Flop lists are cleared. If the Immediate Event List is empty, then the simulator proceeds to step 6. Otherwise, it returns to step 1.

6. The Set and Reset Tags for all flip-flops on the Flip-Flop Tag List are cleared and the Flip-Flop Tag List is cleared. If the Simulation Terminate Tag is set, control is passed on to the translation routine along with the Reference Names of the termination Events. Otherwise the Clock is stepped. If the AQ is empty, control is passed on to the translation routine, else the simulator returns to step 1.

E. Discussion of Idealized Model Simulation

The simulation program whose algorithm has been outlined above operates under the control of another program which translates user commands, formulated in the design language, into a data structure and passes control to the simulation program.[16] After the simulation program terminates itself and returns control, the translator outputs results to the user and either returns control to the simulator or waits for new commands. The problems involved in writing such a translator are difficult; indeed, the translator is likely to be larger than the more complicated simulator of the next section. These problems are the same as those faced in translating compiler level languages into object programs and have been under extensive investigation for some time. The data structure is organized to ease these problems, otherwise they are not considered here.

The behavior of the simulator at any given instant of time is to simultaneously activate all register transfers and level delay line output changes due to past activity. These level changes are then

[16] Note that the simulation program and translation program need not be in core at the same time. This allows additional space for data structure.

instantly propagated through all the combinational logic. Any register transfers triggered by these changes are then simultaneously activated, etc. until the logic settles. Level hazards on differentiator inputs may or may not cause the generation of extraneous control pulses. Unstable circuits, such as the simple example of Figure 4-7, will cause flip-flop input alarms to be generated. The circuit will not oscillate because flip-flops are constrained to change values no more than once in an instant of time. If some delay were inserted into each feedback loop, the circuit would oscillate under simulation.

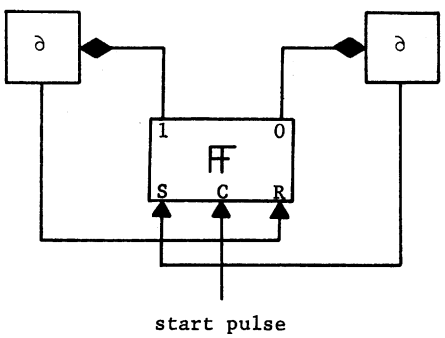


FIGURE 4-7

Unstable Circuit

Although it does not aid in detecting and isolating timing errors, the simulator should be of great help in checking out the gross behavior of a design. This is especially true early in the design cycle. The timing behavior of the simulator is as good as one can get from discrete

time idealized models. Running speed, although not the highest priority design goal, should be competitive with other simulation programs of comparable depth and generality. An outstanding characteristic of the simulator is that combinational levels are evaluated only when their values are needed and only if they may have changed since last evaluated. This feature becomes especially valuable for large designs, because ordinary logic simulators spend a large amount of their time evaluating combinational levels which could not have changed or whose values are not required. Another important characteristic of the simulator is the ease with which small changes can be made in the simulated design.

V. DATA STRUCTURE FOR DETAILED MODELS

We are now prepared to expand the data structure developed in the last section to represent the complete circuit and signal models of Section III. The modified structure will include circuit ambiguity time, signal spread, flip-flop minimum complement times and old, new and hazard signal values for logic hazard detection.[17] The effects of these changes on the size of the data structure and the complexity and speed of the simulator will be discussed. Further modifications will be made so that both ideal and detailed models can be intermixed within the same data structure. A method of partitioning the data structure for simulating large systems will be introduced. This data structure is designed for a simulation system which is operated on-line by the logic designer and communicates with him via a modified form of the design language, as was the case for the idealized model data structure. Therefore the design goals listed at the beginning of Section IV also apply here.

A. Level Logic

Most of the data structure expansion required to represent detailed circuit and signal models occurs in the data elements representing level circuits. Illustrations of the data structures for detailed flip-flop and combinational level strings are shown in Figures 5-1 and 5-2. The structure for detailed delay level strings is not illustrated because it is the same as the combinational level structure without

---

[17] Illustrations of the completed data structure are given in Appendix A. An outline of a simulation algorithm based on this structure is in Appendix B.

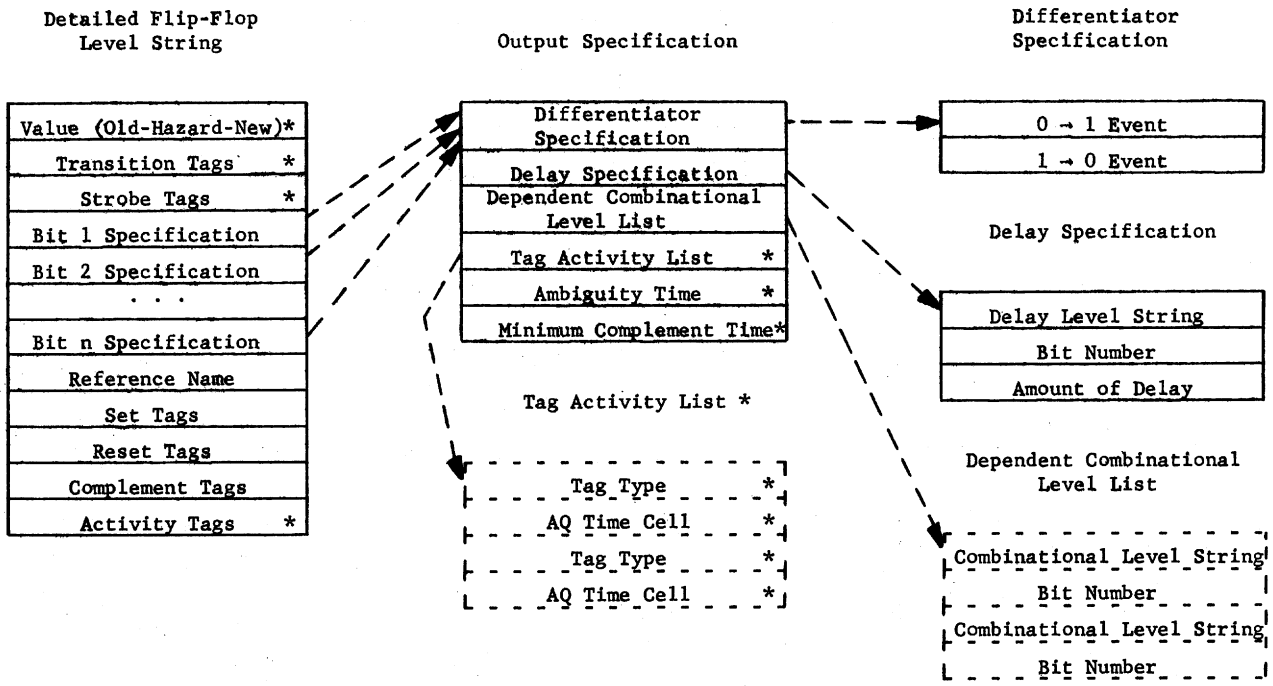


FIGURE 5-1

Detailed Flip-Flop Level String Data Elements

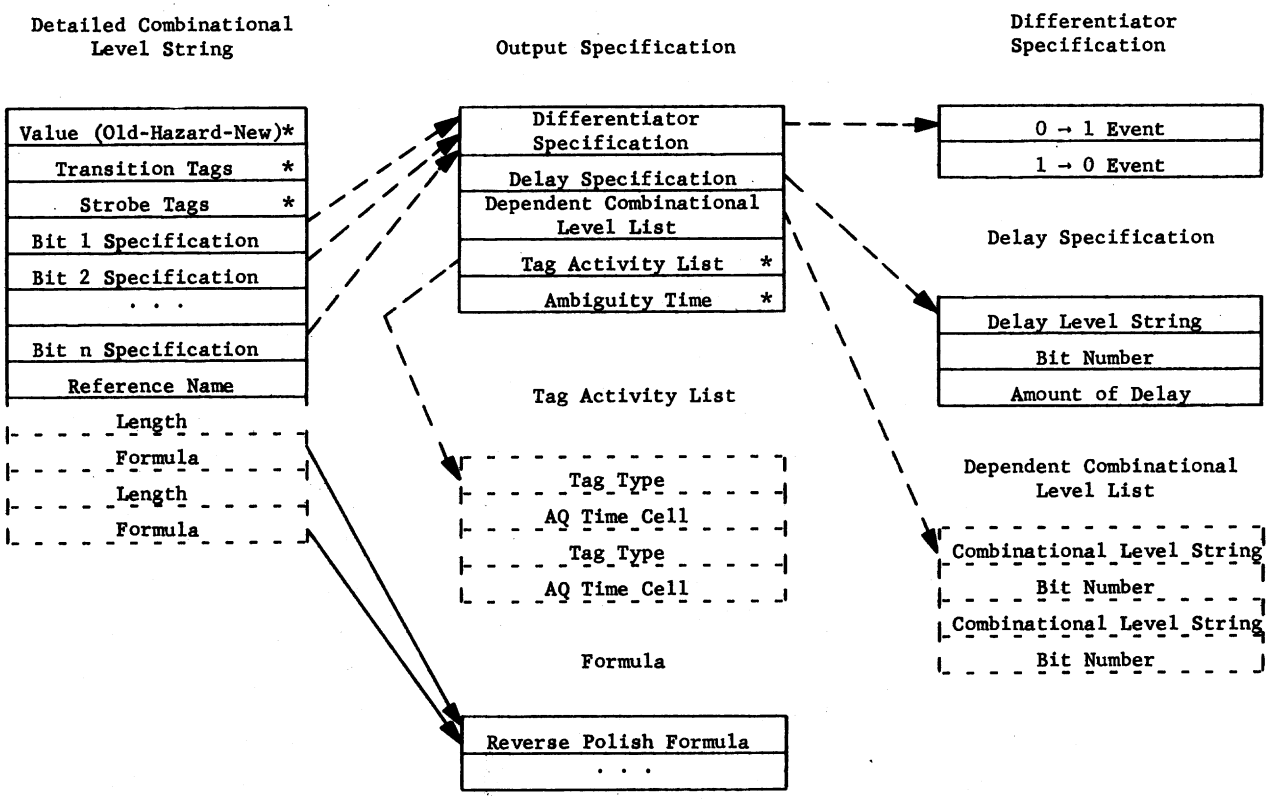


FIGURE 5-2

Detailed Combinational Level String Data Elements

formulas. Data cells which have been expanded or added to the idealized structures of the last section are marked with asterisks. Note that each level value cell has been expanded to include three output value bits - old, hazard and new. It was shown in Section III that these three values are sufficient and necessary to calculate the output hazard values of dependent combinational levels. Detailed level string value cells are therefore three times larger than those for idealized level strings, provided the string length, n, is unchanged.

The second and third cells of detailed level string elements each contain a new tag bit for each level represented on the string. A level's Transition Tag is on during the time spread of each of its value transitions. Whenever a Transfer is executed which changes the value of a flip-flop, the flip-flop's Transition Tag is set. Cell five of the flip-flop's Output Specification contains its ambiguity time, τ_A. This is added to the signal spread of the transfer to determine the time to reset the Transition Tag. When flip-flop value changes are propagated through the dependent combinational logic, each combinational level's transition spread is calculated as a function of its ambiguity time and the transition spread of its inputs. Likewise the signal spread for a level delay's value transition is calculated by adding its ambiguity time to the spread of its input signal's transition. If a level's value is sampled by an Event or if it serves as a Transfer source while its Transition Tag is set, an alarm message is generated by the simulator to inform the logic designer. Each level signal also has a Strobe Tag which is set whenever it is sampled by an Event or Transfer. It remains

set until the spread of the sampling signal is completed. An alarm message is generated if a level's output value changes while its Strobe Tag is set.

The last cell of a detailed flip-flop level string element contains an Activity Tag for each flip-flop on the string. These are used to detect flip-flop complement input rates which are too high. The last cell of a flip-flop's Output Specification contains its minimum complement time, τ_C. Whenever a flip-flop's Transition Tag is set, its Activity Tag is also set. When the Transition Tag is reset, the Activity Tag is reset after a delay of τ_C time units. Therefore the Activity Tag is set during the interval when it is improper to activate any of the flip-flop's complement inputs. An attempt to do so would cause an alarm message to be generated.

In addition to checking for minimum complement time violations, further flip-flop input error detection is accomplished by using the Set, Reset and Complement Tags. Rather than resetting these tags every time the clock is stepped, as is done for the idealized case, they remain on throughout the signal spreads of the set, reset and complement Transfers. If more than one of these tags is on at the same time for the same flip-flop, or if a complement transfer attempts to complement a flip-flop whose Complement Tag is already on, an alarm message is generated.

One of the important features of the idealized simulation system of Section IV is that a combinational level is re-evaluated when its value is needed, and only if one of its inputs may have changed since last evaluated. This is accomplished by setting Input Change Tags for all combinational levels dependent upon a flip-flop or level delay whose

value changes. Unfortunately this same technique cannot be applied to detailed combinational levels. A characteristic of the detailed model of a combinational circuit is that its output may still be changing for a period of time, equal to its ambiguity time, after its inputs are all settled. Furthermore, the output of a combinational circuit does not always change when one of its inputs changes. Suppose a signal were to sample the level during that period of time, and that this was the first time the level had been sampled since one or more of its inputs had changed values. If the simulation program were to attempt to re-evaluate the level at this time, it would be unable to determine whether or not the level might still be changing. The most satisfactory way of guarding against this situation is to re-evaluate detailed combinational levels whenever their input values change. Therefore Input Change Tags are no longer required. Naturally, if a combinational level is re-evaluated and found not to be changing, there is no need to re-evaluate the combinational levels dependent upon it. Since input level values and transition spreads are always known when output transition spreads are calculated, the more accurate method of calculating them discussed in Section III can be used.

B. Control Logic

The data elements representing control logic are shown in Figure 5-3. Note that the only changes are the additions of ambiguity time cells to Event and Transfer elements. When an Event or Transfer is activated, its signal spread is calculated by adding its ambiguity time to the signal spread of the triggering Event or level transition.

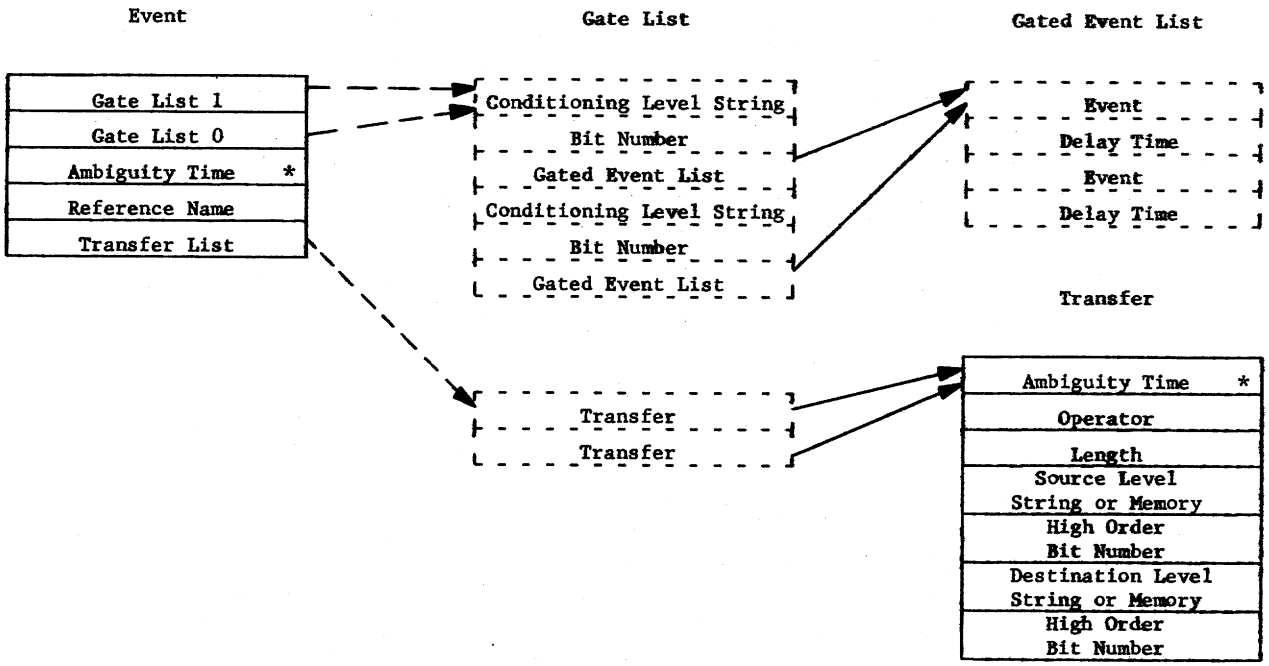


FIGURE 5-3

Detailed Control Logic Data Elements

Modifications to the data structure were considered for detecting Event doublets.[18] Doublets can cause trouble in logic because they may behave as a single Event in one section of the logic, and as more than one in some other section. If an Event doublet triggers a complement Transfer, it is not clear which state the destination register will settle in. This situation would be detected through use of the register's Activity Tags as discussed above. Another place where doublets might cause trouble is on logic interfaces. If the designer is concerned about the possibility of doublets on a line, he can cause them to be detected by using the signal to complement a dummy flip-flop with appropriate minimum complement time.[19] Therefore, it is not considered worthwhile to include special provision in the data structure for Event doublet detection.

C. Time Queuing Data Structure

The structure used to queue up simulation activity is illustrated in Figure 5-4. Asterisks are once again employed to mark data cells which have been added to the Activity Queue discussed in Section IV. Cells have been added to the Event and Delay Value Lists to carry signal spread information. When the simulation program becomes aware that an Event or level delay value change is to be activated, an entry is placed on an Event or Delay Value list. This list is attached to an Activity

---

[18] Two or more occurrences of the same Event at close to the same time; the signal spreads may even overlap.

[19] This is an example of a powerful simulation technique to be discussed more fully in Section VI.

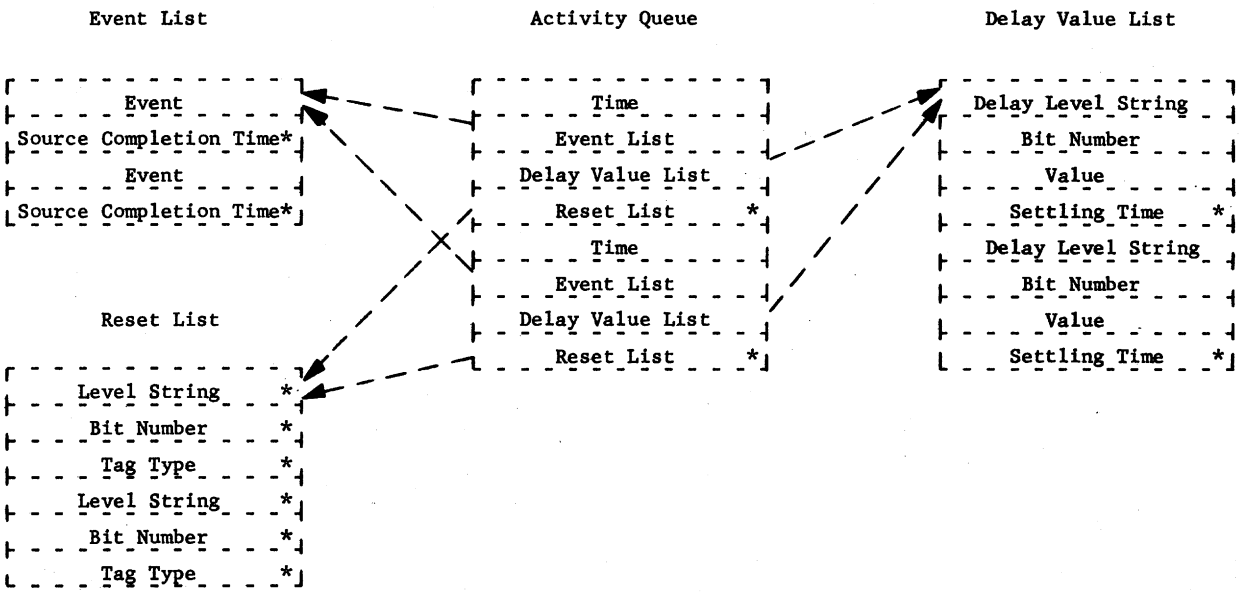


FIGURE 5-4

Time Queuing Data Structure

Queue time cell containing the starting time of the activation; i.e., the earliest possible time the Event or level delay value change could begin. One cell of the entry contains the activation's settling time, or the latest possible time before the Event or level delay value change would be completed if the Event or level delay had zero ambiguity time. Thus the actual completion time of an Event activation is the sum of the Event's ambiguity time and the activation's source completion time. Likewise, the time to reset the level delay's Transition Tag is calculated by adding its ambiguity time to the level transition's settling time.

Reset lists have been added to the AQ structure to queue up the resetting of Transition, Strobe, Set, Reset, Complement and Activity Tags. During the execution of a simulation program, it is often necessary to refer to the reset times of these various tags; in some cases it may even be necessary to change some of the reset times. It would be extremely inefficient to search through the entire AQ structure every time such a reference or change must be made. Therefore Tag Activity Lists have been added to each level's Output Specification. (See Figures 5-1 and 5-2.) Every time an entry is added to an AQ Reset List to reset some level's status tag, a matching entry is added to the level's Tag Activity List. The first cell of the entry specifies the type of tag to be reset. The second cell points to the AQ time cell containing the reset time. When the tag is reset, the Tag Activity List entry is deleted. Therefore a search through a short Tag Activity List is all that is required to fetch the reset time for a status tag. An additional search through a single AQ Reset List is required to delete a tag reset entry.

D. Intermixing Ideal and Detailed Models

The memory space required for a detailed simulation of a design is estimated to be approximately 25% greater than that required for the idealized modeling of Section IV. Running times should be substantially greater, between 2 and 3 orders of magnitude. Most of this additional time is taken in calculating the more complex values and settling times for level signals and in the bookkeeping associated with the various tag bits. As mentioned at the end of Section III, the most fruitful use of detailed logic simulation is in specially tailored testing of known problem areas. Only a limited area of the design need be represented with detailed timing models; the rest could quite satisfactorily be done with idealized models. By setting ambiguity times to zero, the idealized models can be realized using the detailed data structure, but processing time would not be reduced. Therefore the data structure should be modified so that both idealized and detailed Level Strings can be intermixed in a single structure.

There is not enough time savings to warrant inclusion of idealized Events and Transfers in the intermixed structure. Instead, a special value for ambiguity time, called Z, is included. If any Event or Transfer has Z ambiguity time, its signal spread is always zero no matter what the spread of the signal triggering it. In this way idealized Events and Transfers can be modeled in the intermixed data structure.[20]

---

[20] Z ambiguity time can be used by the designer to suppress false timing alarms such as the one generated by the circuit of Figure 3-10.

Z ambiguity time is also used to make Level String Outputs ideal. When a detailed signal is an input to a circuit with idealized outputs, the signal spread is ignored and the input is assumed to occur at the earliest of possible times. Thus, if a level signal generated by a detailed circuit model enters an idealized combinational level block, any transition in the value of the input signal is assumed to occur at the leading edge of its spread. Likewise, when a detailed Transfer uses an idealized flip-flop as a destination, the spread of the Transfer is always treated as if it were zero and the flip-flop changes value immediately.

There are certain references to Level Strings in the data structure where it is necessary to specify which type of model is used to represent the levels. For example, if a level is specified as an Event Conditioning Level within some Event's Gate List, it is necessary to know whether or not to check the level's Transition Tag and set its Strobe Tag. In the data structures discussed thus far a level signal is referred to by specifying its Level String and bit number. In these special cases an additional cell must be included specifying model type, either idealized or detailed.

E. Data Structure Partitioning

The data structures required to represent large designs can become very large; more than one central memory load may be needed. Since it is very desirable to be able to simulate indefinitely large designs, methods should be provided to partition data structures so that not all of them need reside in central memory at the same time. If the simulations are to operate in a time-shared environment, such as Project

MAC's new MULTICS system, they are likely to get better treatment from the storage allocation routines if they restrict their data space requirements over short periods of time. The design language aids segmentation of the data structure because designs are described in sections called components. Inter-component interfaces are specified with all levels and pulses given. The designer is in the best position to know which components interreact the closest with each other and he could designate sets of components to be grouped together to form sections of data structure.

All data elements representing interface levels between these data sections would be grouped together on data section 0, which would reside in central memory permanently along with the simulation program.[21] It is suggested that the entire Activity Queue structure remain in central memory all the time, otherwise partial queues would have to be continually merged and any savings would be offset by bookkeeping costs.

All references to Level Strings, Events and Transfers within the data structure must include a cell specifying the element's data section. During a simulation all activity local to a section is executed before moving on to the next section. Ideal Combinational Levels residing on section 0 are always re-evaluated when their Input Change Tags are set. This is done to avoid referring to information on a "non-permanent" section after moving on to another. To remove the

[21] In the case of a MULTICS type realization, the computer system's normal storage allocation routine would automatically move things in and out of core as they are used. In more conventional systems, the simulation program would have to initiate these swaps.

necessity of propagating level signal changes immediately from section to section as they are discovered, Input Change Lists must be established for each non-permanent section. These lists contain the names of all combinational levels whose inputs are changing, and the settling times of these inputs. Thus when a level signal change propagates through section 0, the level on section 0 is re-evaluated and the combinational level names on its Dependent Combinational Level List must be placed on the appropriate Input Change Lists. When the program moves on to a new section, the signal changes on its Input Change Level List are propagated into the logic. Before the clock can be stepped, all Input Change Level Lists must be empty.

Events may have gate conditioning levels and Transfers on more than one non-permanent data section. When an Event crosses a section interface in the design language description. a new Event is created in the corresponding data structure and is unconditionally triggered by the original Event with zero additional minimum delay and delay ambiguity times. All of the original Event's activities in the destination section are assigned to the new Event, which is part of the data structure on the destination section.

Note that the way in which a design is partitioned into data sections and the order in which the sections are acted upon by the simulator may vary simulation results. This is true because flip-flops are constrained to make no more than one output change at one instant of time. When more than one type of flip-flop input is simultaneously active, either the first input which the simulator processes dominates, or the flip-flop output is not allowed to change at all. Since the

order in which the inputs are processed depends on the data structure partitioning and the order in which the sections become active, the output of a flip-flop with multiple active inputs may also be dependent upon them.

This dependency can be eliminated by forcing the simulator to execute all Events on the Immediate Event List before propagating level changes into differentiators. Then all the Events generated by the first level change propagation would be executed before propagating their effects through the logic, etc. Operating in this manner would seriously reduce the amount of time the simulator could spend on one section before having to move on to the next. This is a high price to pay, so the output values of flip-flops with input errors (multiple active inputs) are allowed to vary with the way a designer partitions his design.

The simulation program can use the Immediate Event List, Immediate Delay Value List, Immediate Reset List and the Input Change Lists to anticipate section usage and ask the system memory allocation routine to ready sections in advance of actual usage. The optimum section size to use depends on the storage allocation algorithm used by the time-sharing system. If the section size is too small there would not be enough activity within it at a simulated "instant" of time to make the partitioning technique pay. If the section size is too large, parts of it might be swapped out because of disuse. For a given design, the smaller the non-permanent section size, the larger section zero must be because more interface signals are required.

F. Summary of Data Structure Characteristics

The data structure we have been discussing can be used as the basis of a simulation system capable of being of great value in the detection and isolation of common timing errors, as well as checking gross behavior of a design. The timing behavior is very realistic and the data structure is capable of both synchronous and asynchronous logic. The data structure allows intermixing of idealized and detailed circuit and signal timing models for optimizing simulation efficiency, and the data sectioning system allows very large systems to be simulated. Only the activity queuing structure, data section 0, two non-permanent data sections and the simulation program need be in central memory at any given time for efficient operation.

VI. SIMULATION COMMAND LANGUAGE

In this section we will propose some extensions to the design language so that it can be used both as input to the simulation system and as the simulation command language. These extensions can be divided into the following categories:

1) Model Declaration Statements

These are to be used to declare timing information about circuits and signals and to specify storage arrays which are useful in simulating logic interfacing with the design under test. Statements are also included to declare the initial state of the design prior to a simulation, and to specify which components are to be grouped together to form a simulation data section.

2) Editing Statements

These are used to make on-line modifications to the design description file which the simulation system is currently working with. Statements and components can be added or deleted and names can be changed.

3) Simulation Commands

These statements are used to control the simulation system. They are used to translate the design description into a simulation data structure, initialize the Activity Queue, start the simulation, and specify conditions for termination.

4) Output Statements

These commands are used to generate output messages during a simulation and to print out the values of level signals after the simulation has terminated.

We will now introduce the proposed design language extensions and give some examples of their use. After this is completed, simulation procedures and techniques will be discussed.

A. Design Language Extensions

The following declarations are introduced to specify the circuit and signal parameters τ_A, τ_D and τ_C of Section III:

ambiguity τ_A , < list of levels, pulses and transfers > ;

min delay τ_D , < level list > ;

min complement τ_C , < register list > ;

Delay Ambiguity declarations can be made for any level, pulse (event), or transfer. In the case that no ambiguity declaration is made for an element, it is represented by an ideal model during simulation. If the designer wishes to have an element represented by a detailed model with no additive ambiguity, he must declare it with τ_A equal zero. A Minimum Delay declaration can be made for any level signal. Normally, level signals are represented by a bit on a Flip-Flop or Combinational Level String. If τ_D is declared for a level, a Delayed Level String bit is attached to its output and the value of the level is represented by the delay output. This is true for both ideal and detailed models. Minimum Complement Time declarations can only be made for registers and sub-registers. τ_C is assumed to be zero if undeclared.

Examples:

min complement 40, A[0:31], B[4:10] ;

min delay 20, C[0:31], D ≡ A[4] ∧ B[7] ;

ambiguity 15, A, D, A → C, osc1 ;

To properly test a design, it may be necessary to simulate interfaces with core memories, magnetic tape drives and other memory devices. The following declarations are introduced to specify fixed and variable size storage arrays:

memory < memory name > [< bit indices >, < number of words >
< address name > [< address indices >]];

stack < stack name > [< bit indices >,
< address name > [< address indices >]];

Memory declarations[22] are used to specify storage arrays with fixed size and word addresses. A declaration includes the array's name, bit indices (therefore word length), number of words, and the names of its address levels. Valid Memory addresses extend from zero to number of words minus one. Stack declarations are used to introduce variable length storage arrays. Words may be added to and deleted from the top of a stack during simulation. Addressing is done relative to the top of the stack. Stack declarations include their names, bit indices (word length), and the names of their address levels.

Memories and Stacks can be used as sources and destinations of transfer statements. Memories are specified in transfer statements the same way as registers are. The current values of the address levels determine which cell is used. Stack transfer specifications may contain additional arguments called address indexes. The address indexes are

[22] Similar to memory declarations used by Y. Chu in Reference (17).

added to the current values of the address levels to determine how far the cell to be used is from the top of the stack. If an attempt is made to access or modify a non-existent Stack or Memory Cell, an error message is transmitted to the system user.

The following statement types are used to add and delete words on the top of a stack:

push < stack name > < level expression and integer list >;

pop < stack name > < level expression or integer >;

Push statements cause the value of each level expression or integer to be placed on the top of the stack, beginning with the first on the list. If a level expression or integer is not the same length as the Stack's word length, the left-most bits are either filled with zeros or truncated. When pop statements are executed, the level expression or integer is evaluated and that number of words are deleted from the top of the stack.

The execution of Push and Pop statements generally changes the addresses of information already on the stack. Theoretically, a Stack may grow indefinitely during a simulation. In any given realization of the simulation system, there would be an upper bound on stack growth. Since the address levels and address indexes are specified in advance, only a finite number of cells from the top can be addressed. An attempt to Pop words from an empty stack will cause an error message to be reported to the user.

Examples:

```
memory      CORE[0:31, 1024, MAR[6:15]],
            INDEXREG[0:31, 64, IR[0:5]];

* read core into accumulator;

tp0:        0 ~ MAR;
tp1 ∧ RC:   if IR[0:5] = 0 then CAR → MAR
            else CAR + INDEXREG → MAR;
            0 ~ AC;
tp4 ∧ RC:   CORE → AC;
```

In the above example, the subregister CAR is the address portion of the instruction register. Bits 0 through 5 of the instruction register specify which of 64 index registers to use. Index register 0 always contains zero. Note that more than one Memory or Stack can be declared in a single statement.

In the following example, a Stack will be used to represent the magnetic tape in a rough simulation of an incremental magnetic tape drive. The tape drive can read or write forward or backspace by single eight-bit characters. The tape drive is driven by a thirty-two bit machine so all tape movements are in four character blocks. The register TDA is used in the simulation to contain the location of the read-write head relative to the last character written on the tape. When a new character is written by the tape drive, its erase head may destroy characters further down the tape, so no information beyond the last character written can be read. In the simulation the first thing done on a WRITE command is to delete this information.

```
* rough simulation of incremental tape drive;

register      TDA[0:15];
stack         TAPE[0:7, TDA];

* backspace tape WCT words (WCT previously declared on interface);

BACKSPACE:    TDA + 3 x WCT => TDA;
              0 ~ BACKSPACE;

* read 4 tape characters into buffer register;

READ:         0 ~ BR;
              TDA - 4 → TDA;
              delay 100;
              TAPE[0:7, 3] → BR[24:31];
              TAPE[, 2]    → BR[16:23];
              TAPE[, 1]    → BR[ 8:15];
              TAPE         → BR[ 0:7 ];
              0 ~ READ;

* write buffer register onto tape;

WRITE:        pop TAPE, TDA;
              0 ~ TDA;
              delay 100;
              push TAPE, BR[24:31], BR[16:23],
                         BR[ 8:15], BR[ 0:7 ];
              0 ~ WRITE;
```

The following statements are used to specify the initial state of a design before simulation begins. If register or memory contents are undeclared, their initial values are zero. Initial stacks are empty if their contents are not specified.

```
initialize    < register name > = < value > , ... ,
              < register name > = < value > ;

initialize    < memory name >, < address > : < value > , ... ,
              < value >, < address > : < value > , ... , < value > ;

initialize    < stack name >, < value > , ... , < value > ;

initialize    < delay name > = < value > , ... ,
              < delay name > = < value > ;
```

Level delays are initialized over their entire lengths - no transitions can be stored anywhere except at their input terminals. All flip-flop and combinational level values are assumed to be settled. The outputs of unitialized level delays take on the same values as their inputs. Although separate statement types are given for registers, delays, memories and stacks, there is no reason why these should not be intermixed in the same initialize statement.[23]

Example:

```
initialize    A = 7, CORE, 20 :  7654, 444312, 67334, B = 16,
              INPUTLIST,. 5, 4, 3, 2, 1, 0,  C = 41 ;
```

The following statement is used to declare which components are to be grouped together to form a simulation data section.

```
section    < component list > ;
```

If a component is later declared to be a part of another section or is deleted, it is removed from its present section. All components not declared in a section statement are assigned to section 0 and kept in core permanently during a simulation.

An important service to be provided by the command language is the ability to make on-line modifications to a design being simulated. The following editing statement types are suggested for this function.

---

[23] It is not obvious which number base should be used for simulation input and output. Because the hardware is simulated in such detail, there are strong arguments for base eight numbers because engineers find them convenient during logic debugging.

```
delete component    < component list > ;

delete              < component name > ;  < statement > ; ... ;
                          < statement > ;  end delete ;

add                 < component name > ;  < statement > ; ... ;
                          < statement > ;  end add ;

rename              < new name > / < old name > , ... , < new name > /
                          < old name > ;
```

The actions of these statements are fairly self-evident. Delete component statements are used to remove entire components. The second and third statement types are used to delete and add statements within some component description. A new component can be created by making the first statement on an add list a component declaration. An example of this is given below. Rename statements are used to change the names of components, levels and pulses.

Examples:

```
delete component    tapecontrol ;

delete              commoncontrol ;

sum:                A[0:2] → B[0:2] ;
                    C[0:3] → D[0:3] ;
end delete ;

add                 commoncontrol ;

sum:                A[0:2] → D[0:2] ;
                    C[0:3] → B[0:3] ;
end add ;
```

```
add                 inputoutputcontrol ;
component           inputoutputcontrol ;
interface           commoncontrol ;
                    .
                    .
                    .
end component       inputoutputcontrol ;
end add ;

rename              iocontrol/inputoutputcontrol, ccontrol/commoncontrol ;
```

We are now ready to introduce command statements which control the translation of a design description into a simulation data structure, and which initiate and terminate a simulation.

translate for simulation ;

end translate ;

start < pulse list > ;

stop ;

restart < pulse list > ;

singlestep < pulse list > ;

The Translate for Simulation command is used to cause the design description file which the simulation system is working with to be translated into a simulation data structure file. Once this command is given, all delete and add commands will automatically be executed on both the design description and the simulation data structure files until the End Translate command is given. After that, additions and deletions are made only to the design language file.

Use of a Start command causes the Activity Queue to be emptied and all pulses on the command's argument list to be placed on the Immediate Event List. All oscillator pulses are also placed on the Immediate Event List and all Clock Event pulses (discussed below) are added to the Activity Queue. Register and Memory values are set to zero and Stacks emptied. All Initialize statements are executed in the order which they appear in the design. All combinational levels and delay lines are evaluated, and simulated time is set to zero. Simulation commences from that point and continues until an error alarm is detected, the Activity Queue becomes empty, or the Terminate Transfer is executed. A Stop statement can be inserted in the design wherever a register transfer statement is legal. The simulation terminates whenever it is executed.

When a Restart command is given, the pulses on its argument list are added to the Immediate Event List and simulation commences. Singlestep commands behave the same way except that the simulation terminates whenever it moves forward to a new time with a non-empty Immediate Event List. The designer may use this command to single-step time when he is trouble-shooting a design problem.

Examples:

```
start              tp0, pulselin, carry2 ;
tp3:               if  AC = 0  then stop ;
CLOCK = 97943:     stop ;
```

The last example shows a Clock Event causing termination. The special symbol CLOCK refers to a register containing the current value of simulated

time and may only be used in statement labels of the above form. Clock Event pulses are placed in the initial Activity Queue when Start commands are given.

The next set of statements are used to generate output messages during a simulation:

print < level and text list > ;

tprint < level and text list > ;

text < text name >, < string of output characters without ";" > ;

Print and tprint statements are inserted into a design just as stop statements. Whenever one of them is executed, an output message is generated for the user. Tprint includes the value of the simulated clock along with the message. Messages consist of the new values of level signals and fixed character strings specified with Text declarations. The following example illustrates how the output statements might be used to trace the usage of blocks of hardware.

```
* Trace on fixed point multiply instruction

text          m1,  Multiply Arguments Are ;
text          m2,  Product Is ;

MULTIPLY:     tprint   m1, AC, BR ;

¬ MULTIPLY:   tprint   m2, CA ;
```

In this example, when the signal MULTIPLY comes on, the AC and BR registers are multiplied together. The product ends up in the CA register before MULTIPLY is turned off. With the trace included in the simulated design, the following kind of messages are presented to the user every time the multiply instruction is used:

```
T = 47800      Multiply Arguments Are   AC = 440  BR = 100.
T = 47920      Product Is  CA = 4400.
```

The following special forms of the Print command can be used to output the contents of the Activity List. This is especially valuable when simulation terminates because of detection of an error alarm.

```
print  AL   ;   * entire Activity List ;
print  AEL  ;   * all Event Lists attached to Activity List ;
print  ADVL ;   * likewise for Delay Value Lists ;
print  ARL  ;   * likewise for Reset Lists ;
```

The following statements are introduced to allow the user to specify simulation alarm conditions which he is not interested in detecting. The statements instruct the simulator not to stop on the specified alarms or report them to the user.[24]

```
suppress   hazard alarm   < list of differentiated levels > ;
suppress   sampling alarm   < list of level/sampling pulse pairs > ;
suppress   undefined transfer   < list of transfer/activating pulse pairs > ;
suppress   register input alarm   < register list > ;
```

Examples:

```
suppress   hazard alarm      SYNC1,  AC = 256 ;
suppress   sampling alarm      AC[4], tp1, BR[10], tp2 ;
suppress   undefined transfer     AC → BR, tp5 ∧ (IR = 410) ;
suppress   register input alarm      IOR, PBR, MSK[4:10] ;
```

[24] The meaning of these alarm conditions is pointed out in Appendix B. Note that illegal Memory and Stack address, empty Stack, and register minimum complement time alarms are not suppressable because these are considered modeling errors.

B. Logic Testing Procedures

We are now prepared to discuss the general procedures which a logic designer might use to test his design by simulation. The first thing he would do is use the regular computer system software to create or retrieve a design description file for the simulation system to work in. The design in this file may contain more components than he wishes to simulate, so he removes them with Delete commands. Simplified models for the deleted components which interfaced with the remaining design can be created using Add commands. Design language models may also be introduced for external devices which interface with the design, but were not included in the original file. The designer may wish to change some of the design's timing parameters with Add and Delete commands.

If the designer is interested in checking for the occurrence of certain conditions during simulation, he may add special logic to the design to detect them. The special logic can initiate output messages or even terminate the simulation, if he wishes. This technique amounts to including special debugging logic in the simulated design. Dummy differentiators may be placed on interface level signals to detect level hazards. Interface pulses may be used to complement dummy flip-flops to detect pulse doublets closer together than the minimum complement times of the dummy flip-flops.

Additional Print and Tprint statements may be included to output information during the simulation. Register and storage array initial state declarations may be changed. Declarations are made stating which components are to be grouped together to form simulation data sections. When the designer is satisfied that the design language description is

complete, he causes it to be translated into a simulation data structure file by issuing a translate for simulation command. He then gives a start command to initialize the Activity Queue and begin the simulation.

If the simulation terminates due to an error alarm, the designer will investigate the design description and the state of the simulated design to determine if the alarm is valid or the result of improper modeling of delay parameters. Print commands can be given on-line to interrogate the system about level values and the Activity List contents when simulation terminated. If the designer decides that the design was improperly modeled, he changes some of the delay time parameters and issues either a Start or a Restart command. If he is unable to determine what the problem is, he may insert more debugging logic into the design and/or single-step part of the simulation. If he wishes, he can give register transfer commands on-line when simulation is not active.

After he has isolated a problem, he may wish to simulate some alternate solutions. The regular computer system software can be used to store copies of the present design description and simulation data structure files away for future use while he tries the alternatives. When he finds an alternate he likes, he may make changes to the original design description file. When he is tired or wants to think about a problem he can save his files to be retrieved when he returns.

VII. CONCLUSIONS

The work reported in this thesis has been directed toward the development of a logic simulation system for design verification. The system would accept the Dennis Design Language as input and operate on-line in a large time-shared computer environment. It would serve as a design tool to interact with an intelligent designer during the design process, rather than being an automatic process which exhaustively checks out a design.

The idealized model simulation system discussed in Section IV has advantages over other such systems found in the literature. Simulation efficiency is greatly improved because combinational levels are re-evaluated only when their values are required, and may have changed since last evaluated. This advantage tends to increase with the size of the design being simulated. Flip-flop input error detection helps locate many of a system's solid design errors. Synchronous and asynchronous designs can equally well be simulated. The internal data structure of the simulator is a direct representation of a logically sufficient, but not minimal, subset of the design language. Translation from the complete design language into this subset and from there into the data structure should not be too difficult. Likewise, the translation of output information into terms of the input description should present no difficulties. It would not be unreasonable for early versions of the simulation system to require designers to use the subset of the design language directly represented in the data structure to describe their designs. The data structure was formulated so that incremental changes could be made to it rather than retranslating the

entire structure when a small modification is made to the source description. It is acknowledged that an incremental translator would be complicated and that there are difficult problems that must be solved in order to construct one.

Important factors considered by an engineer evaluating the performance of a design are whether or not the interface behavior is as specified and whether or not it is deterministic over the specified range of operating conditions. The internal behavior of the design need not be deterministic and logic designers may take advantage of that fact to maximize the design goals.[25] An example of this is the case where the value of some flip-flop is not used by an instruction and will be cleared before the next instruction. The designer may be able to reduce the cost of implementing the instruction by using logic which causes the value of that flip-flop to be non-deterministic.

If the internal behavior of a design is deterministic, obviously its interface behavior must also be deterministic. Signal spread was introduced to the data structure as an attempt to check whether or not the behavior of each element of a design is deterministic. The spread of a signal transition or event activation is the range of its occurrence time probability density function. The detailed model simulation system

---

[25] Unfortunately there are cases where designers have violated wiring rules or have taken advantage of special circuit characteristics, which are not checked during circuit testing, to minimize cost or maximize speed. Such practices are ill-advised because they tend to reduce system reliability.

uses signal spreads to detect the sampling of a changing level by a pulse or ambiguous flip-flop outputs due to multiple simultaneous input pulses. The circuit models are considered adequate for these purposes. The weakness in the modeling results from considering each signal spread to be independent, when the occurrence time density functions are actually dependent. This results in flagging correct situations as logic faults. It is felt that the system is still useful because these incorrect error detections can be eliminated by remodeling or error message suppression.

Acceptable methods were introduced to the simulation data structure and algorithm for detecting level hazards, intermixing idealized and detailed models in the same structure and partitioning data structures so that more complex machines could be simulated. Extensions were proposed to the design language to furnish special information needed as input to the simulation system and so that the language could also be used as the on-line simulation command language.

There are a number of ways of extending or improving the work reported here. The first of these would be the development of a more satisfactory method of checking for non-deterministic logic behavior. A more general approach to modeling signal spread, which takes account of spread dependencies, is to make restrictions on either the number of levels of signal interdependence (how many logic stages of past signal history to be maintained by the model) or the class of designs to be simulated. The latter restrictions would be used to eliminate the requirement of indefinitely long signal histories for completely accurate models of certain pathological networks.

Further work to be done includes the translation of the data structures and rough algorithm outlines discussed here into a computer program to be implemented on Project MAC's new MULTICS system, when it becomes available. This will require a routine for translating the directly realized subset of the design language into the simulation data structure. If the complete design language is to serve as input to the simulation system, a pre-processor for translating it into the accepted subset must be written. The non-trivial problems involved with incremental translation of corrections must be solved if that feature is to be included in the simulation system.

A topic of considerable interest is that of providing adequate provisions for outputting information to the designer during a simulation. The output commands introduced in Section VI are comparable with simulation output techniques found in the literature. If these commands were used to any great extent, the simulation system would become severely output limited and the advantages of on-line operation would be lost. This seems to be a fruitful application area for displays. For example, a display might be used to simulate a computer control panel[26] which could be modified on-line. A by-product of this approach might be the development of more functional computer operator and maintenance console designs.

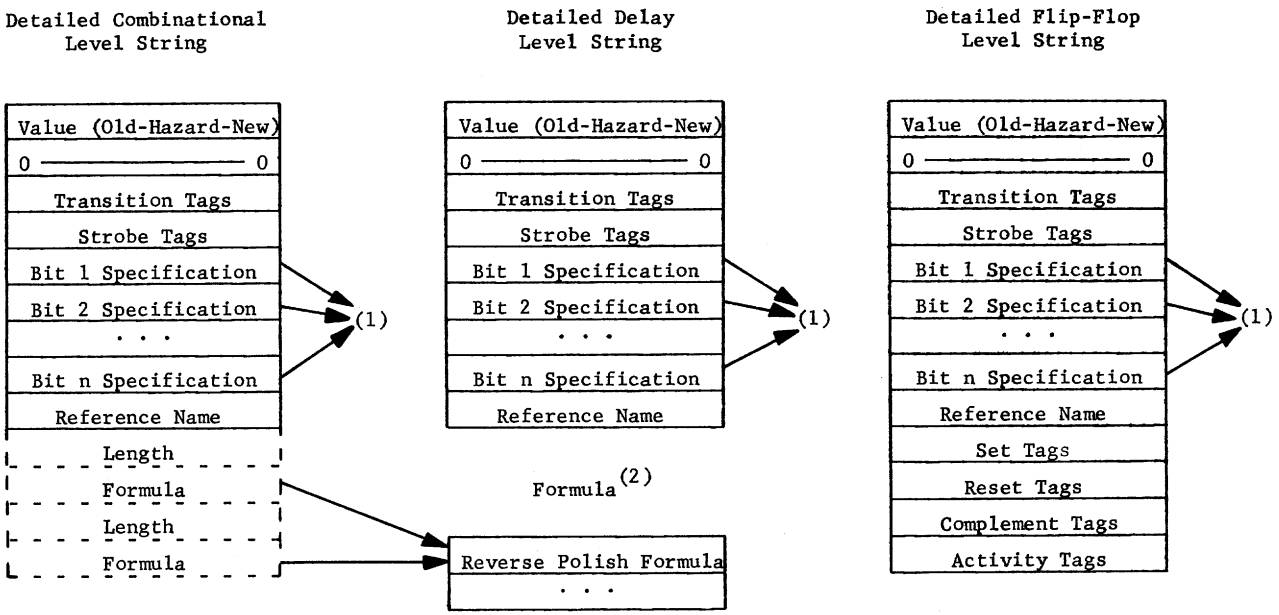
---

[26] This is similar to the printed simulation output discussed in Reference (1).

APPENDIX A - INTERMIXED SIMULATION DATA STRUCTURE

This appendix consists of a set of illustrations of the intermixed data structure of Section V. The conventions used are the same as the ones used for the Ideal data structure illustrations in Section IV. Note that Ideal and Detailed Level String elements are the same size to make it simple to change the model to be used for a given level circuit. A level string can contain representatives of both models in any intermixed order. There are a variety of ways of representing the circuit and signal models in a data structure. The only claim made about the structures shown here is that they seem reasonable for the design goals of Section IV. The Tag Activity List in Figure A-3, the data portion of the Stack in Figure A-4, and all of Figures A-6 and A-7 are shown as list structures because their lengths are modified during a simulation. The criterion for choosing whether or not to represent other structures with list structures or fixed blocks was the likelyhood of their lengths being modified when a designer modifies his design. There is a trade-off between the ease of incremental modification of the data structure to represent design changes, and the access speed of information during a simulation. If incremental modification of the data structure is not done, all structures which remain of fixed size during a simulation should be realized as fixed blocks.

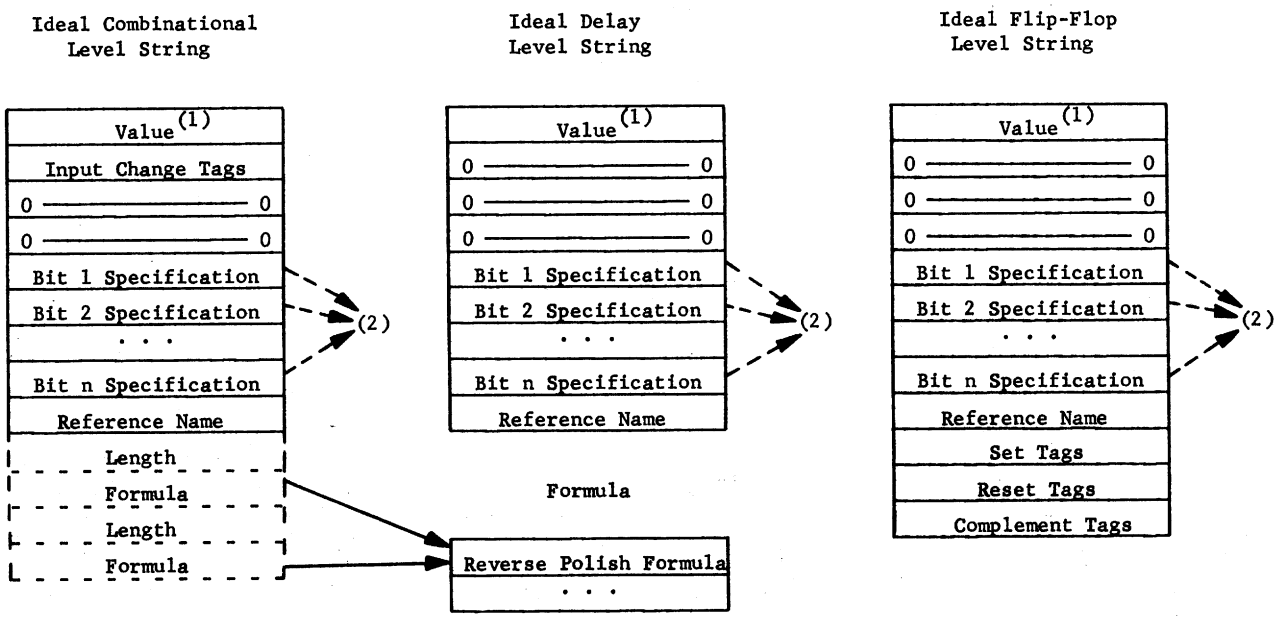
Footnotes are used with the figures to clarify the meaning of some of the cell contents and to point out interrelationships between figures. Appendix B outlines a simulation algorithm based on this data structure.



(1) Pointers to Output Specification element of Figure A-3.
(2) Refer to Appendix C for discussion of formulas.

FIGURE A-1

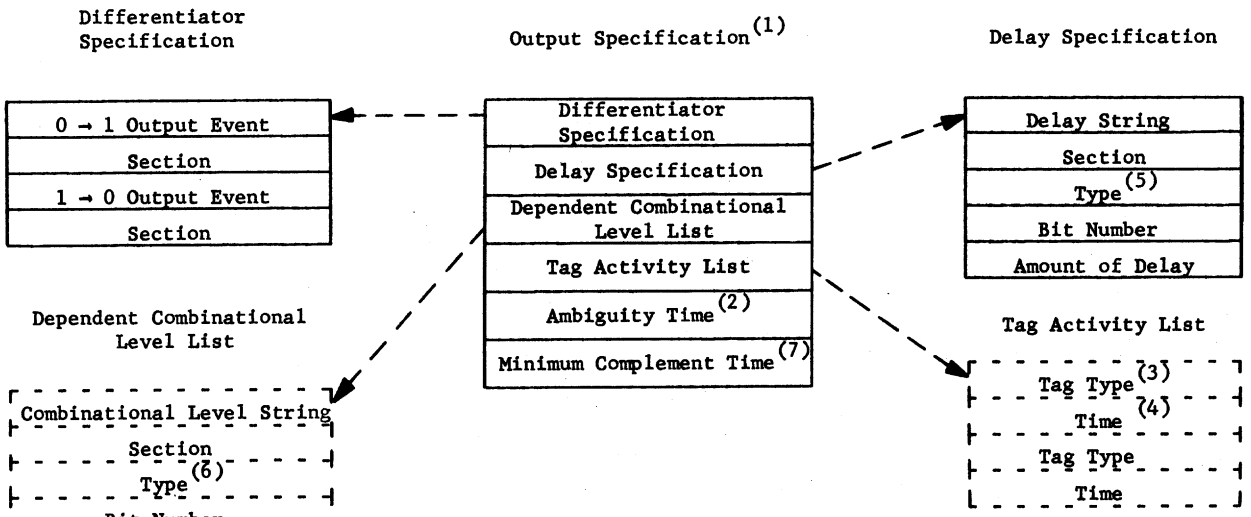
Detailed Level String Data Elements



(1) Old Values and Hazard Values not carried.
(2) Pointers to Output Specification element of Figure A-3.

FIGURE A-2

Ideal Level String Data Elements



(1) Pointed to from Level String elements of Figures A-1 and A-2.
(2) Always special value Z for ideal outputs.
(3) Transition, Strobe, Set, Reset, Complement, Activity or Hazard Value.
(4) Pointer to Time cell on Activity Queue of Figure A-6.
(5) Specifies whether or not Hazard Values must be carried.
(6) Specifies whether or not a new value must be computed for dependent level when the output changes.
(7) Included only for detailed flip-flops.

FIGURE A-3

Level Output Specification Structure

Constant Level String[1]

Value
0 ——— 0

Memory[2]

Address Length
Address Level String
Section[3]
Type
High Order Bit Number
Maximum Address, m
Reference Name
Word 0
Word 1
. . .
Word m

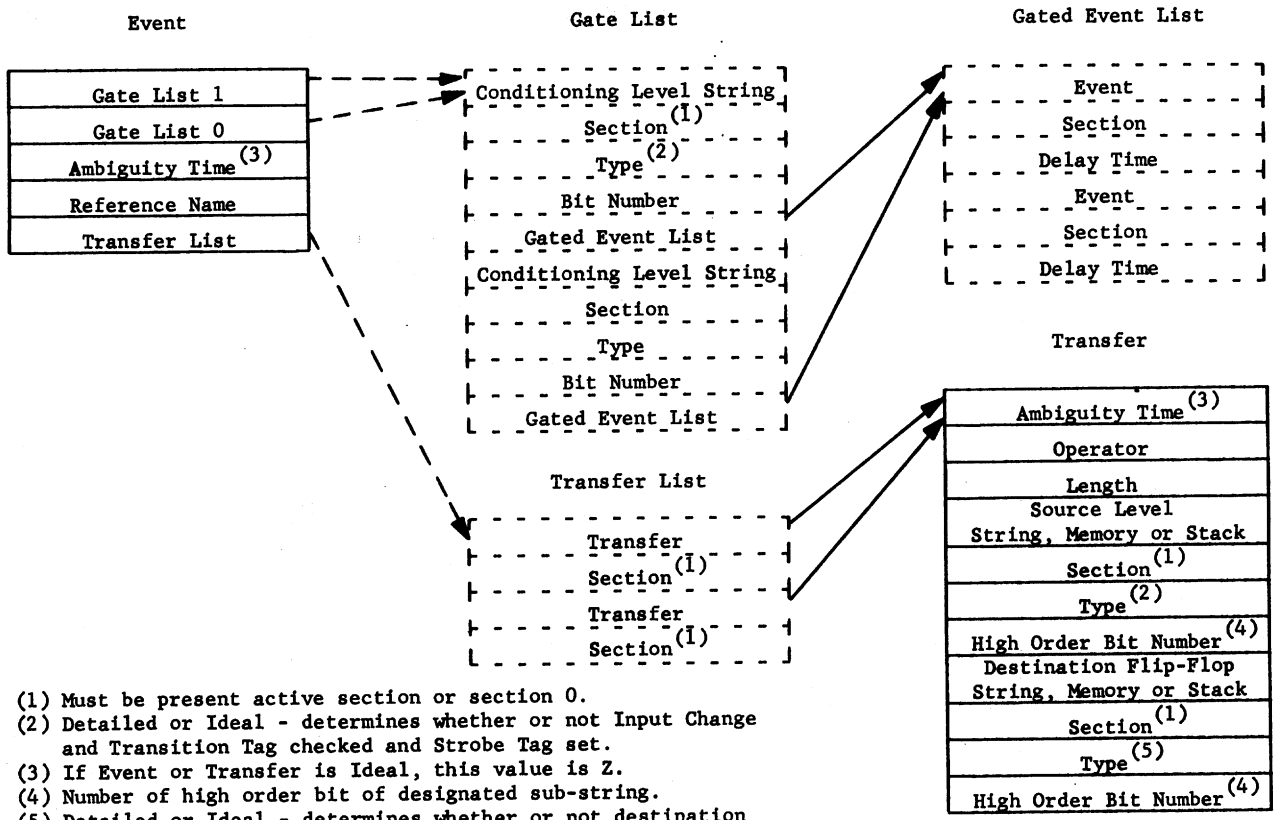
Stack[2]

Address Length
Address Level String
Section[3]
Type
High Order Bit Number
Maximum Address, m
Reference Name
Word 0
Word 1
. . .
Word m

(1) Note that Constant Level Strings do not require Transition or Strobe Tags because they are Ideal models. They require no Output Specifications because their values never change.
(2) Used to simulate special interfaces and to store output data.
(3) Must be some section as Memory or section 0.

FIGURE A-4

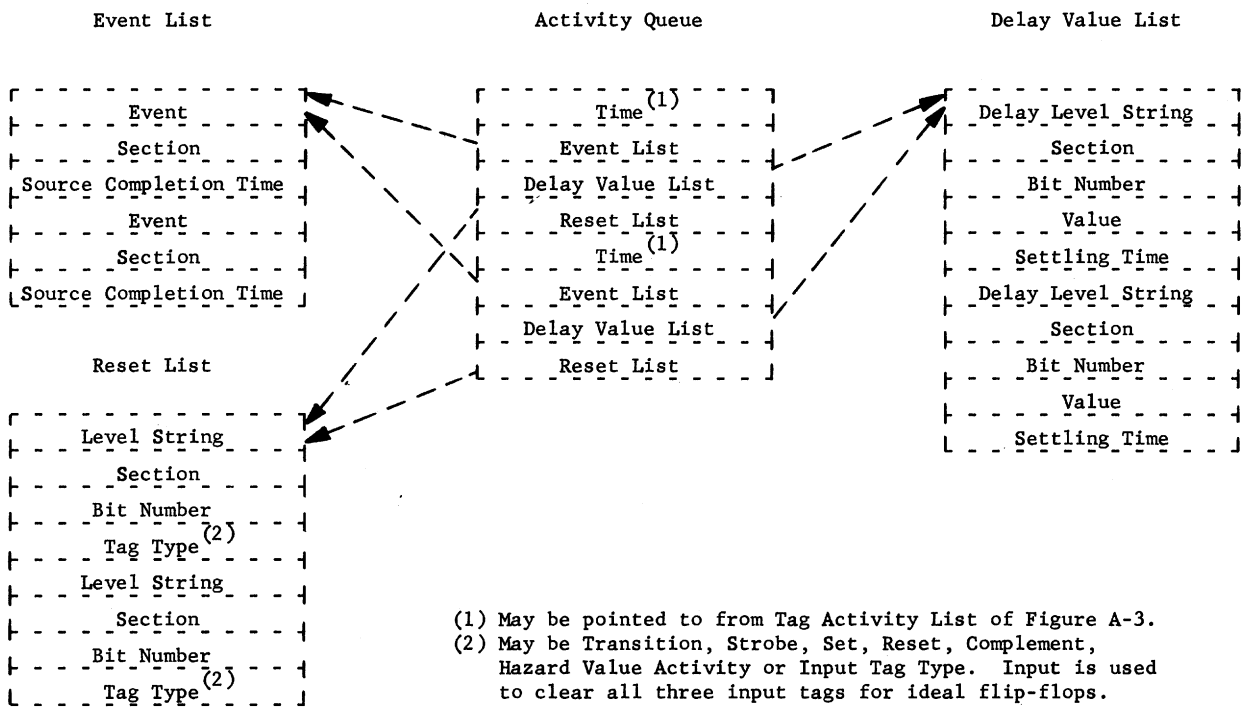
Constants, Memories and Stacks



(1) Must be present active section or section 0.
(2) Detailed or Ideal - determines whether or not Input Change and Transition Tag checked and Strobe Tag set.
(3) If Event or Transfer is Ideal, this value is Z.
(4) Number of high order bit of designated sub-string.
(5) Detailed or Ideal - determines whether or not destination Set, Complement and Reset Tags should have spread.

FIGURE A-5

Control Logic Data Elements



(1) May be pointed to from Tag Activity List of Figure A-3.
(2) May be Transition, Strobe, Set, Reset, Complement, Hazard Value Activity or Input Tag Type. Input is used to clear all three input tags for ideal flip-flops.

FIGURE A-6

Activity Queuing Data Structure

Section Input List[1]

Level String
Bit Number
Level String
Bit Number

Detailed Flip-Flop List

Detailed Flip-Flop String
Section
Bit Number
Transfer Completion Time
Detailed Flip-Flop String
Section
Bit Number
Transfer Completion Time

Ideal Flip-Flop List

Ideal Flip-Flop String
Section
Bit Number
Ideal Flip-Flop String
Section
Bit Number

Active Level List

Level String
Section
Bit Number
Level String
Section
Bit Number

Ideal Input Tag List

Ideal Flip-Flop String
Section
Bit Number
Ideal Flip-Flop String
Section
Bit Number

(1) One of these lists is provided for each non-permanent data section. All pointers entered on these lists are for interface signals found on section 0.

FIGURE A-7

Temporary Storage Lists

APPENDIX B - INTERMIXED DATA STRUCTURE SIMULATION ALGORITHM

In this appendix we shall discuss a simulation algorithm based on the intermixed data structure discussed in Section V and illustrated in Appendix A. This will be done in a format similar to that used in Section IV to discuss the simulation algorithm based on idealized circuit and signal models.

The center of a simulation is the Activity Queue, which is a time-ordered list of all future activity for the simulator to undertake. The structure of the AQ (Figure A-6) consists of a list of time cells containing progressively larger values of time. Each time cell has a pointer to an Event List, a Delay Value List and a Reset List. The Event List contains the locations (addresses and section numbers) of the Events which the simulator knows must be activated when simulated TIME reaches the value contained in the time cell. An Event's time of activation is the earliest possible time at which it could occur (starting time of Section III). Listed with an Event's location on the Event List is a source completion time. The Event's settling time, or the latest possible time at which it could occur, is found by adding its ambiguity time (contained in the data block describing it) and its source completion time.

The Delay Value List contains the locations (string addresses, bit numbers and sections) of all level delays whose output values the simulator knows will change when simulated TIME reaches the value contained in the time cell. Accompanying each level delay's location is its new value and a settling time. This settling time is added to the delay's

ambiguity time to determine the settling time of the output change. Likewise, the Reset List contains the location (string addresses, bit numbers, sections and tag types) of all status tags which the simulator knows must be reset when simulated TIME reaches the value contained in the attached time cell.

When the simulator determines that an Event is to be activated, it adds the Event's location and source completion time to the (possibly empty) Event List attached to a time cell containing the Event's starting time. If no such time cell exists, one is created and placed on the AQ in the proper time position. Likewise when the simulator finds that a level delay value should change, or a status tag should be reset, entries are added to Delay Value and Reset Lists attached to the correct AQ time cells. Once an Event is placed on an Event List it is not removed until activated, but the source completion time portion of the entry may be modified. Likewise Delay Value List entries are not deleted until activated, but the new values and settling times may change. Reset List entries may be deleted before activation time. During the execution of a simulation the simulator may attempt to set a status tag and find it already set. When this happens the simulator must fetch the tag's old reset time and compare it with the newly computed one. If the new reset time is chosen, the old Reset List entry is deleted and a new entry is added to the Reset List attached to the AQ time cell containing the new reset time.

Each status tag is part of the description of some level. When an entry for that tag is added to or deleted from a Reset List, a matching entry is also added to or deleted from the Tag Activity List attached to

the level's Output description (Figure A-3). The matching entry consists of a cell containing the status tag type (Transition, Strobe, Set, Reset, Complement, Hazard or Activity) and a pointer to the AQ time cell which the Reset List is attached to. The simulator uses the Tag Activity List to fetch old tag reset times when comparisons are required, and to locate old Reset List entries when they are to be deleted.

The simulator operates under the control of another program which translates user commands, formulated in the extended design language of Section VI, into the data structure. This program, which we will call the translator, also translates outputs from the simulator into messages for the user. A simulation begins with an initial state for all level signals and an initial Activity Queue based on instructions from the user. The top time cell on the AQ is called the CLOCK and contains an initial value of zero. During the rest of this appendix the value of the CLOCK, or top time cell, is referred to as simulation TIME. When all of the entries attached to the top time cell have been activated the cell is deleted, stepping the CLOCK. The simulation stops when the simulator steps the CLOCK and finds the AQ empty or the Simulation Termination Tag set. This tag is set when the Terminate Transfer is executed or when the simulator detects and informs the translator about a simulation alarm. The Event List attached to the top time cell on the AQ is called the Immediate Event List. It contains those Events which are currently active. Likewise the Immediate Delay Value List and Immediate Reset List contain value changes and tag resets which are currently active.

The simulation data structure is divided into a number of data sections. At any given time, the simulator will be working with section 0 and one of the other sections, called the "non-permanent" sections in Section V. The non-permanent section being worked at a given time is called the Active Section and is selected by a round robin process. When simulation begins an Active Section is selected. All value changes in the Immediate Delay Value List for section 0 and the Active Section are activated. All status tags on the Immediate Reset List for section 0 and the Active Section are reset. All Events on the Immediate Event List for section 0 and the Active Section are activated. The activation of an Event may cause new Events to be added to the Immediate Event List. The process is continued until there are no more Immediate Event List entries for section 0 and the Active Section.

At this point all level changes brought about by the Event activations and the level delay value changes are propagated through the logic. This will cause more entries to be added to the AQ; in particular, more Events may be added to the Immediate Event List. Any of these belonging to section 0 and the Active Section are activated and their changes are propagated through the logic. This is continued until no new Events are added to the Immediate Event List for these sections. The propagation of level changes through the logic will generally cause section interface levels to change. Each non-permanent data section has a Section Input List (Figure A-7). When an interface level changes, its location is placed on the Section Input List of its destination non-active section. When this section becomes active, the level changes on its Section Input List are propagated through the logic.

After all immediate activity for section 0 and the Active Section has been completed, the next section on the round robin is made active. All status tags on the Immediate Reset List are reset and immediate delay value changes are activated for the new Active Section. All Events on the Immediate Event List for the Active Section are activated. The level changes brought about by the changing flip-flop, delay, and interface levels are propagated through the logic. Additional Events entering the Immediate Event List for section 0 and the Active Section are activated, etc. until no new ones are added.

This process is continued on around the sections until all immediate activity has been completed; i.e., the immediate lists and all Section Input Lists are empty. The simulator is then ready to step the CLOCK. First it must make arrangements for the ideal flip-flop input tags which were set during the last instant of time to be reset. This is done by adding a number of entries to the Tag Reset List attached to the next AQ time cell. The Simulation Termination Tag is then tested to see if it has been set. If so, control is passed on to the translator. Otherwise the CLOCK is stepped and the simulator begins over with the new set of immediate activity lists.

During the processing of an instant of simulated time, the simulator uses a number of lists to store temporary information (Figure A-7). The Section Input Lists were mentioned above. An Ideal Flip-Flop List is used to keep track of ideal flip-flops with active inputs. A Detailed Flip-Flop List serves the same function for detailed flip-flops. An Active Level List is used to remember level signal changes which have not

yet been propagated through the logic. Finally, there is an Ideal Input Tag List which is used to record ideal flip-flops whose input tags will require resetting before stepping the CLOCK.

1. Outline

The following is a fairly detailed outline of the simulation algorithm discussed above. The phrases "the translator is informed that ... alarm has occurred" and "output information is passed on to the translator" occur several times in the outline. These phrases do not imply that the simulation is interrupted for these output messages, but rather that they are added to an output buffer. The simulator may pass control over to the translator to process these messages at regular intervals and whenever the output message buffer becomes full.

Before a simulation begins, all delay line, flip-flop and detailed combinational level values are initialized and the Input Change Tags for all idealized combinational levels are set. The Activity Queue is initialized and all other lists of temporary information are emptied. The first section to become the Active Section is the lowest numbered one other than zero.

I. The next section on the round robin becomes the Active Section.

A. All Immediate Reset List entries for section 0 and the Active Section are activated and deleted from the list.

1) This causes various Transition, Strobe, Set, Reset, Complement and Activity Tags to be reset for detailed level bits.

2) If the Tag Type entry (Figure A-6) is Input, all three flip-flop input tags (Set, Reset and Complement) are reset. Input Tag reset entries are only used to clear ideal flip-flop input tags.

3) A special Tag Type (Figure A-6) is reserved for detailed flip-flop Transition Tags. When a detailed flip-flop's Transition Tag is reset, its New Value replaces its Old Value, its Hazard Value is reset, and an entry is added to the AQ to cause its Activity Tag to be reset. (See V - B (3) below.)

4) If a flip-flop Hazard Value bit is reset, the flip-flop's location (level string, bit and section number) is added to the Active Level List. The flip-flop's Output Specification is checked to see if it feeds a level delay. If it does, an entry is added to an AQ Delay Value List to cause the level delay value to make the same change at TIME plus the amount of delay listed in the Delay Specification (Figure A-3). The reset time for the flip-flop's Transition Tag plus the amount of delay is used as the source completion time on the new entry. (See V - B (4) below for special case which causes a Hazard Value bit reset entry to be added to the AQ. This is an optional feature.)

B. All level locations (strings, bit and section numbers) on the Active Section's Section Input List are added to the Active Level List, and the Section Input List is emptied. These levels are all interface signals and therefore are on section 0.

II. The value changes for section 0 and the Active Section listed on the Immediate Delay Value List are executed.

A. The settling time for each change is calculated by adding the delay line's ambiguity time to the settling time included in the Delay Value List entry (Figure A-6).

1) The delay line's Transition Tag is set. An entry is added to the AQ to cause it to be reset at the delay line's settling time. If the Transition Tag is already set, its old resetting time on the AQ is replaced with the new one.

2) If the delay line ambiguity time is #, the Transition Tag is not set and the delay line settling time equals TIME, the current value of CLOCK.

B. If the delay line's value or settling time has changed, its location (delay string, bit and section number) is added to the Active Level List.

1) The delay's Output Specification Block is checked to see if it feeds a second level delay line. If so, an entry is added to the Delay Value List attached to the AQ time cell containing TIME plus the amount of delay listed on the Delay Specification (Figure A-3). The added entry consists of the location of the second delay line (string, bit and section), the current Hazard and New Values of the input delay line and a settling time equal to the sum of delay of the second delay line and the settling time of the input delay line. If the second delay line is ideal, the Hazard Value in this entry is zero no matter what the input delay line's Hazard Value might be. If another entry is already present on the same Delay Value List for the same level delay bit, the old entry is deleted.

2) If the level delay's New Value has made a transition, its Output Specification is checked to see if it feeds a differentiator sensitive to this transition. If it does, the differ-

entiator's output Event is added to the Immediate Event List. The settling time of the level delay is taken as the Event's source completion time (Figure A-6).

3) If the level delay's Hazard Value has changed to one and it feeds a differentiator, the translator is informed that a Hazard Alarm has occurred and is given the Reference Name and bit number of the level delay. Any differentiator fed by the level delay and not activated in (2) above is also activated.

C. Each value change entry is removed from the Immediate Delay Value List as it is activated.

III. The section 0 and Active Section Events on the Immediate Event List are activated one at a time and deleted from the list until no more remain.

A. An Event is removed from the list when it is activated. It's completion time is computed by adding its ambiguity time to the source completion time accompanying it on the Immediate Event List. If the Event has Ø ambiguity time, TIME is taken as its completion time.

B. All conditioning levels on the activated Event's Gate Lists are tested (Figure A-5). All Event locations contained on Gated Event Lists attached to conditioning levels with correct values (1's on Gate List 1 and 0's on Gate List 0) are added to the AQ. The activation time of each of these Events, or the value of the AQ time cell to which the entries will be attached, is calculated by adding TIME to the delay time accompanying the Event on the Gated Event List. The source completion time of each of these Events is computed

by adding the same delay time to the completion time of the presently activated Event.

1) When an ideal level is tested, its Input Change Tag is checked to see if it needs to be re-evaluated (this tag can only be set for combinational levels). If so, it may be dependent on other ideal combinational levels which may also need re-evaluation. Therefore, the subroutine which evaluates ideal combinational levels should be recursive.

2) When a detailed level is tested by an Event or Transfer with non-zero signal spread (completion time not equal to TIME), the level's Strobe Tag is set. A Reset List entry is added to the AQ so that the tag will be reset at the sampling Event or Transfer's completion time. Thus a detailed level's Strobe Tag is set during the interval of time when it might be sampled.

If the Strobe Tag is already set when the level is tested, the tag's previous reset time is compared with the completion time of the sampling signal. If the previous reset time is earlier, it is replaced by the completion time.

The Old Value is the one always used when testing detailed levels.

3) If the Transition Tag of a detailed level being tested is set, the translator is informed that an Event Sampling Error has occurred and is given the Reference Name of the sampling Event and the Reference Name and bit number of the level.

C. The Transfers on the activated Event's Transfer List are executed. The completion time for each is the sum of its ambiguity time and the completion time of the activated Event. If a Transfer's ambiguity time is ∅, TIME is taken as its completion time.

1) Ideal Transfer sources may require re-evaluation as in B (1) above.

2) Strobe Tags are set for detailed Transfer sources as in B (2) above.

3) If the Transition Tag of a detailed Transfer Source is set, the translator is informed that an Undefined Transfer alarm has occurred and is given the Reference Name and bit number of the level and the Reference Names of the Transfer and activated Event.

4) If the transfer source is a memory, the memory's address value is checked to make sure that it is no greater than m, the maximum cell address. If the address is too large, the translator is informed that an Illegal Memory Address alarm has occurred and is given the Reference Names of the memory, the transfer, and the activated Event.

5) If the transfer source is a stack, a similar test is made on its address value. If it is out of bounds, the translator is informed that an Illegal Stack Address alarm has occurred and is given the Reference Names of the stack, the transfer, and the activated Event.

6) If the transfer destination is a memory or stack, the above address tests are made and the transfer is executed.

7) If the transfer destination is an idealized register, flip-flop Set, Reset and Complement Tags are set for those bits whose source values are correct. If an attempt is made to set a Complement Tag when it, or one of the other two tags, is already on, the translator is informed that a Register Input Error has occurred and is given the Reference Name and bit number of the flip-flop and the Reference Names of the Transfer and activated Event. The same thing is done if an attempt is made to set a Set or Reset Tag when one of the other two tags are on. When a flip-flop's first input tag is set, its string and bit number is added to the Ideal Flip-Flop List.

8) If the transfer destination is a detailed register, flip-flop Set, Reset and Complement Tags are also set for those bits whose source values are correct. A Register Input Error is reported to the translator under the same conditions discussed in (7) above. In addition, if an attempt is made to set a flip-flop's Complement Tag while its Activity Tag is on, the translator is informed that a Minimum Complement Time Error has occurred and is given the Reference Name and bit number of the flip-flop and the Reference Names of the Transfer and activated Event. When a tag is set for the first time since last reset, an entry is added to the AQ to cause it to be reset at the transfer's completion time. Subsequent attempts to set the tag result in a comparison of its old reset time and the completion time of the transfer. If the old reset time is earlier, it is replaced with the completion time.

If a detailed flip-flop's three input tags are all off and a transfer sets one of them, the flip-flop's location (string, bit and section) and the completion time of the transfer are added to the Detailed Flip-Flop List. If the first tag to be turned on is the Set or Reset Tag, subsequent attempts to set the same tag before setting one of the others will also add an entry to the Detailed Flip-Flop List, provided that the completion time of the transfer is less than the reset time for the tag. When a group of pulses are resetting a flip-flop, the flip-flop must be reset by the minimum of the completion times of the pulses. Therefore an entry is added to the Detailed Flip-Flop List when a new input is activated with a completion time which may be earlier than the previous active inputs of the same type. Once an input error is detected, no more entries are added to the Detailed Flip-Flop List. The detailed flip-flop evaluation routine (discussed in V below) will not change the output value or settling time of a flip-flop with an input error because its new value is ambiguous.

9) If a Push or Pop Transfer is activated, the necessary cells are added to or deleted from the specified stack and the maximum stack address is re-evaluated. If an attempt is made to Pop words from an empty stack, the translator is informed that an Empty Stack Pop error has occurred and is given the Reference Names of the stack, the Pop Transfer, and the activated Event.

10) If an Output Transfer is activated, the specified output information is passed on to the translator.

11) If the Terminate Transfer is activated, the Simulation Terminate Tag is set and a Terminate Message is passed on to the translator including the Reference Name of the activated Event.

IV. New output values are computed for the flip-flops listed on the Ideal Flip-Flop List.

A. The input tags are not reset. If more than one input tag is set for the same flip-flop, its output value is unchanged. Otherwise the Set Tag causes it to become 1, the Reset Tag 0, and the Complement Tag causes the value to complement.

B. If a flip-flop's output value changes, its location is added to the Active Level List.

1) The flip-flop's Output Specification is checked to see if it feeds a differentiator sensitive to this transition. If it does, the differentiator's output Event is added to the Immediate Event List. TIME is taken as the accompanying source completion time.

2) The Output Specification is also checked to see if the flip-flop feeds a level delay. If it does, an entry is added to an AQ Delay Value List to cause the level delay value to make the same change at TIME plus the amount of delay listed in the Delay Specification (Figure A-3). The accompanying settling time is also the sum of TIME and the amount of delay listed in the Delay Specification. If another entry for the same level delay bit is already present on the same Delay Value List, the old entry is deleted.

V. New output values are computed for the flip-flops listed on the Detailed Flip-Flop List.

A. If the Detailed Flip-Flop List contains more than one entry for the same flip-flop, all but the entry with minimum completion time is deleted.

B. The new output value for each flip-flop is calculated and its entry is removed from the Detailed Flip-Flop List.

1) The Old Value bit is unchanged while the New Value bit is calculated the same way as the new value of an ideal flip-flop in IV - A above.

2) If the flip-flop's New Value bit changes, its Transition Tag is set if previously off. An entry is added to the AQ to reset the tag at the flip-flop's new settling time. This is calculated by adding the input completion time listed on the Detailed Flip-Flop List to the flip-flop's ambiguity time. Thus the Transition Tag is on whenever the flip-flop's value might be changing, and therefore ambiguous.

3) When a flip-flop's Transition Tag is first turned on, its Activity Tag is also set. When the Transition Tag is reset, as in I - A (3) above, an entry is added to the AQ to reset the Activity Tag τ_C time units later, where τ_C is the minimum complement time of the flip-flop. Therefore the Activity Tag is on whenever it is illegal to strobe the flip-flop's complement input. If the activity Tag is already on when the Transition Tag is turned on, the Activity Tag's reset entry (there must be one) is deleted from the AQ. This technique for resetting

the Activity Tag is used because it is not necessary to change Activity Tag reset entries every time the flip-flop's settling time changes.

4) If the flip-flop's Transition Tag is already on when its New Value bit changes, the Hazard Value bit is set. The flip-flop's old settling time (the time when the Transition Tag is reset) is compared with its new settling time. If the old settling time is earlier, it is replaced by the new one and an entry is added to the AQ to have the Hazard Value bit reset at the old flip-flop settling time. (This feature is optional.)

5) If the flip-flop's Transition Tag is already on and the New Value bit is unchanged, the new settling time is compared with the old one. If the new settling time is earlier, it replaces the old one. If the Hazard Value bit is on, the Transition Tag reset time can be no earlier than the Hazard Value bit reset time (see 6 above). If no reset time is listed for the Hazard Value bit, the Transition Tag reset time is not changed.

C. If a flip-flop's Hazard Value bit, New Value bit, or settling time has changed, its location (string, bit and section) is added to the Active Level List.

1) The flip-flop's Output Specification is checked to see if it feeds a level delay. If it does, an entry is added to an AQ Delay Value List to make the same change in the level delay output at TIME plus the amount of delay. The entry consists of the location of the delay (string, bit and section), the Hazard

and New Value bits of the input flip-flop, and a settling time equal to the amount of delay plus the settling time of the input flip-flop. If the delay line is ideal, the Hazard Value bit in the entry is zero no matter what the flip-flop's Hazard Value might be. If another entry for the same level delay is already present on the same Delay Value List, it is deleted.

2) If the flip-flop's New Value bit has changed, its Output Specification is checked to see if it feeds a differentiator sensitive to this transition. If so, the differentiator's output Event is added to the Immediate Event List. The settling time of the flip-flop is used as the Event's source completion time.

3) If the flip-flop's Hazard Value has changed to one and it feeds a differentiator, the translator is informed that a Hazard Alarm has been detected and is given the Reference Name and bit number of the flip-flop.

VI. At this point the Active Level List contains the locations of all flip-flops, level delays, and section interface levels whose values or settling times have changed and have not been propagated through the rest of the level logic. These changes are now propagated through the use of the Dependent Combinational Level Lists, which are part of each Output Specification.

A. If a dependent level is ideal, its Input Change Tag is set.

1) If the Input Change Tag is already set, there is no need to propagate the change past that point. Whenever a new value is

computed for an ideal combinational level, its Input Change Tag is reset.

2) If the level is an input to a differentiator, its new value must be computed. If the value makes the transition which the differentiator is sensitive to, the differentiator's output Event is added to the Immediate Event List. TIME is used as the Event's source completion time.

3) If the level is an input to a level delay, its new value must also be computed. If the value changes, an entry is added to an AQ Delay Value List to cause the level delay value to make the same change at TIME plus the amount of delay listed in the Delay Specification. The value change entry's settling time is the same as its starting time, TIME plus the amount of delay.

4) If the level's dependent combinational levels are on a non-permanent section other than the Active Section, its new value must be computed. If the value changes, the level's location (string, bit and section) is added to the destination section's Section Input List if not already on it.

5) If the new value of the level is calculated in (2) or (3) above and found not to have changed, the change need not be propagated further. If the new value was not calculated or was found to change and the level's dependent combinational levels are on section 0 or the Active Section, the change is propagated to all of them.

B. If a dependent level is <u>detailed</u>, its new value and settling time is computed.

1) Since detailed level outputs are evaluated at all times, the subroutine which evaluates detailed combinational output levels and settling times is never used recursively.

2) If the level's New Value bit has made a transition and it feeds a differentiator sensitive to this transition, the differentiator's output Event is added to the Immediate Event List. The level's settling time is taken as the Event's source completion time.

3) If the level's Hazard Value has changed to one and it feeds a differentiator, the translator is informed that a Hazard Alarm has been detected and is given the level's Reference Name and bit number.

4) If the level's dependent combinational levels are on a non-permanent section other than the Active Section and its value or settling time has changed, its location (string, bit and section) is added to the destination Section Input List, if not already on it.

5) If the level's dependent combinational levels are on section 0 and the Active Section and its value or settling time has changed, the change is propagated to the dependent levels.

C. After a level's changes have been propagated to all combinational levels dependent on it, it is removed from the Active Level List.

VII. All entries on the Ideal Flip-Flop List are transferred to the Ideal Input Tag List. If a flip-flop is already on the Tag List, the second entry can be deleted. At this point the Ideal Flip-Flop, Detailed Flip-Flop and Active Level Lists are empty. If the Immediate Event List

contains some more entries for section 0 or the Active Section, the simulator returns to step III.

VIII. If the Immediate Event List or one of the Section Input Lists is not empty, the simulator returns to step I. Otherwise entries are added to the Reset List attached to the AQ time cell next to the top to reset Input Tags for all of the ideal flip-flops on the Ideal Input Tag List. If the Simulation Terminate Tag is set, control is passed on to the translator. If not, the CLOCK is stepped. If the AQ is empty, control is passed on to the translator along with a Simulation Completed message. Otherwise the simulator returns to step I with the lowest numbered section other than zero becoming the Active Section.

2. Discussion of Simulator

The algorithm outlined above accurately simulates the models of Section III in most respects. In those cases where flip-flop inputs may lead to ambiguous outputs, the algorithm reports an alarm and chooses one of the possible outputs. There is a weakness, however, in the way that output signal spread is calculated for differentiators. The simulator assumes that it can correctly calculate the settling time of a level transition when it first becomes aware that the transition is occurring. This is not true for a flip-flop because subsequent set or reset inputs with less signal spread may reduce the spread of its output. This in turn can be propagated through the logic to cause combinational signal spreads to be reduced. Therefore it is possible for the simulator to discover that an Event, generated by a differentiator and already activated, actually has smaller signal spread than originally calculated.

It is not possible to make signal spread corrections to previously activated Events in the data structure of Appendix A. In order to be able to do so, an additional list would have to be added for each detailed level's Strobe Tag and detailed flip-flop's Set, Reset and Complement Tag. The Strobe Tag List would contain the names and completion times of each Event and Transfer sampling the level. (In the case of Transfers, both the Transfer's and the activated Event's locations would be recorded.) Then if it becomes necessary to adjust an Event's completion time, the new reset time for the Strobe Tag can be recalculated unambiguously. When a new Event or Transfer samples a level, an entry is added to the Strobe Tag List. When the Strobe Tag is Reset the list is cleared. Input tag lists would function in a similar way so that

flip-flop output settling times could be recalculated unambiguously when an input completion time is revised. Since it may be necessary to revise the completion times of Events crossing data section boundaries, each non-permanent data section would have an Event Spread Revision List.

The above structures and procedures were left out of the above outline and the figures of Appendix A because they would just add further complications. It should be clear how to incorporate them into the data structure and simulation algorithm. Note that additional cost is primarily in extra data space requirements because the lists are not referred to except when completion time revisions are necessary.

APPENDIX C - COMBINATIONAL LEVEL FORMULAS

The combinational formulas used in the simulation data structure (Figures A-1 and A-2 of Appendix A) are modified Reverse Polish representations of the design language combinational formulas found in the description of the system being simulated. They are used by a push-down store routine which makes a single pass from the top to the bottom of a formula and evaluates it. When the evaluator reads the name of an argument (a pointer to a level string and the bit number of the left-most bit)* it fetches the argument and adds it to the top of the push-down store. When it reads the name of an operator on the formula, the correct number of arguments are removed from the top of the store, operated on, and the result is added to the store. The number of arguments is fixed for each operator type; all operators discussed here have either one or two arguments. There is no upper bound to argument lengths, so more than one push-down store cell may be required to store an argument. Special symbols are included in the formula character set which allow the concatenation of more than one level substring value to form a single argument, the expansion of a single level bit value into a multibit argument, and to change the current value for argument length. Another special symbol, Ω, is used to signify the end of a formula. The result of the evaluation of a well-formed formula is a single argument in the push-down store, which is the value of the formula.

---

*Throughout this appendix argument locations are specified by a pointer to a level string and the left-most bit number. In the actual data structure, the argument's section number and model type (idealized or detailed) must also be included.

Normal Boolean formulas map into the data structure in a straightforward way. For example, the design language formula {A[0:4] $\wedge$ ($\neg$ B[3:7] $\vee$ C[0:4])} would be translated as {A', a, B', b, $\neg$, C', c, $\vee$, $\wedge$, Ω}, where A', B' and C' are pointers to the level strings representing A, B and C, and a, b and c are the bit numbers of A[0], B[3] and C[0]. Each symbol is contained in its own cell on the formula, and the operators must be distinguishable from pointers to level strings. The sample formula is pointed to from the combinational level string representing the five bits to be evaluated using the formula. Immediately preceding the cell pointing to a formula is a cell containing the length of the value to be calculated. (Again refer to Figure A-1 or A-2). In this case the argument length is five, and five bit arguments are fetched when the evaluator reads argument names. If the cell preceding the pointer to the above formula had contained a three, the meaning of the formula would be {A[0:2] $\wedge$ ($\neg$ B[3:5] $\vee$ C[0:2]}.

It may be necessary to concatenate the value of more than one level substring together to form a single argument. The special concatenation symbol, /, is used for this purpose. For example, the formula {A[0:9] $\oplus$ B[3:7],C[0:4]} would be translated {A', a, /, 5, B', b, 5, C', c, $\oplus$, Ω}. Note that the concatenation symbol is followed by length-argument name pairs until the lengths add up to the current argument length. In the above example the argument length is ten and the concatenation stops after the value of the substring C[0:4] is fetched. Another special symbol $, is introduced to allow a single bit value to be expanded into a multibit argument. For example, the formula {A[0] $\wedge$ C[0:4]} would be translated {$, A', a, C', c, $\wedge$, Ω}, where the formula length would be five.

The argument lengths for the relational operators (=, ≠, <, ≤, >, and ≥) are generally greater than the length of the single bit resultant. Therefore the special symbol, #, is included for changing the current argument length. For example, the single bit formula {A[0:4] = B[3:7]} would be translated {#, 5, A', a, B', b, =, Ω}. When the formula is evaluated, the execution of the "=" operator causes the argument length to reset to one. This same technique can also be used to fetch arguments for the convenience operators and, which forms the logical "and" of all bits of its argument, or, which forms the logical "or" of all bits of its argument, and mod2, which forms the "sum mod 2" of all bits of its argument. For example, the formula {and (C[0:4]) = or (A[0:2])} would be translated {#, 5, C', c, and, #, 3, A', a, or, =, Ω}.

In the above discussion we have shown how to translate well-formed combinational level formulas from the current version of the design language into the simulation data structure. We shall now propose that the design language be extended to include the common arithmetic operators +, -, ×, ÷ and rem (remainder) in combinational level formulas. These functions can be easily and cheaply realized by the simulation system and would be useful for rough simulations of arithmetic components before they are designed in detail, or after they have been completely checked out. These operators may also be used to make macroscopic models of arithmetic functions which may be inserted into a design. Their operation may be compared with detailed simulations of the same functions, and differences in their behavior reported to the designer. This may often be preferable to inserting canned answers to be compared against during simulation.

Each arithmetic operator requires two arguments. It is proposed that all arguments be treated as positive integers and that the length of the result be the same as the length of the longest argument. If the result of one of these operations is actually longer than this, the extra high order bits would be truncated. When the shorter arguments are translated into the data structure, zeros are concatenated on the left end to make both arguments and the result all the same length. Only the subtraction operator can produce "negative" results, and these would be represented in two's complement form.

The arithmetic formula {A[0:15] + (3 × C[0:9])} would be translated {A', a, T, t, /, 6, Z, z, 10, C', c, ×, +, Ω}, where T is a pointer to a constant level string with the sixteen bit value 3, and t is the left-most bit number. Likewise Z and z specify a six bit constant level string of all zeros. Therefore the length of all arguments and the resultant is sixteen. What if n, the level string length for simulation system, is less than sixteen? This was not a problem for the non-arithmetic formulas because they can be split into separate formulas for each group of bits. The special right shift symbol, →, is introduced to solve this problem. It is used to truncate low order bits from the top argument on the push-down store and reduce the current value of argument length by the same amount. For example, suppose the eight high order combinational bits given by the above formula are represented on one level string, and the other eight on another. The data structure formula (of length eight) for the high order bits would be {#, 16, A', a, T, t, /, 6, Z, z, 10, C', c, ×, +, →, 8, Ω}. The formula (again of length eight) for the low order bits would be {A', a', T, t', C', c',

×, +, Ω}, where a' and c' are the bit numbers for A[8] and C[8] and t' is eight greater than t in the original formula. Note that for the high order or left-most eight bits, the complete sixteen bit result is generated and then the eight low order bits are shifted off. This technique was not necessary for the low order bits because a formula could be generated for them with all arguments of length eight. This would not be possible if the addition operator were replaced by a subtraction operator. Therefore another special symbol, ←, is introduced which truncates high order bits from the top argument on the push-down store and reduces the current value of argument length by the same amount. The two symbols, → and ←, can be used together to select any contiguous bits of a result. An alternate for the low order eight bit formula given above is {#, 16, A', a, T, t, /, 6, Z, z, 10, C', c, ×, +, ←, 8, Ω}.

Note that when a formula is used to evaluate detailed levels, independent settling times must be calculated for each bit. The rules illustrated in Figure 3-8 can be used for non-arithmetic formulas because they are specified in detail. There are a number of alternate detailed realizations of arithmetic formulas and each output bit generally depends on a number of input bits. Since the main objective of arithmetic formulas is simulation efficiency, crude detailed models of them are justified. Therefore it is proposed that the settling time for each changing level be calculated by adding its ambiguity time to the <u>maximum</u> of the input settling times.

APPENDIX D - DESIGN LANGUAGE DESCRIPTION

In this appendix we shall present a brief discussion of the more important features of the logic design language developed at M.I.T. under the supervision of Professor J. B. Dennis. A syntax was constructed for the language by G. J. Burnett in his MS thesis, Reference (16), submitted in August, 1965. Subsequently, a set of notes describing the language was written by H. F. Ledgard for M.I.T. course 6.535 in November, 1965. These notes were based primarily on Burnett's thesis, but a number of modifications were made to conform to Professor Dennis's notation and to define the language more precisely. Most of the material presented here is an edited form of these notes, Reference (17). Ledgard also prepared a companion set of notes, Reference (18), which contains a number of examples of the usage of the language. The reader who is interested in a more complete description of the language is referred to all three of the above references. Note that the language is still under development and therefore is subject to change. Some additions to it have been proposed in this thesis, but they have not been included here.

1. Basic Structure

The design language borrows heavily from the ALGOL programming language for its syntax, as does the similar language which is described by Y. Chu in Reference (17). For example, semicolons are used to delimit statements, statements may be continued from line to line indefinitely, multi-character special symbols are underlined, and the special symbols begin and end are used group several statements together to form a single statement.

The specification of a system in the design language consists of the descriptions of its basic sub-systems, or sections of logic, called components. The description of a component begins with the declaration:

component < component name or abbreivation > ;

Following this is a description of all signal interfaces which the component has with other components. Each interface description consists of three parts:

a) The declaration:
interface < external component name or abbreviation > ;
naming the interfacing component.

b) A series of input and output declarations giving all connections to the interfacing component.

c) The statement:

end interface;

After all interfaces have been specified, the internal logic of the component is described with a series of statement types to be discussed later. The component description ends with the statement:

end component < component name or abbreviation > ;

example:

```
component MEMORY CONTROL;
    interface PROCESSOR;
        input level DATA[0:17], ADDRESS[0:11];
        output level WORD[0:17];
        input pulse read, alter, write;
        output pulse rddone, wrdone, ready;
    end interface;
    interface MEMORY;
        output level YSEL[0:63], XSEL[0:63], B[0:17], R, W, I;
        ---
    end interface;
register ---
---
---
---
end component MEMORY CONTROL;
```

Note that in the above example both pulse and level interface signals can be declared. Levels are referred to by a series of capital letters and numbers in the design language. A series of small letters and numbers

are used to name pulses. The language can be used to describe pulse excited logic, level excited logic, and mixtures of the two.

2. Level Logic

One of the basic building blocks of a component is the register. It is a group of one or more binary elements, generally flip-flops, that are used to store information in the form of "1's" and "0's". Registers are declared and bytes of registers are renamed by the following declarations:

register < register list > ;	Declaration of the capitalized names and bracketed components of the register list as registers; the brackets contain the index of the first (leftmost) and last (rightmost) cells included in the register. After a register is initially declared or if a register has only one cell, the brackets and indices may be dropped.
subregister < subregister list > / < register > ;	Bytes of the register on the right of the slash can also be referred to by the subregister names on the left.

examples:

register A[0:2], B;	Register A contains 3 flip-flops indexed 0 through 2; register B contains 1 non-indexed flip-flop.
subregister C[0:8], D[2], E[0] / F[0:10];	Previously declared register F is being subdivided into 3 bytes. Note that C[0:8] and F[0:8] can be referred to interchangeably, as can D[2] and F[9], and E[0] and F[10].

A good portion of logic descriptions are built from combinational levels dependent upon other combinational level and register values. As a result, a complete set of logical operators are included to formulate combinational expressions:

$\vee$	The logical "or" of its two operands.
$\wedge$	The logical "and" of its two operands.
$\neg$	The logical "not" of the operand on the right.
$\oplus$	The logical "sum module 2" of its two operands.

=, ≠, <, >, ≤, ≥	The relational operators "equal", "not equal", "less than", "greater than", "less than or equal" and "greater than or equal".
and (< level list >)	The logical "and" of all cells on the argument list.
or (< level list >)	The logical "or" of all cells on the argument list.
mod2 (< level list >)	The logical "sum module 2" of all cells on the argument list.
examples:	
S ∧ R	The levels S and R are combined in a logical "and".
A[0:2] ∨ B[0:2]	The registers A and B are combined in a logical "or", carried out on a cell by cell basis.
S ∧ A[0:2]	The single level S is combined in a logical "and" with each cell of register A.
A[0:2] = B[0:2]	This expression has the logical value "1" (true) if each cell of register A has the same logical value as the corresponding cell of register B; the logical value "0" otherwise.
and (A[0:2], B[0:2])	This expression has the logical value "1" if all cells of registers A and B have the logical value "1"; otherwise the logical value is "0".
mod2 (A[0:2])	This expression has the logical value "1" if an odd number of the cells of register A have the logical value "1"; otherwise "0".

Equivalence statements are introduced to assign names to combinational expressions so that they may be referred to conveniently. These statements are of the form:

< name > ≡ < expression > ;

The name assigned by the statement may be used to replace any occurrence of the named expression.

examples:

ADD ≡ (IR[3:8] = 42);
SUM[0:31] ≡ A[0:31] ⊕ B[0:31] ⊕ C[0:31];
C[i-1] ≡ C[i] ∧ (A[i] ⊕ B[i]) ∨ (A[i] ∧ B[i]);

The basic level delay unit of the design language is diagrammed in Figure D-1. Output level G remains at "0" until the input level F undergoes a "0" to "1" transition, at which time level G becomes "1" for t units of time and then returns to the "0" state.

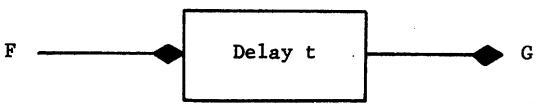


FIGURE D-1

Basic Level Delay Unit

This delay unit is described by the following statement:

G ≡ delay (F, t);

3. Control Event and Transfer Statements

One of the basic operations of a digital system is the transfer of information from one register to another. The following symbols are used to describe the excitation of transfers and the transfers themselves:

:	The control operator, indicating that the expression to the left of the ":" represents an event that initiates the action indicated to the right of the ":". The scope of the control operator extends to the next control operator or to the end of the component.

=>	The jam transfer operator, indicating that the level expression to the left of the "=>" is to be transferred to the register given to the right of the "=>" on a cell by cell basis.
→	Ones transfer operator, indicating that the ones of the level expression to the left of the "→" are to be transferred to the register given to the right of the "→" on a cell by cell basis. If no left operand is given, all cells on the right are set to "1".
~	Zeros transfer operator, indicating that the zeros of the level expression to the left of the "~" are to be transferred to the register given to the right of the "~" on a cell by cell basis. If no left operand is given, all cells on the right are reset to "0".
↑	The complement transfer operator, indicating that each cell of the register given to the right of the "↑" is to be complemented only if the corresponding cell of the level expression to the left is a "1". If no left operand is given, all cells on the right are complemented.

examples:

sum: A[0:2] => B[0:2];	On the occurrence of the sum pulse, the contents of A[0] replaces the contents of B[0], A[1] replaces B[1], and A[2] replaces B[2].
sum ∧ S: A[0:2] => B[0:2];	On the occurrence of the sum pulse, if the level S is "1", then A[0] replaces B[0], etc.
¬ S: A[0:2] => B[0:2];	When level S has a "1" to "0" transition, A[0] replaces B[0], etc.
sum ∨ S: A[0:2] => B[0:2];	On the occurrence of the sum pulse or a "0" to "1" transition of level S, A[0] replaces B[0], etc.

```
sum: A[0:2] → B[0:2];          On the occurrence at the sum pulse.
     A[0:2] ~ B[0:2];          A[0] replaces B[0], etc.; C[0],
     ↑ C[0:2];                 C[1] and C[2] are complemented;
     C[0:2] ↑ D[2:4];          and if the old value of C[0] = 1,
                               D[2] is complemented, etc.
```

Note that control events may be either pulse occurrences or level transitions. The last example illustrates the use of a control event to initiate several actions. Since all actions within the scope of the same event are activated simultaneously, the order in which they are listed is irrelevant. Therefore, in the last example the old value of C[0:2] is used to complement D[2:4], rather than the new complemented value. The same control event expression may be used to the left of any number of control operators. This has the same effect as grouping all statements within the scope of these control operators together under the scope of any one of them. All statements would be activated simultaneously when the control event is activated.

Control events may also be used to activate pulses. Any further actions initiated by the pulse occur simultaneously with the pulse's activation.

examples:

```
sum:    tpl;                   This has same effect as the last
        A[0:2] => B[0:2];      example.
tpla:   ↑ C[0:2];
tpl:    C[0:2] ↑ D[2:4];
        tpla;

S:      sum;                   The pulse sum is activated when
                               level S makes a "0" to "1"
                               transition.
```

Delay statements are used in the language to introduce pulse excited delay lines. The scope of a delay statement extends to the next control operator, delay statement, or the end of the component.

examples:

```
sum:    delay (50 ns);          50 ns after the occurrence of
        1 → R;                  pulse sum, flip-flop R is set
                                to "1".

tp1:    0 ~ B;                  On the occurrence of pulse tp1,
        delay (50 ns);          flip-flop B is cleared to "0";
        1 → R;                  after a 50 ns delay, flip-flop
        0 ~ S;                  R is set to "1" and flip-flop
        delay (100 ns);         S is cleared to "0"; after an
        tp2;                    additional 100 ns delay pulse
                                tp2 occurs.

tp2:    delay (4 μs, TP2D);     This statement declares a single-
        1 → B;                  shot with output level TP2D.
                                On the occurrence of pulse tp2,
                                level TP2D goes to "1" and remains
                                there for 4 μs before returning
                                to "0". When TP2D changes from
                                "1" to "0", flip-flop B is set
                                to "1". Level TP2D may be used
                                as an input to combinational
                                logic network, just as flip-
                                flops outputs are.
```

Equivalence statements may be used to assign pulse names to control events, just as they are used to name combinational levels.

example:

```
sum ≡ (tp1 ∨ tp4) ∧ S;          Pulse sum occurs whenever pulse
                                tp1 or tp4 occurs and level S is
                                in the "1" state.
```

Pulse oscillators are declared in the following manner:

clock < pulsename > (< time between pulses >);

examples:

```
clock osc (160 ns);             osc pulses occur every 160 ns.

go ∨ osc: delay (160 ns);       This is an example of a delay
          osc;                  loop oscillator. Pulse osc
                                occurs every 160 ns, once pulse
                                go occurs.
```

4. Integers

It is sometimes convenient to specify constants such as register indices and delay values symbolically and assign values to the symbols at some other place in a design description. Therefore, a capability for declaring and assigning values to integers is included in the design language:

integer < integer list >;	This statement type is used to declare integers used within the design. A series of small letters and numbers are used to name integers.
< integer > := < value >;	This statement type is used to assign a value to an integer.

example:

```
integer  n, dl;                          Integer n is used to specify the
register  A[0:n], B[0:n], C[0:n];        maximum bit index for registers
     . . .                               A, B and C.  Integer dl is used
n:= 31;       dl:= 100;                  to specify the length of a pulse
                                         delay line.
     . . .
sum:  delay (dl ns);
     . . .
```

5. Delimiters

A number of special delimiting characters can be used within component descriptions. Some of them were used in preceding examples.

;	Semicolon, used to delimit statements in the language.
*	Asterisk, used to mark the beginning of a comment statement.
[ ]	Brackets, used to enclose level and pulse indices.
,	The comma, used to separate arguments and expressions on a list. The comma can also be used in certain contexts as a concentration character, indicating that a series of registers or levels separated by commas is to be considered as a unit.

( )	Parentheses, used to separate operators from their operand lists and to enclose expressions where necessary for clarity.
begin end	Block delimiters, used to enclose a number of statements which are considered as a unit; i.e., as a single statement.

6. Iteration Statements

The above declaration, equivalence, action and delay statements are amply sufficient for describing any design which is describable in the present form of the design language. The remaining statement types are included for conciseness of notation and ease of expression. The basic iteration statement is constructed as follows:

for < index > := < for list > do < statement >;

The scope of the for statement may be extended to include any number of statements by simply enclosing them within a begin - end pair, as in ALGOL.

example:

```
for i := 0, 1, 2, 3 do    begin  0 ~ A[i];
                                 0 ~ B[i];  end;
```

The "for list" in the above example is an example of what will be referred to as an "initialization list". In order to extend the use of the basic iteration statement, we define the following alternative construction:

```
< for list >   ::=   < initialization list >
< for list >   ::=   < initialization list > step < integer >
                       while    < Boolean expression >
< for list >   ::=   < initial value of index > through
                     < final value of index >
```

and the following symbols are used in constructing "for lists":

+, -, ×, /	arithmetic operators, with their normal meaning in constructing arithmetic expressions.

examples:

```
for   i := 0 step 1 while i ≤ 3  do begin    0 ~ A[i];
                                             0 ~ B[i];  end;

for   i := 0 through 3  do begin    0 ~ A[i];
                                    0 ~ B[i];  end;
```

The material within the begin - end brackets of the for statement is an indexed description of logic to be constructed. The indexing is present because a number of pieces of hardware have the same construction, but different names or positions in a device. The index takes on, in sequence, those values specified by the for list until the list expires or the Boolean expression becomes false. The index can be used wherever needed between the begin - end pair and has no meaning outside these brackets.

To eliminate the use of certain index values for a limited number of statements within a for statement, we define the "where" statement as follows:

```
where  < index >  ≠  < constant list > do begin < statement > end;
```

The where statement is used in constructions of the following form:

```
for     . . .     while     . . .     do
        begin
        . . .
        . . .
        where  < index >  ≠  < constant list >  do  < statement >;
        . . .
        . . .
        end;
```

The constant list contains valid index values from the encompassing for statement. The statements within the scope of the where statement are not converted to hardware for these index values.

7. Conditional Statements

Actions are often conditioned by level signals in digital systems. We have already discussed techniques for constructing control event expressions, which can handle conditional actions as in the following examples:

```
go ∧ (READ ∨ WRITE) :     done;
go ∧ TRANS ∧ (READ ∨ WRITE) :     1 → A[0];
go ∧ ¬ TRANS ∧ (READ ∨ WRITE) :     0 ~ A[1];
```

These statements are interpreted as follows: "On the occurrence of the go pulse, if either of the levels READ or WRITE are "1", then the pulse done is activated; if level TRANS is also "1", then "1" replaces the value of A[0], otherwise "0" replaces the value of A[1]. As the conditioning expressions become more complicated, and if a number of actions are conditioned by slightly different expressions, the following construction might be preferred:

```
if < boolean expression > then < statement > else < statement > ;
```

where the term <statement> refers to action, equivalence or iteration statements and the "else" part of the construction is optional. The same structure can be used to construct combinational expressions:

```
if < boolean expression > then < expression > else < expression >
```

we may now rewrite the above examples as:

```
go:  if  READ ∨ WRITE  then
      begin    done;
               if  TRANS  then  1 → A[0]  else  0 ~ A[1];
      end;
```

Example of combinational expression:

```
ADDRESS[8:23]  ≡  if  IOREQUEST  then  IOADD[8:23]
          else    if  DATAREQUEST  then  CAR[8:23]
          else    if  INSTREQUEST  then  PC[8:23];
```

Note that "if" statements are often less clumsy than an equivalent set of statements in the previous notation. Their most important virtue is that they are usually easier for people to understand.

8. Define Feature

This feature is provided so that units of logic, which may occur more than once in a design, need only be described once. Its usage is directly analogous to macro definition statements in macro-assembly language computer programs. A unit described using the define feature is specified in the following format:

```
define  < unit name >  (< parameter list >);
        < declarations >
          - - -
          - - -
end definition  < unit name >;
```

The parameter list contains an ordered sequence of parameters needed to specify the unit; e.g., input and output signals, internal signal names, number of flip-flops, etc.

example:

```
define adder (A, B, S, n);
input level A[1:n], B[1:n];
output level S[1:n];
integer n;
- - -
- - -
end definition adder;
```

The designer may insert a previously defined unit in his design by writing:

insert < unit name > (< parameter list >);

Each time the designer inserts a defined unit, he must provide unique values for the parameters specifying internal signal names. Unique names are automatically generated for internal signal names not included as part of the parameter list.

example:

insert adder (AC, BR, SUM, 24);

9. Summary

The design language presented here is capable of comfortably describing many types of digital systems, both synchronous and asynchronous. It could serve as the input language for a processing system which automatically generates wiring instructions for hardware or a design simulation system. However, the language is not to be considered complete or rigorously defined. The naming problem and the problem of structuring the design language are particularly difficult to solve.

BIBLIOGRAPHY

(1) S.R. Cray and R.N. Kisch, "A Progress Report on Computer Applications in Computer Design," Proc. WJCC, pp. 82-85; February, 1956.

(2) H.L. Engel (Ramo-Wooldridge), "Machine Language in Digital Computer Design," Proc. WJCC, pp. 182-186; May 6-8, 1958.

(3) T.A. Connolly, "Automatic System and Logical Design Techniques for the RW-33 Computer System," IRE International Conv. Rec., Pt. 2, pp. 124-132; 1960.

(4) W.L. Gordon, "Data Processing Techniques in Design Automation," Proc. EJCC, pp. 205-209; 1960.

(5) W.A. Hannig and T.L. Mayes, "Impact of Automation on Digital Computer Design," Proc. EJCC, pp. 211-232; 1960.

(6) A.L. Leiner, et al., "Using Digital Computers in the Design and Maintenance of New Computers," IRE Transactions, Vol. EC-10, No. 4, pp. 680-690; 1960.

(7) W.K. Orr and J.M. Spitze, "Design Automation Utilizing a Modified Polish Notation," Proc. FJCC, pp. 643-650; 1964.

(8) K. Jacoby and A.R. Laliberte´ (Philco), "Using a Computer to Design a Computer," Computers and Automation, Vol. 15, No. 4, pp. 36-39 and 58; April, 1966.

(9) I.S. Reed, "Symbolic Design Techniques Applied to a Generalized Computer," M.I.T. Lincoln Lab., Lexington, Mass. TR No. 141; January 3, 1957.

(10) T.C. Bartee, I.L. Lebow and I.S. Reed, "Theory and Design of Digital Systems," The McGraw-Hill Book Co., Inc., New York, N.Y.; 1962.

(11) H.P. Schlaeppi, "A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)," IEEE Transactions, Vol. EC-13, No. 4, pp. 439-448; 1963.

(12) H. Schorr, "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language," IEEE Transactions, Vol. EC-13, No. 6, pp. 730-737; 1964.

(13) R.M. McClure, "A Programming Language for Simulating Digital Systems," Journal of the ACM, Vol. 12, No. 1, pp. 14-22; January, 1965.

(14) G. Metze and S. Seshu, "Computer Compiler, Part I - Preliminary Report," Report No. R-264, University of Illinois Coordinated Science Laboratory; August, 1965.

(15) Y. Chu, "An Algol-Like Computer Design Language," Commun. ACM, Vol. 8, No. 10, pp. 607-615; October, 1965.

(16) G.J. Burnett, "A Design Language for Digital Systems," MS Thesis at M.I.T.; August, 1965.

(17) H. Ledgard, "A Design Language for Digital Systems," M.I.T. Course 6.535 Notes; November 23, 1965.

(18) H. Ledgard, "A Set of Basic Logical Building Blocks; Examples of the Design of Logical Networks," M.I.T. Course 6.535 Notes; November 23, 1965.

(19) Y.C. Lee and R.J. Merkert (RCA), "Evaluating Worst-Case Conditions," Electronic Des., Vol. 11, pp. 54-61; March 1, 1963.

(20) D.A. Huffman, "The Synthesis of Sequential Switching Circuits," Journal of the Franklin Institute, Vol. 257, No's. 3-4, pp. 161-190, 275-303; March and April, 1954.

(21) S.H. Caldwell, "Switching Circuits and Logical Design," John Wiley and Sons, Inc., New York, N.Y.; 1958.

(22) M. Lehman, R. Eshed and Z. Netter, "The Checking of Computer Logic by Simulation on a Computer," The Computer Journal, Vol. 6, No. 2, pp. 154-162; 1963.

(23) J.F. Griffin and M.J. Hains (IBM), "An Experiment with the Simulation of Machine Logic and Control," 1965 IEEE Intl. Conv. Rec., Part 3, pp. 51-66; March, 1965.

(24) G.N. Stockwell (Nortronics), "Computer Logic Testing by Simulation," IRE Trans., Vol. MIL-6, pp. 275-282; July, 1962.

(25) S. Rubin, "Simulation and Model of an Input-Output Unit for Airborne Computers," Technical Report AFAL-TR-64,312, Air Force Avionics Laboratory; 1964.

(26) M.A. Breuer, "Techniques for the Simulation of Computer Logic," Commun. ACM, Vol. 7, No. 7, pp. 443-447; July, 1964.

(27) R.P. Larsen and M.M. Mano, "Modeling and Simulation of Digital Networks," Commun. ACM, Vol. 8, No. 5, pp. 308-312; May, 1965.

(28) A.L. Scherr, "An Analysis of Time-Shared Computer Systems," M.I.T. MAC-TR-18 (THESIS); June, 1965.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-31

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 146 (153 images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number):_____

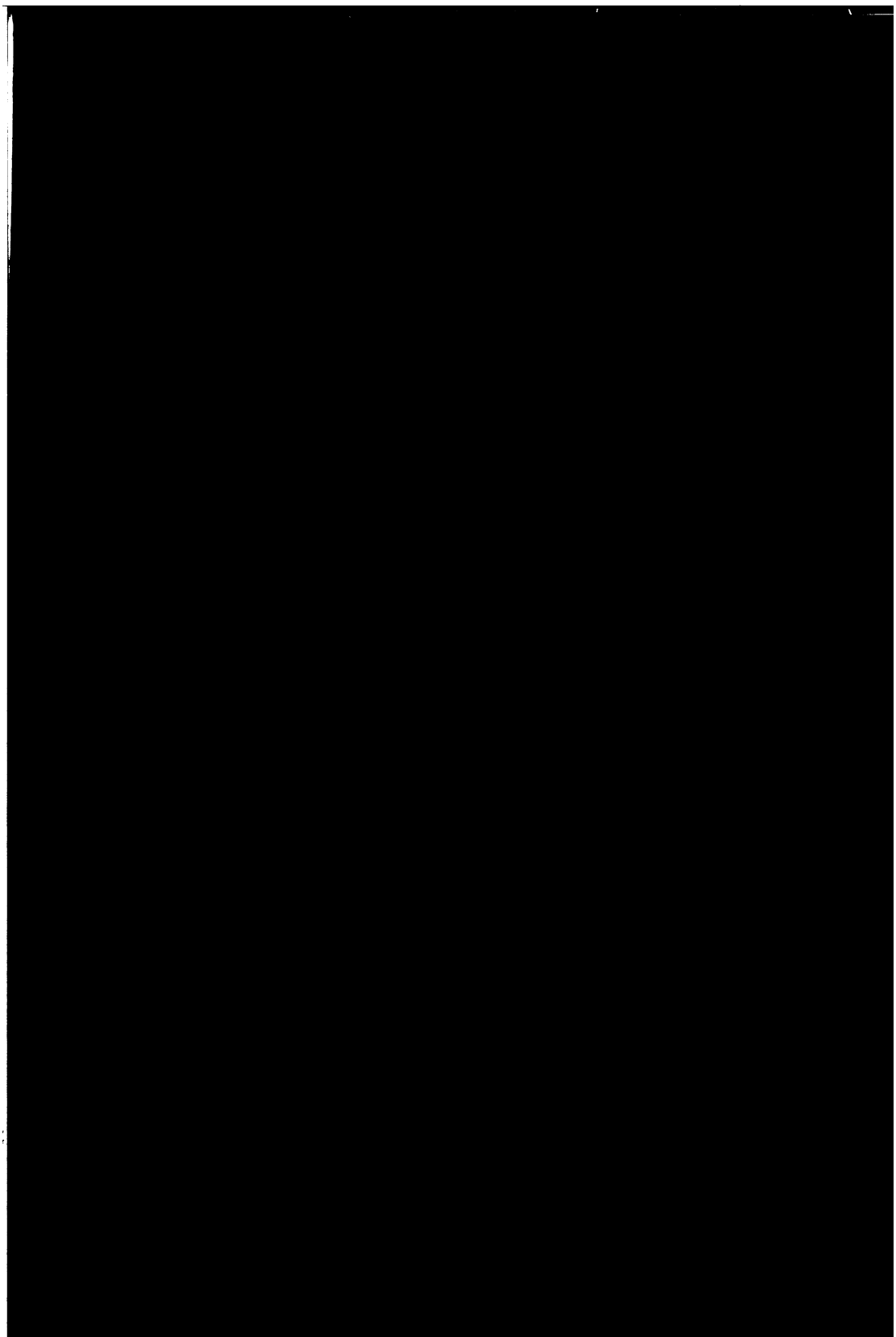
Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP1 (1-146) UNNUMBERED TITLE PAGE, 2-145, WHITE BLANK
(147-153) SCANCONTROL, COVER, FUNDING AGENT, DOD, TARGET'S(?)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 12/26/95 Date Returned: 12/28/95

Scanning Agent Signature: Michael N. Cook



Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/26/1995

M.I.T. Libraries
Document Services