Basic Techniques of Gate Level Simulation
- A Tutorial -

Flávio Rech Wagner

RT nº 12 CPGCC/UFRGS outubro 1984

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Cx. Postal 1501 - Telex (051) 2680
Av. Osvaldo Aranha, 99 - Fone: 21.8499
90.000-PORTO ALEGRE-RS-BRASIL

ABSTRACT

Gate level simulators are those which simulate digital logic circuits composed only of basic gates such as NAND's, NOR's, etc. There are two basic types of gate level simulators: compiled ones, in which the logic structure is directly translated into a sequence of instructions of the host computer, and table-driven ones, in which the logic structure is translated into a table structure, which is then interpreted by a simulation program. Table-driven simulators can assume any delay values for the gates. Not only single propagation delays can be considered, but also more complicated models, which include inertial delays, differing transition times, rise and fall times, etc. Accurate design verification can be obtained only if multiple-valued logic is used.

Key-words: Gate level simulation, compiler-driven and table-driven simulation, timing models, simulation algorithm.

RESUMO

Simuladores a nível de portas lógicas são aqueles que simulam circuitos lógicos digitais compostos apenas de portas lógicas elementares como NANDs, NORs, etc. Existem dois tipos básicos de simuladores: compilados, nos quais a estrutura lógica é traduzida diretamente para uma sequência de instruções do computador hospedeiro, e dirigidos por tabela, nos quais a estrutura lógica é traduzida para uma estrutura de tabelas, que é então interpretada por um programa simulador. Simuladores dirigidos por tabela podem assumir quaisquer valores para os tempos de propagação das portas lógicas. Não apenas tempos de propagação simples podem ser considerados, mas também modelos mais complicados, que incluem tempos de propagação inerciais, diferentes tempos de transição, tempos de subida e descida, etc. Verificação acurada do projeto só pode ser obtida se lógica com múltiplos valores for usada.

Palavras-chave: Simulação a nível de portas lógicas, simulação de código compilado e dirigida por tabelas, modelos de comportamento temporal, algoritmo de simulação.

SUMMARY

1. INTRODUCTION

This paper is a tutorial about techniques employed in gate level simulators. It tries to present in a simple way the models, algorithms and data strucutures which are currently (or were) used.

No considerations are made about system characteristics, like implementation language, input description language (to specify elements and interconnections of the circuit) and translation (to generate the data structures to be processed). These features are mainly dependent on the user needs and on the facilities offered in the installation where the simulator is implemented. Other surveys cover these aspects [BRE 76] [SZY 75].

Furthermore, no emphasis is placed on hazard and race considerations, because this is the subject of digital logic texts. The reader which knows these concepts wil be able to correctly choose the simulation model, among those described in this paper, to meet his needs. More sophisticated techniques of hazard and race detection, using 5- and 8-valued logic, can be found in the references [BRE 76] [WAG 84]

Finally, this paper does not cover simulation techniques for the functional level [SZY 73] [CHA 74], which is considered as an abstraction of the gate level, because this level imposes some severe modification in the models employed at the gate level.

2. ZERO DELAY SIMULATION

In zero delay simulators [HAR 67] [SES 62] the propagation time through all devices is assumed to be zero. As a consequence, only a logic verification of the circuit can be made, i.e., the boolean equations which correspond to the circuit are evaluated. No timing analysis is made, and so problems like races and hazards cannot be detected. For example, consider the simple circuit shown in Fig. 1. Due to the propagation delay of the inverter, when the input A switches from 0 to 1 a short 0 pulse could be generated at the output C. However the zero delay model wiil not predict such hazard.

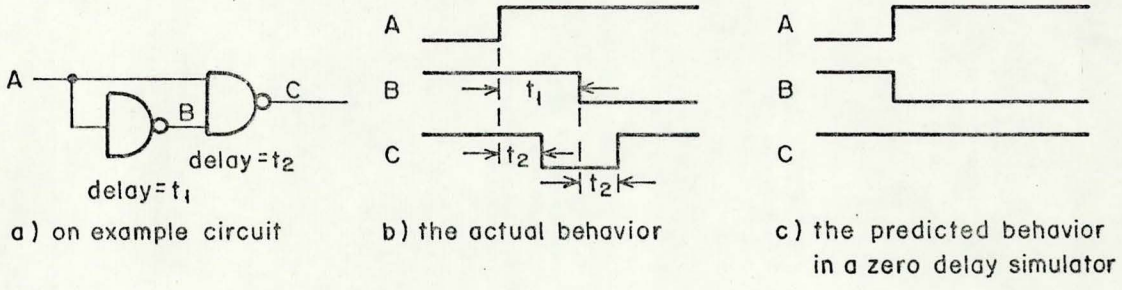


Figure 1 - Zero delay simulation

The circuit must be levelized to provide a correct ordering in the boolean equation evaluation. Levelizing consists in assigning a level number to each element in the circuit. If Li is the level of element i, and if an element a has inputs from elements b, c and d, then La = 1 + max (Lb, Lc, Ld). Furthermore, all feedback loops must be identified and broken. The circuit is modeled in a canonical form like in Fig. 2. The level 0 is assigned to all primary inputs Xi. An example of a levelized circuit is shown in Fig. 3.

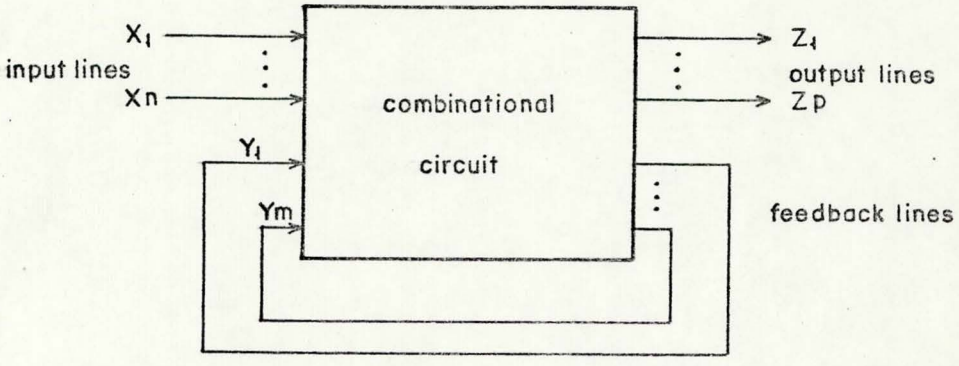


Figure 2 - Circuit model in a zero delay simulation

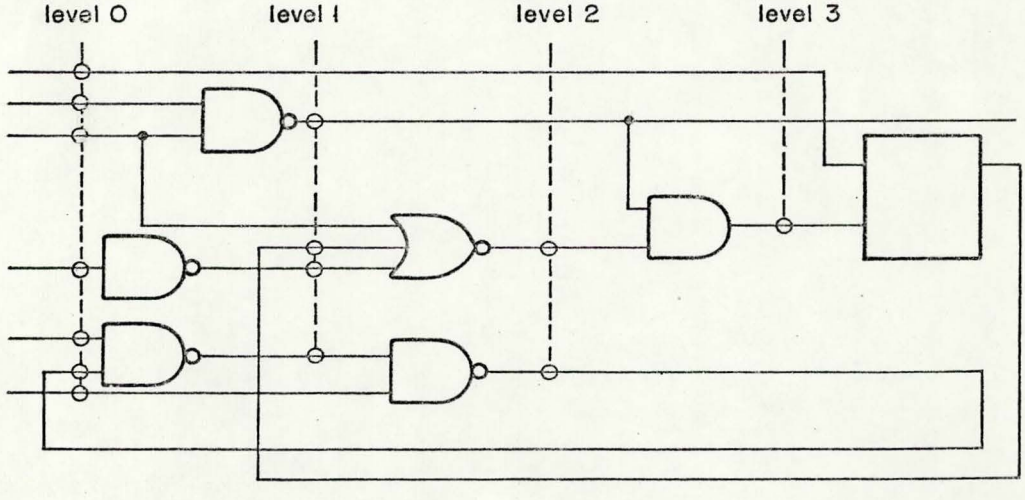


Figure 3 - A levelized circuit

Simulation is carried out by the following algorithm. All level 1 elements are evaluated first, then all level 2 elements and so on. After all elements have been evaluated, the values of feedback signals are interrogated to determine if they have changed. All logic levels ahead of a

feedback break point which changed must be evaluated again. This process is repeated until the circuit has stabilized. After that a new input vector is taken and processed.

It is easy to imagine that an oscillation can occur, and the circuit does not stabilize, i.e., after each simulation pass through all levels some feedback line has different values (alternates between 0 and 1). A solution to the problem is to stop the evaluation after a period of time specified by the user, to print a warning message and to abandone the current input vector. Another solution is, when the oscillation is detected, to assign a value "unknown" to the lines which are oscillating and to proceed with the simulation. We will see 3-valued logic techniques in sections 13 and 14.

3. COMPILER-DRIVEN SIMULATORS [HAR 67] [SES 62]

In general, zero delay simulators are of the compiled variety, in which the evaluation of the boolean expression is compiled into instructions of the host computer. In Fig. 4 we have an example of a simple logic circuit and the computer instructions generated to simulate this circuit in an hypothetical computer. We see that levelizing is needed to obtain the correct ordering of the instructions.

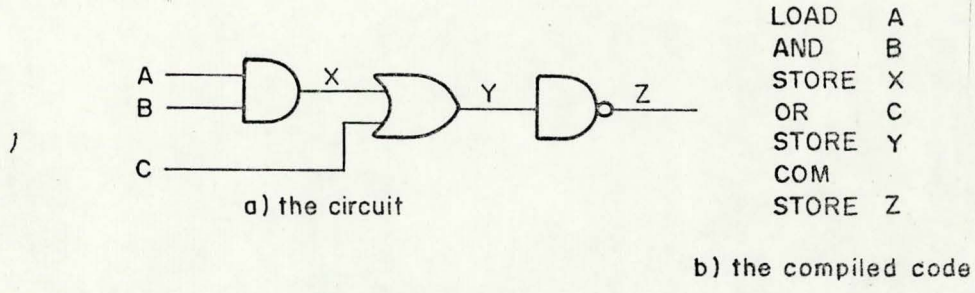


Figure 4 - Compiler-driven simulation

The structure of the simulator is shown in Fig. 5. The preprocessor must identify and break the feedback lines and levelize the circuit. It receives the circuit description in some appropriated input language. The code generator converts the yet levelized circuit description into a sequence of host machine instructions. After that, the description will be treated as any other computer program, which can be loaded and executed.

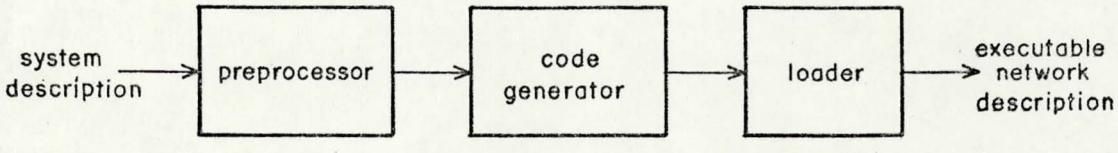
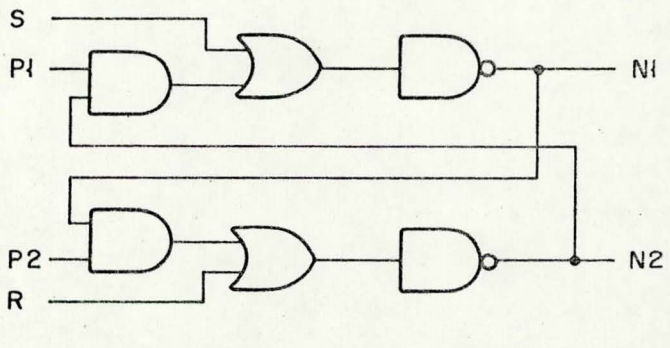


Figure 5 - Structure of a compiler-driven simulator

4. STIMULUS BYPASS

Stimulus bypass is a technique developed by Hardie and Suhokie [HAR 67] for a compiler-driven simulator, and consists in the bypassing of specific computer instructions which correspond to a logic block for which it can be determined that no state change will occur. For example, suppose the flip-flop of Fig. 6a. If it is known that inputs S and R are in their inactive states (value 0), then all instructions which correspond to the evaluation of signals N1 and N2 do not need to be executed, because the flip-flop certainly cannot change its state. Obviously, we will need some instructions to test the bypass condition. This little loss of memory, however, will be compensated by a great time saving. Fig. 6b shows the compiled code for the flip-flop, where the first three instructions test the bypass condition.



a) flip-flop

```
LOAD            S
OR              R
SKIP IF ZERO    *+H2
LOAD            P1
AND             N2
OR              S
COM
AND             P2
OR              R
COM
STORE           N2
AND             P1
OR              S
COM
STORE           N1
```

bypass condition

b) the compiled code

Figure 6 - Stimulus bypass

5. SELECTIVE TRACE [ULR 69]

Let us define the active part of a circuit as those circuit elements whose inputs are changing at a determined moment. In Fig. 7, be A1 the active part of circuit N at t1. Usually this active part is between 2 and 10% of the total number of elements in the circuit [BRE 76], but a ratio of 1% was also reported [HAR 67]. The outputs of elements in A1 are connected to the inputs of another elements, which form the subset B1 of N. However, the activity which takes place in A1 will affect only a subset of B1 at t2, say A2. This subset will be the new active part of N. So, "logical activity may be viewed as a vehicle which travels on narrowly confined but continuously shifting paths of activity through an otherwise idle digital network" [ULR 69]. Such paths of activity suggest a simulation technique which follows these paths and avoids idle elements. Ulrich [ULR 69] has called it "exclusive simulation of activity", but it is nowadays often called "selective trace" [SZY 75]. It would be wasteful to simulate all the logic when only a small portion of the circuit is going to change.

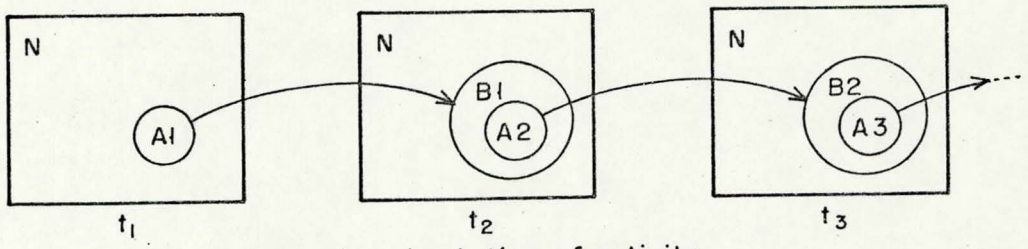


Figure 7 - Exclusive simulation of activity

Let us define an event as a change in value of a signal line. The output of a stable element will only change value when one or more of its inputs change value. Hence, an element needs only be simulated when an event occurs at one

of its inputs. For this reason, this technique is also called "event-directed simulation" [BRE 76].

driven

Almost the totality of simulators developed in the last decade use this technique. As we shall see later (section 8), it needs some scheduling mechanism to order the events in some data structure and to search for the next event to be executed. Bening [BEN 79] alerts that, depending on the kind of logic network organization, the event management can cause such time overhead that levelized logic evaluation would be faster for this network. This would be true for pipelined logic networks, where over 90% of their gates must be evaluated between each clock period.

6. TABLE-DRIVEN SIMULATORS

There is a basic requirement for the implementation of the selective trace technique. We must know what are the elements connected to the output of an element (let us call these elements as the "fan-out elements") to see what is the effect of an event occuring at the output of this element. In Fig. 8, when an event occurs at the output of element i, we have to evaluate element J, but then also the other inputs to element j, say elements k and l, need to be known. Hence, we need also a fan-in list for each element (let us call the fan-in elements as those connected to the inputs of a certain element). For these reasons, event--directed simulators employ a special data or table structure, to reflect the interconnections of the circuit, and are called then "table-driven".

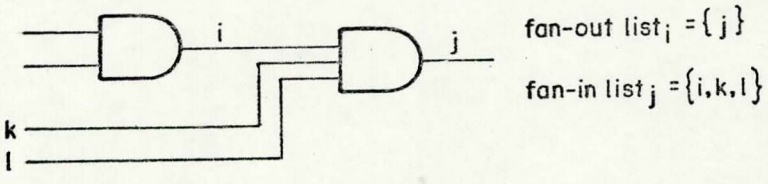


Figure 8 - Fan-in and fan-out lists

This table structure allows the simulation control program to identify and schedule for future evaluation only the elements which are affected by the currently active elements. This technique seems like the stimulus bypass adopted in compiler-driven simulation because both avoid the evaluation of inactive elements. However, in the stimulus bypass technique we need to evaluate a few instructions which correspond to the inactive element to see that it does not need to be evaluated at all. Here, because of the table structure, the paths of potential activity are automatically followed, since idle elements receive no input signals and consequently are not scheduled for evaluation. In compiler-driven simulators we make a static levelizing, before the

simulation begins. Here, the table structure forces the evaluation of the elements in the correct order automatically, and for this reason we speak about a "dynamic levelizing" [BRE 76].

In compiler-driven simulators, the network structure is interspersed within the instructions which evaluate the element outputs. In table-driven simulators, the structure is given in form of tables, which contain fan-in and fan-out lists for each element. The evaluation routine for each element (see section 12) is separated from the network structure, and needs a pointer to it in the table structure, together with the fan-in and fan-out lists.

7. UNIT DELAY SIMULATION

Unit delay simulators assume that the propagation delay time is non-zero and the same for all devices. They operate with two queues, one for the elements which are to be evaluated at the current simulated time, and other for those to be evaluated at the next time step. The elements in the current time step queue (CTSQ) are evaluated and its fan-out elements are put in the next time step queue (NTSQ). When all elements in the CTSQ were evaluated, the simulated time is incremented, the NTSQ becomes the CTSQ, and the old CTSQ is cleared to become now the new NTSQ. The selective trace technique is naturally employed, and so also the table--driven approach.

As with zero delay simulators, the networks is expected to stabilize between application of input patterns. If this does not occur, it means that an oscillation is present due to some timing problem.

In Fig. 9 we have the simulation algorithm of a unit delay simulator |BRE 76|. **La** is the CTSQ and **Lb** the NTSQ. The switching of these two queues at each time step is made by the variables **a** and **b** which alternate between 0 and 1. The simulator works with two queues L0 and L1. Each entry in a queue is a pair (i, v'(i)), where **i** is an active element (possibly a pointer to some entry in a table, where is contained information about the element, such as its type, its fan-in list, its fan-out U(i) and its present output logic value v(i)) and v'(i) is its new logic value, to be assigned at the next time step. Oscillation is assumed if a and **b** are interchanged more than K times between application of two consecutive input patterns, where K is specified by the user. Fig. 10 shows a simple circuit, with the computations carried out by the simulation algorithm and the resulting timing diagram.

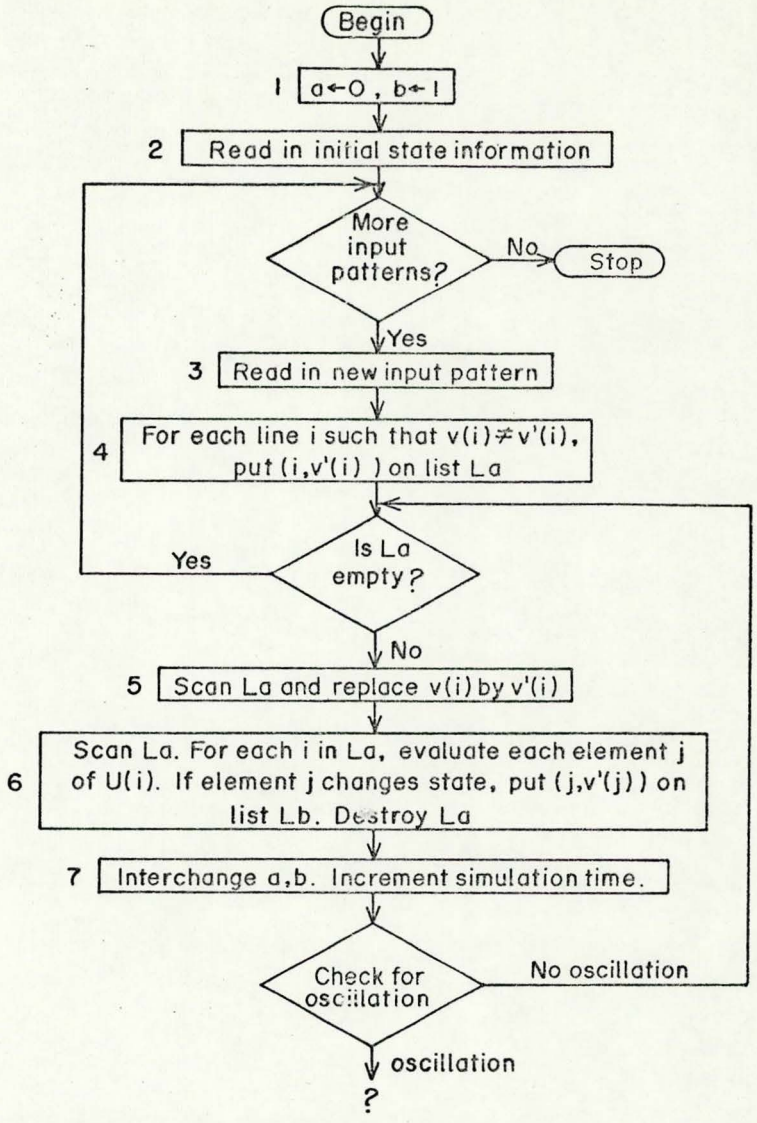
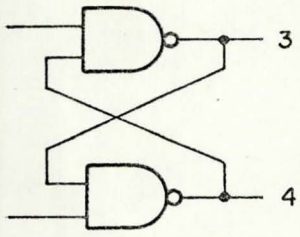


Figure 9 - Simulation algorithm for unit delay model

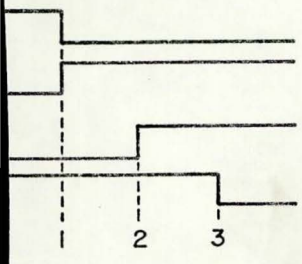
In Fig. 1 we have seen that the zero delay model is not capable of detecting even a simple hazard. In Fig. 11 we see the same circuit as in Fig. 1, assumed to have a unit delay in all gates, and the computations carried out by the algorithm of Fig. 9. We see that the hazard is correctly predicted in line C, corresponding to the 0 to 1 transition in line A. So, since there is a finite delay associated with each gate, the unit delay model allows some race and hazard analysis to be performed. However, the most hazards and races are due to the difference in propagation delays of gates, and this kind of timing condition cannot be detetecd by this model [WAG 84].



a) example circuit

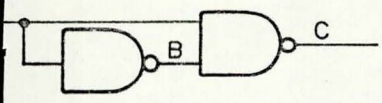
Initialization
1. a←0, b←1
2. v(1)←1, v(2)←0, v(3)←0, v(4)←1
time=1
3. v'(1)←0, v'(2)←1
4. $L_0 = \{(1,0),(2,1)\}$
5. v(1)←0, v(2)←1
6. $L_1 = \{(3,1)\}$, $L_0 = \{\}$
7. a←1, b←0
time=2
5. v(3)←1
6. $L_0 = \{(4,0)\}$, $L_1 = \{\}$
7. a←0, b←1
time=3
5. v(4)←0
6. $L_1 = \{\}$, $L_0 = \{\}$
7. a←1, b←0

b) computations (numbers according to the blocks in Figure 9)



c) timing diagram

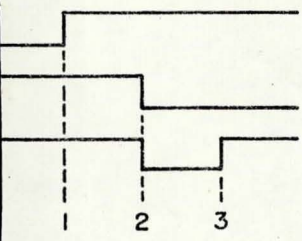
Figure 10 - Simulation run example for the unit delay model



a) example circuit

Initialization
1. a←0, b←1
2. v(A)←0, v(B)←1, v(c)←1
time=1
3. v'(A)←1
4. $L_0 = \{(A,1)\}$
5. v(A)←1
6. $L_1 = \{(B,0),(C,0)\}$, $L_0 = \{\}$
7. a←1, b←0
time=2
5. v(B)←0, v(c)←0
6. $L_0 = \{(C,1)\}$, $L_1 = \{\}$
7. a←0, b←1
time=3
5. v(c)←1
6. $L_1 = \{\}$, $L_0 = \{\}$
7. a←1, b←0

b) computations (numbers according to the blocks in Figure 9)



c) timing diagram

Figure 11 - Hazard detection in unit delay simulation

8. ASSIGNABLE DELAY AND TIME FLOW MECHANISM

The unit delay model can handle some simple cases of hazards and races. However, for a more precise design verification, we need more accurate and realistic timing models. The actual delays in a circuit are distributed over a wide range of values, which can depend on the number of device fan-ins and/or fan-outs or in differences in device technology. In an assignable delay model, we can assign different delays to each device in the network. This has many consequences: 1) the user has to assign these delays in the network input description (or assume that each kind of device has always the same delay, what is not a very realistic assumption); 2) the delay of each device has to be stored in the table structure which represents the network, together with other information about the device; and 3) we need another simulation algorithms.

The assignable delay model implies a simulation algorithm much different from those used in zero and unit delay simulators. When an element i is processed because one of its inputs has changed value (selective trace) and it is determined that its output line will change its value, then this output line must be scheduled to change at time tc + D, where tc is the current simulated time and D the delay associated with the element. This corresponds to an event being placed in a space for future events, and of course this space will have at some moment events scheduled for many different future times. Furthemore, as time proceeds, we must access the next event to be executed, it means, the event in this space which has the smallest schedule time. The technique used to schedule and unschedule events (we shall see in section 10 that this is also necessary in some cases) and to access the next event is called the scheduling mechanism, or the time flow mechanism.

9. TIME-MAPPING

There are no references about distribution of events over time in digital logic simulation. Hovewer, compared to another kind of systems, it can be said that digital systems require in general a large number of events, and that they present a dense distribution, i.e., there are events at nearly all time steps and it can occur any number of simultaneous events at each time step. This kind of distribution leads to a special kind of time flow mechanism which is well suited to digital logic (see Fig. 12). The events are stored in lists, one for each time step, and these lists are pointed out by a linear list (the Time Queue |SZY 75|), where consecutive nodes point to consecutive time steps. Thus, each node in the Time Queue is called a time slot.

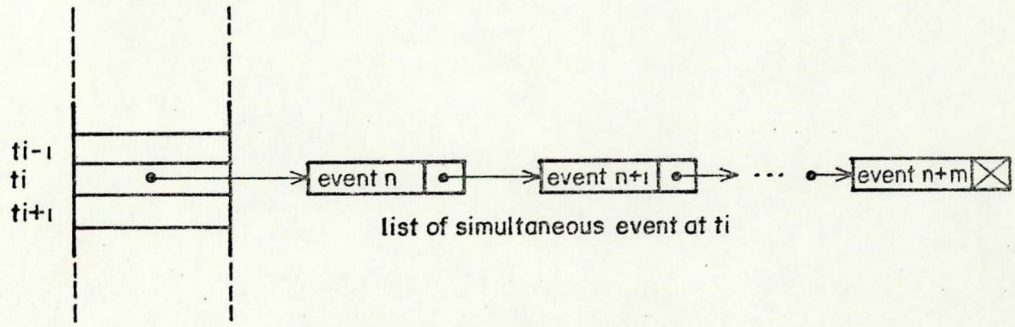


Figure 12 - Time-Mapping technique

As time proceeds, events yet executed are no more needed. A dynamic allocation of memory |KNU 68| is probably needed in association with this time flow mechanism. The events use space from a commom free area, and after they are processed, the space they occupied is returned to this area.

The scheduling mechanism used here is a time-to--address mapping technique. Be dx the delay of the simulated element which output line is detected to change and thus must be scheduled in the Time Queue. Be t the basic time interval between time slots, selected as the greatest common divisor of all element delays. Since dx is a multiple of t, dx/ t is directly added to the current time tc and the result is both the time of the predicted signal change and the address of the time slot into which the corresponding event must be inserted. We will refer us to this time flow mechanism as the time-mapping technique [ULR 69].

Now we have a problem. How long must be the Time Queue, in order to handle all the events which can occur in a complete simulation run ? This certainly will be a prohibitive memory space. We have to limit the Time Queue to some extent, and adopt some algorithm to map events which are scheduled to occur out of the c-rrent time frame of the Time Queue. We will describe two basic approaches to the solution. Recently, more sophisticated time flow mechanisms have been proposed, like the converging lists data structure [ULR 76], but they will not be covered here.

9.1. Timing Wheel

The timing wheel [BRE 76] is a cyclic list of lenght L. Limiting the list length to L, we have a time frame of the timing wheel (i.e., the time interval which currently corresponds to the time slots) from t to t+L-1, where t is the current time associated with the first time slot of the wheel. For events which are to be scheduled out of this time frame, an entry must be made in an overflow events list. Whenever one full rotation of the timing wheel is completed, two things are done: 1) the time frame is

adjusted to t+L to t+2L-1, and 2) the overflow list is scanned and any entries which can now be placed onto the timing wheel are transferred.

As simulated time proceeds, and a time slot t' is reached, the time slots from t to t' are no more needed in this cycle. As t' approaches t+L, less time slots in the current time frame are useful and more events must be scheduled in the overflow list. A solution to avoid this inefficiency is to re-scale the time associated with the time slots each half revolution around the timing wheel. Thus, when t' reaches t+L/2, the time associated with the first half of the wheel is re-scaled to correspond to t+L to t+3L/2-1.

An implemented example of this concept is the Time Queue of the TEGAS-2 system [SZY 75]. It is a list with 2*Z entries (see Fig. 13). Each entry is a pointer to an Event Notice List. The current time slot is pointed by PT (Present Time). The number of cycles yet occurred through the Time Queue is stored in MTC. Each cycle is a pass through the first Z time slots of the Time Queue. Thus, the current simulated time since the beginning of the simulation run is ST = MTC * Z + PT - 1.

The current time frame of the time Queue goes from MTC*Z to MTC*Z + 2*Z-1. Events which are to be schedule out of this frame are stored in the MTEL (Macro Time Event List), together with their simulated time, and ordered by increasing time. The first event in the Event Notice List at time slot Z+1 is always a special event MTEL UPDATE. When PT reaches Z+1 the MTEL UPDATE is executed and cause the following actions:

a) pointers from Z+1 to 2*Z replaces pointers from 1 to Z;
b) pointers from Z+1 to 2*Z are cleared;

c) MTC is incremented;
d) PT is set to 1;
e) a new MTEL UPDATE is scheduled at Z+1;
f) the MTEL is scanned. Each event found which will be in the new time frame of the Time Queue is removed from the MTEL and stored in the Event Notice List which corresponds to the time slot for the event.

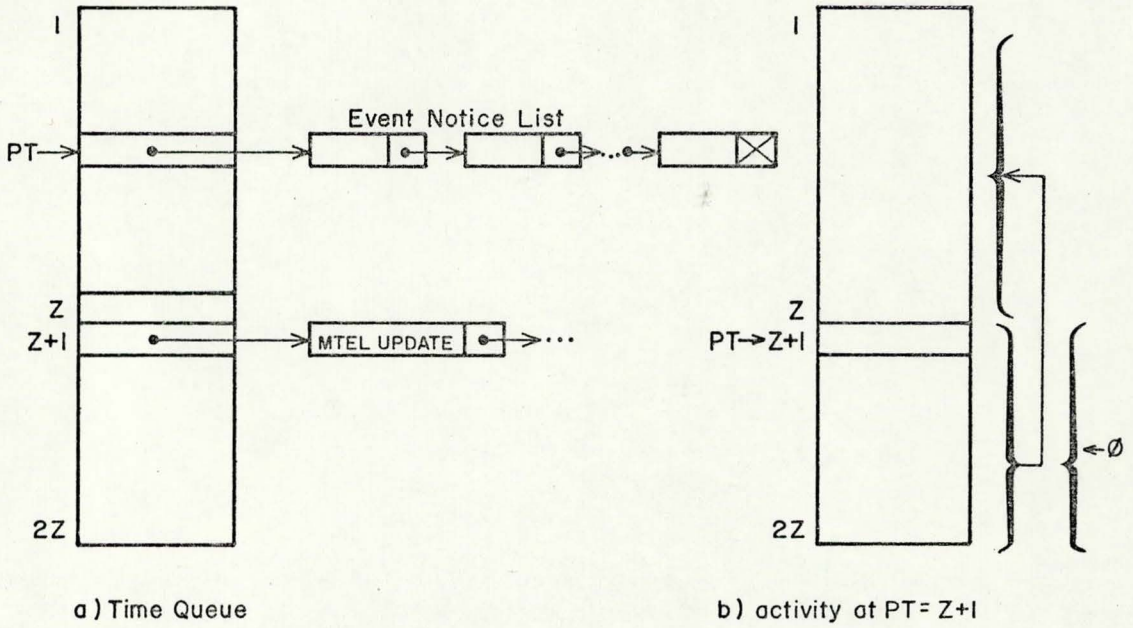


Figure 13 - Time Queue in the TEGAS-2 system

9.2. Δt-loop

The Δt-loop [ULR 69] is also a circular list, where each time slot points to a list of simultaneous events. However, the length L of the loop is made equal to the greater delay in the network divided by Δt, the basic time step. Thus, if **ti** is the current time, and **dx** the delay associated with an element **x** to be be scheduled, the time slot is found by the addition of **ti** and dx/Δt, using a modulo **L** addition, and this time is always within the current time frame, which

goes from ti to ti+L-1. The mechanism really does an automatic and permanent re-scaling of the time associated with the time slots, in contrast with the timing wheel, where the re-scaling is made once each L/2 time steps.

The Δt-loop offers two advantages over the timing wheel. First, there is no need of an overflow list, and therefore we gain in space. Second, there is no need of a special event which transfers events from the overflow list to the timing wheel. This event (the MTEL UPDATE in the TEGAS-2 system) causes an overhead of time which does not exist in the Δt-loop. However, there could be one drawback in the Δt-loop approach, namely, how great must be L? If it is too great, there could be an advantage in the timing wheel, where we have a compromise between time and memory.

Determination of values for Δt and L is left to the user. It's wise to follow these basic rules:

1. all element delays must be of the form dx=Dx* t, where Dx is a positive integer and serves as a delay constant for element x.

2. the Greatest Common Divisor (GCD) of the element delays is the optimum (largest) value for Δt.

3. the maximum element delay constant dmax should not exceed L.

4. L should be kept as small as possible.

10. TIMING MODELS

10.1 Propagation delay

This is the simplest timing condition which can be processed, and is the one we are considering so far in the assignable delay model. The input signal change to a gate is considered to propagate through the gate, and the change appears at the output only tp time units later, as we see in Fig. 14. The signal transitions are assumed to be instantaneous. All the inputs have the same time effect on the output, that is, all have the same propagation delay. Because of this, the gate can be modeled as in Fig. 15, with a functional part free from delay and a delay part associated only with the output. When an input change occurs, the gate is immediately evaluated and the new output value is scheduled as an event at the current time plus the propagation delay.

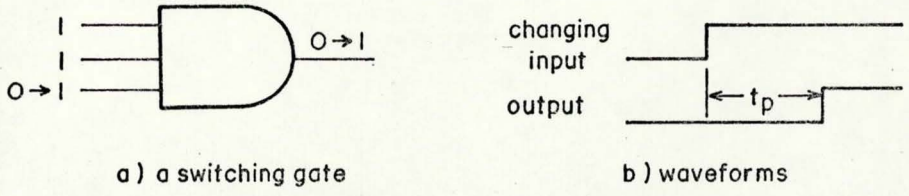


Figure 14 - Propagation delay model

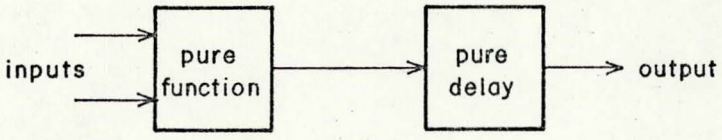


Figure 15 - Gate model with propagation delay

Some gate level simulators allow the representation of flip-flops [EVA 78]. Flip-flops, however, have inputs with different functional and time effects on the output. For example, the clear input forces the output to 0 after a propagation delay tc, the preset input forces the output to 1 after a propagation delay tp, and the clock input causes the output to follow the input after a propagation delay tck. It is obvious that the model of Fig. 15 cannot be used, because the output delay is different for each input. A more realistic model would associate the delays with the inputs, as in Fig. 16. Furthermore, flip-flops have other timing conditions such as data setup and hold times and minimum pulse widths for clock, clear and preset inputs. These conditions will not be covered here. Evans [EVA 78] presents a general algorithm for processing flip-flops with all these conditions. In general, gate level simulators adopt a gate equivalent representation for flip-flops.

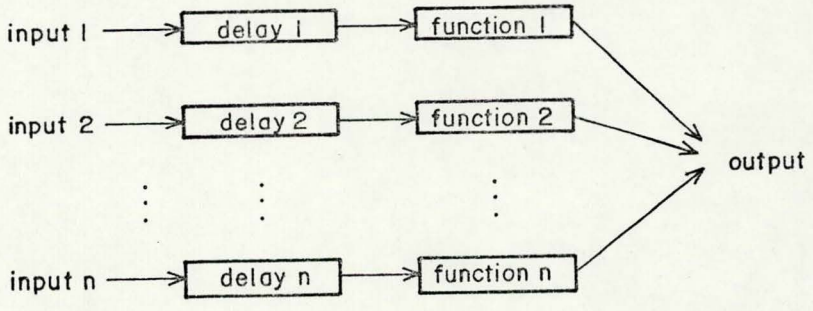


Figure 16 - Model for inputs with different time and functional effects at the output

10.2 Inertial delay

An input signal must have some minimum width to give a device the needed energy in order to force an output change. This minimum width is called the inertial delay [BRE 76] ΔI of the device. In Fig. 17 we see the effect of the inertial delay on a device. If the input signal width is greater than ΔI, the device behaves as it had a propagation delay ΔI. If, however, the input signal time width is smaller than ΔI, the event yet scheduled to occur at $ti+\Delta I$ must be unscheduled and the output remains unchanged.

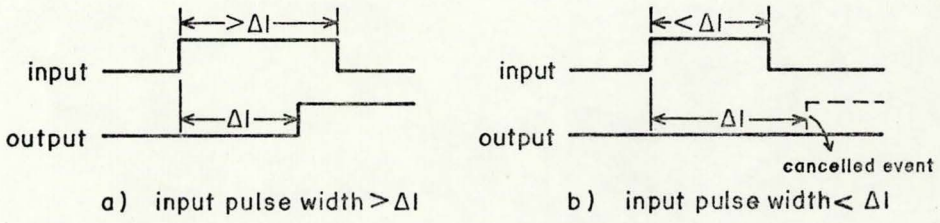


Figure 17 - Inertial delay model

Here a complication arises in our simulation algorithm. When the input goes back to 0 at tj, we must know: 1) that an event is scheduled at $ti+\Delta I$, and 2) that this event must be cancelled. The answer to the second question is easy: always when an input transition makes the future output value equal to the present value. The answer to the first question requires an improvement in the data structure which represents the network and the event. This is shown in the next section.

This inertial delay model is a very simplified one. It can be shown that it does not give the response given by a real gate, when the input changes in very small intervals, sistematically smaller than the gate inertial delay.

10.3 Differing transition delay times [CHA 71]

An improvement in the propagation delay model towards a more realistic modeling is the possibility of assigning different delays to the negative-going and positive-going transitions. These parameters, given in data sheets as tphl and tplh, respectively, are shown in Fig. 18. They are not difficult to model in event-driven simulators. According to the output change direction, an event is scheduled after tphl or tplh time units. However, this model introduces a possibility which is ilustrated in Fig. 19, where some device (with non-inverting logic) has a tplh of 10 time units and a tphl of 5 times units. Suppose that the device receives an input signal like in Fig. 19b. At t1 the input goes from 0 to 1. A corresponding 0 to 1 transition is scheduled to occur in the output at t1+10. At t1+3, however, the input goes back to 0. This would require a 1 to 0 transition in the output scheduled to t1+3+5 = t1+8. At this point, hovewer, the output will be still 0. In other words, the input pulse is narrower than necessary to switch the output. Thus, the event at t1+8 is not scheduled, and the previous scheduled event at t1+10 must be cancelled.

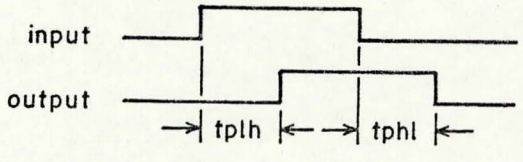
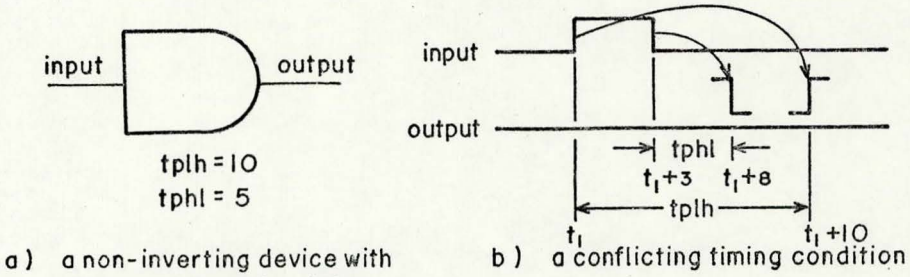


Figure 18 - Different transition delay times



a) a non-inverting device with different transition delays

b) a conflicting timing condition

Figure 19 - A conflicting timing condition in a non-inverting device with different transition delay times

This event cancellation problem can be stated with three questions:

1. when do we know that a possible existing future event must be cancelled?

2) how do we know that there is a scheduled future event for this output? and

3) how do we find the scheduled future event for the output, which must be unscheduled? This event is in some list pointed out by some time slot of the Time Queue, and a linear search through the Time Queue would be time consuming. We present a solution adapted from Tokoro et al [TOK 78].

The events in the Time Queue which belong to the same signal are chained together and a pointer to the beginning of the chain is maintained in the entry corresponding to the signal in the circuit description table (see Fig. 20). The place where a new event must be inserted in the chain is found. To accomplish this objective, each event also has stored the time associated with it. Be PV (Preceding Value) the

value of this signal in the event immediately before the place where the new event must be inserted. If there is no such event, PV is the current output value. Be SV (Scheduling Value) the value of the signal in the new event to be inserted. If PV = SV, then the new event is not scheduled and any subsequent events in the chain for this signal are cancelled.

A very simple solution to the task of unscheduling events is, instead of removing the nodes (that is, adjusting pointers), only to mark the cancelled events with a flag. When the corresponding time slot is processed, the flag is detected in the node and the event is just skipped.

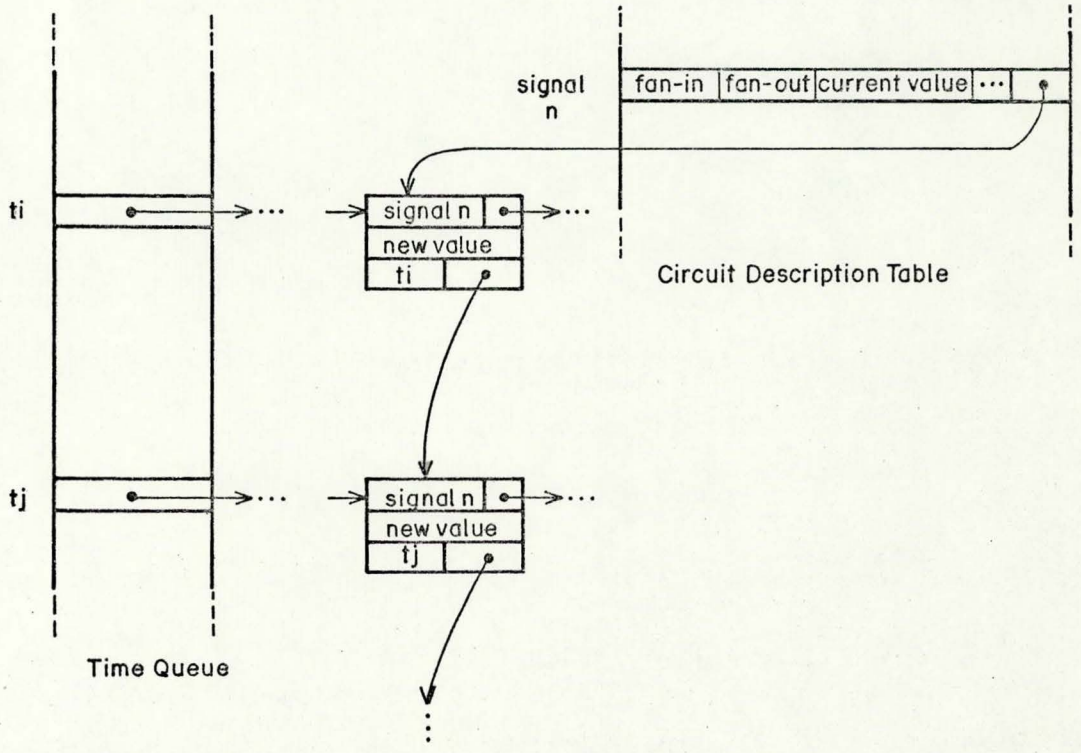
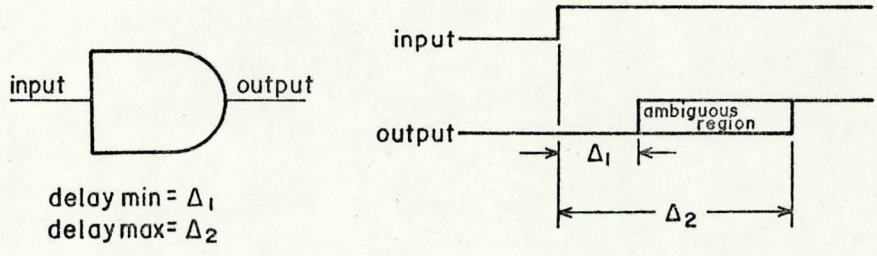


Figure 20 - Table and event structure for event cancellation

10.4 Ambiguity region

The propagation delay model presented in section 10.1 allows the user to select a different delay for each gate. We know from the data sheets that mininum and maximum propagation delays are specified for each type of gate. In simulating a network with many gates of the same type, we could assign for each one a different delay, within the range found in the data sheets. However, before the circuit is built, we do not know what is the exact delay of each gate of the same type, and hazard pulses and races can appear exactly due to unexpected relations among delays in the circuit.

A model suggested to accurately handle this delay interval specified in data sheets assumes, as shown in Fig. 21, that the output response of a device is unknown (or ambiguous) during the interval between the minimum and maximum possible delays. This is known as the delay ambiguity model [SZY 70] [CHA 71]. This model requires that we represent, instead of two-valued signals (binary 0 and 1), three-valued signals (0, 1 and unknown, which we will represent by u), and also that we have a algebra to perform logical operations among signals. This algebra is depicted through 3 operators (AND, OR, NOT) in Fig. 22. The internal representation and evaluation of three-valued signals is considered in section 14.



a) a non-inverting device b) timing diagram for the ambiguity region model

Figure 21 - Ambiguity region model

AND	0	1	u
0	0	0	0
1	0	1	u
u	0	u	u

OR	0	1	u
0	0	1	u
1	1	1	1
u	u	1	u

NOT	0	1	u
	1	0	u

Figure 22 - Three-valued algebra

A simple example (adapted from [SZY 70]) shows that the ambiguity region model can detect a hazard. In the network of Fig. 23a, the inverters are supposed to have a minimum delay of 4 and a maximum delay of 6 time units, while the AND gate has respectively 3 and 5 time units. A 0 to 1 transition occurs in line A at t=0, and a 1 to 0 transition in line B at t=1. Fig. 23b shows the resulting timing diagram if a nominal delay is used for the devices (5 for the inverters and 4 for the AND gate). We see that no output is expected at line E. Fig. 23c shows what happens when we apply the ambiguity region model. A hazard is predicted to line E between times 8 and 11, that is, a hazard pulse could happen in this interval if inverter C had a delay 2 time units greater than inverter D.

The ambiguity region model has one serious drawback. It gives erroneous results (too pessimistic) depending on the network [BRE 76] [BEN 79].

The error is due to two factors:

1. the actual delays of the gates are constant, between the minimum and maximum values. The model, however, implies that the gate, during the same simulation run, can have different values.

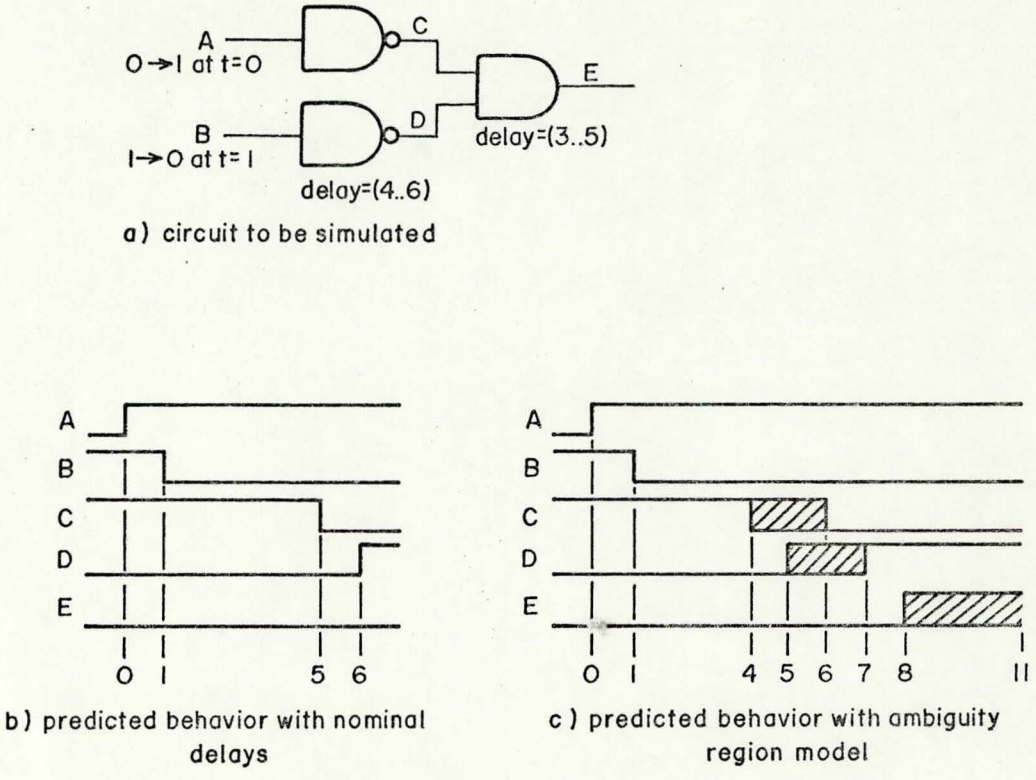
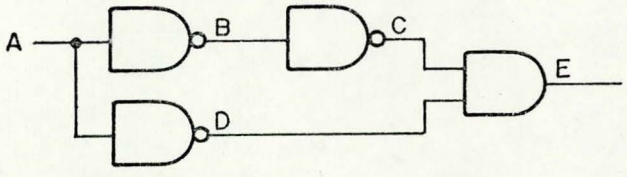


Figure 23 - Hazard detection with the ambiguity region model

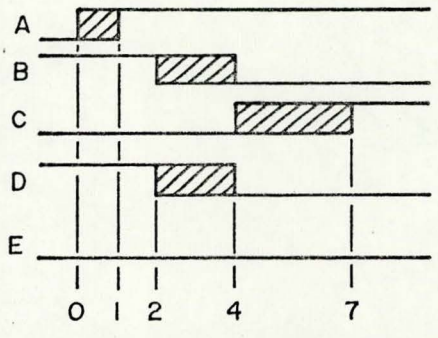
2. a signal transition at some point occurs at a well determined moment within the ambiguity region. The model, however, allows that a signal transition affects different paths which emanate from this point as it occurred at different moments for each path.

This can be seen in the network of Fig. 24a, where all inverters are supposed to have a delay between 2 and 3 time units. Due to the simulation of the previous logic, the input signal A carries with it an ambiguity region. In Fig. 24b we suppose that this region has a width of 1 time unit. We see that no signal change is predicted at output E. In Fig. 24c, however, we assume an ambiguity region of 2 time units for signal A. A hazard is predicted at the output E between times 6 and 8. Thus, this hazard prediction depends on the input

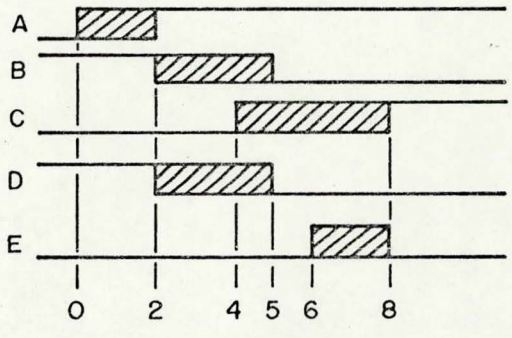
ambiguity interval. This happens because the network has a point of reconvergent fan-out: the output E in this case is the union of two paths emanating from the same source. If the input transition occurs at t=0, signal C can appear earlier at t=4; if the input transition occurs at t=2, signal D can appear later at t=5. Thus, this coincidence of signals C and D between t=4 and t=5 can exist only if the input transition occurs at different moments for each path, what is impossible in reality but is allowable by the model.



a) circuit to be simulated (delays= 2..3)



b) predicted behavior with input ambiguity region of 1 time unit



c) predicted behavior with input ambiguity region of 2 time units

Figure 24 - Pessimistic results using the ambiguity region model

Breuer and Friedman [BRE 76] suggest an alternative to process minimum and maximum delays, using a Monte Carlo approach [GOR 78]. The propagation delay of each type of device is given by some statistic distribution. The delay of each gate could be selected randomly between the minimum and maximum values. For each set of randomly selected delays would be carried

out a simulation run. Of course this method would be exhaustive and time consuming. Magnhagen [MAG 77] shows a much more sophisticated implementation, in which each signal delay is given by a probability function and the gates are evaluated accordingly by a probabilistic calculation.

10.5 Rise and fall times

A further improvement towards a more realistic modeling is to represent and process rise and fall times. In Fig. 25a we have the definition of these two times. A way of modeling them is to assume that, during the transition time, the signal is in the unknown state (yet defined in the last section), as in Fig. 25b.

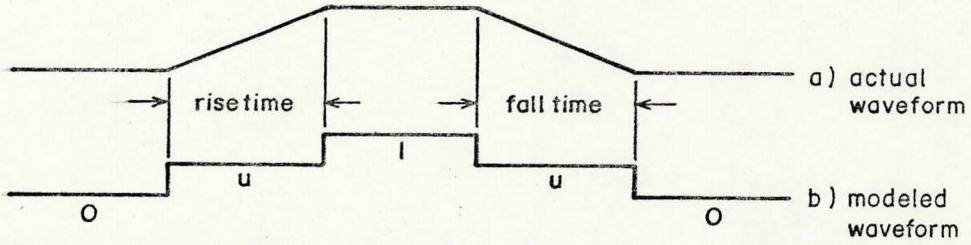


Figure 25 - Modeling rise and fall times

A solution to include rise and fall times in the scheduling mechanism is proposed by Tokoro et al [TOK 78], in conjunction with differing transition delays. All changing signals are supposed to pass through the unknown state. The model, according to Fig. 26 (suppose a non-inverting device), allows the user to assign different values for the delays $\Delta 1$, $\Delta 2$, $\Delta 3$ and $\Delta 4$, which are respectively the propagation delays for the 0 to u, u to 1, 1 to u and u to 0 transitions (not appropriately, these delays are called in the model, respectively, minimum rise time, maximum rise time, minimum fall time and maximum fall time). It is easy to see that the predicted output rise time will be

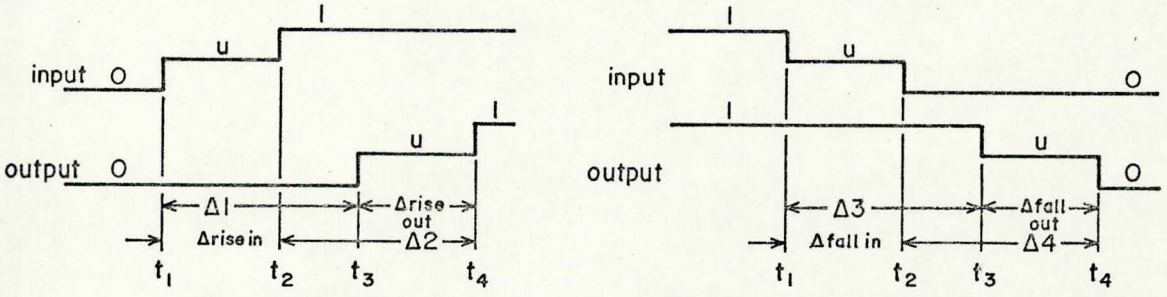
$$\Delta \text{rise out} = t4 - t3 = t2 + \Delta 2 - (t1 + \Delta 1) = t2 - t1 + (\Delta 2 - \Delta 1)$$

$$\Delta \text{rise out} = \Delta \text{rise in} + (\Delta 2 - \Delta 1)$$

and also that

$$\Delta \text{fall out} = \Delta \text{fall in} + (\Delta 4 - \Delta 3).$$

In other words, the output rise (fall) time is equal to the input rise (fall) time plus some specified value $\Delta 2 - \Delta 1$ ($\Delta 4 - \Delta 3$), which can be even equal to zero, if $\Delta 1 = \Delta 2$ ($\Delta 3 = \Delta 4$).



a) scheduling the output rise time b) scheduling the output fall time

Figure 26 - Including rise and fall times in the scheduling mechanism

This model, however, is not accurate, because actually the output transition time is not dependent on the input transition time in such a way. It was suggested [BEN 79] [KOC 69] a much more precise modeling, in which the output transition time for each device type is obtained from a table as a function of the input transition time and the number of loads on the output, as in Fig. 27. One such a table is needed for each transition direction.

input transition time	output loads 1	2	3	⋯	m
X_1	X_{11}	X_{12}	X_{13}	⋯	X_{1m}
X_2	X_{21}	X_{22}	X_{23}	⋯	X_{2m}
X_3	X_{31}	X_{32}	X_{33}	⋯	X_{3m}
⋮	⋮	⋮	⋮	⋱	⋮
X_n	X_{n1}	X_{n2}	X_{n3}	⋯	X_{nm}

Figure 27 - Output transition time table

11. PARALLEL SIMULATION

In gate level simulation, each signal has only 2 possible values (excepting three-valued simulation, which we shall see in section 14), and thus can be stored in only one bit of the host computer. To avoid lost of space, we could think, for a computer with B bits per word, in storing B signals in a word. This, however, would difficult the simulation, because many bit string manipulations would be required, and the processing would be time consuming.

Many alternatives have been suggested to efficiently use all host computer word bits. We could think in processing the same network for B different input patterns in parallel, or for B different initial states in parallel [BRE 76]. Another efficient and more useful solution is to process B faults in parallel [THO 75]. Fault simulation is needed for test generation and is normally provided in gate level simulators. All these three alternatives suppose the same network being processed for B different conditions. The simulation is carried out as if we had B different machines. For these reason, signal values associated with one machine cannot affect the processing of the other machines, and thus we cannot use arithmetic operators (which have carry effect) in the element evaluation, only logical operators which maintain the independence among word bits.

Another solution is used in functional simulation [SZY 73]. In this context, signals are normally associated with values stored in registers of X bits. Since in general all bits of a register are transformed in the same way by the circuit, it would be wasteful to consider each register bit as an independent signal and to process it separately from the others. It is quite natural to store all X bits in a single word and to process them in parallel. Of course we need $X \leq B$, and we will not use the remaining B - X bits.

12. ELEMENT EVALUATION

Always when an output change occurs at a gate, all gates connected to it (its fan-out elements) must be evaluated. There are three basic methods which are used to evaluate gate functions.

The first and more obvious method is to develop a special subroutine (written in assembler or any high-level language) for each device type, as shown in Fig. 28 for an AND gate. The subroutine receives parameters like a pointer to the gate fan-in list, the number of gate inputs and the output signal name. A pointer to the subroutine exists in the table structure, in the entry corresponding to the gate in the circuit description table (see section 15). This method allows parallel simulation (see section 11), but the evaluation time required increases linearly with the number of gate inputs.

```
AND (number-of-inputs, fan-in-list, output-name);
integer number-of-inputs;
array fan-in-list;
boolean output-name;
begin
    integer i, x;
    i ← number-of-inputs; X ← 1;
    while  i ≠ ∅
    do     begin
           X ←X and fan-in-list [i];
           i ← i - 1
           end;
    output-name ← X
end;
```

FIGURA 28 - Evaluation subroutine for an AND gate

The second method counts the number of dominant inputs to the gate [SCH 72]. For example, a 0 input is dominant in an AND gate. If we know how many inputs are equal to zero, we know the output value. This method does not require a fan-in list associated with each gate, only a variable where the dominant input count is stored. The subroutine for an AND gate seems like in Fig. 29. It receives as parameters the input count, the new input value which propagates to the gate, and the output signal name. This method does not allow parallel evaluation, but the evaluation time required is independent of the number of gate inputs.

```
AND (input-count, new-input-value, output-name);
integer input-count;
boolean new-input-value, output-name;
begin
     if new-input-value = 0
     then input-count ← input-count + 1
     else input-count ← input-count - 1,
     if   input-count = 0
     then output-name ← 1
     else output-name ← 0
end;
```

FIGURE 29 - Input count evaluation for an AND gate

The third method is the fastest, but will require a large amount of memory. It uses a table look-up [ULR 72]. Input parameters to the evaluation subroutines, like gate type, new input state and current output state, are compressed into a n-bit argument and used as a pointer to a table, where for each argument value is found an action to be taken. Fig. 30 shows a simple example. We suppose that we have two possible gate types (AND and OR), with only two inputs. A 4-bit argument is formed. The first bit is the gate type, the following bit is the current output value and the last two bits are the new input value. A 16-entry table is then formed. This table can

be much more sophisticated. For example, if we include in the argument information about whether the gate is yet scheduled or not, we could make inertial delay processing. For a device like a flip-flop, with timing considerations such as data setup and hold times and minimum pulse widths and with inputs with different functional effects (clock, clear, preset), a table could be built which could include several schedule and unschedule actions. A further advantage of this method is that all devices in the circuit are evaluated by an unique subroutine.

argument	*action*
0 0 0 0	*nothing*
0 0 0 1	*nothing*
0 0 1 0	*nothing*
0 0 1 1	*schedule output=1 at* t_c+t_{plh} *(AND)*
0 1 0 0	*schedule output=0 at* t_c+t_{phl} *(AND)*
0 1 0 1	"
0 1 1 0	"
0 1 1 1	*nothing*
1 0 0 0	*nothing*
1 0 0 1	*schedule output=1 at* $t_c + t_{plh}$ *(OR)*
1 0 1 0	"
1 0 1 1	"
1 1 0 0	*schedule output=0 at* $t_c + t_{phl}$ *(OR)*
1 1 0 1	*nothing*
1 1 1 0	*nothing*
1 1 1 1	*nothing*

new input value

current output value

0: AND

1: OR

FIGURE 30 - Table look-up evaluation

13. INITIALIZATION

To begin a simulation run, the network must be in some specified initial state. Each signal in the network must be assigned some value. One solution would be to assign the same value (0 or 1) to all signals, but this could lead to some inconsistencies, like the input and output of an inverter set to the same value. The other solution would be a consistent assignment to all signals, but this approach would be very hard to accomplish by the user in large circuits.

One intermediate solution is to assign specified values only to the input signals [EVA 78]. A zero delay simulation is then performed as initialization phase, until the circuit has reached a stable state. After stabilization, the circuit state is displayed and the user can see if a value assignment was made for all signals.

A better and more realistic solution is to assign some specified values (0 or 1) to selected signals in the circuit, and automatically set to u ("unknown") all remaining signals [ULR 72]. As before, a zero delay simulation is performed from this initial state. After stabilization, the user can see if any signals remain at the undefined state. An alternative to avoid a too long initialization is to allow each signal to emit a single u to 0 or u to 1 transition.

It can happen, however, that some specified state is destroyed by the signal propagation during initialization. This can be seen in Fig. 31. If the user assigns a value 0 to signal C, signals A, B and D would be suposed equal to u. After a single transition in each gate, signal D would be set to 1 (a dominant input equal to 0) and signal C to u (inputs 1 and u, output u according to the rules in Fig. 22), and thus the user specified C=0 would be destroyed.

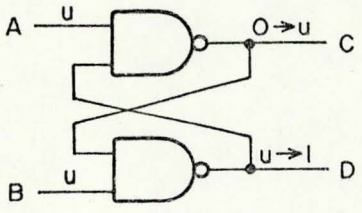


Figure 31 - Inconsistent initialization

The LAMP system [CHA 74] avoids this situation using a 4-valued simulation: 0, 1, 2 (non-propagating unknown) and 3 (propagating unknown). Value 2 is used only in the initialization phase. As we can see in the truth table for a 2-input NAND gate in Fig. 32a, a value 2 in one input maintains the output equal to its previous value, except if we have a value 0 (dominant) in the other input. Thus, value 2 does not propagate through the circuit and does not destroy any user assigned initial state. Value 3 is the normal unknown value, as we can see in Fig. 32b, and is used during the remaining simulation. In Fig. 31, if the user assigns a 0 to signal C, and all other signals are equal to 2, D will be set to 1 (NAND of 2 and 0 is 1), but the feedback will not change C (NAND of 2 and 1 is the previous value).

inputs		output
2	0	1
2	1	Q
2	2	Q
2	3	Q

a) non-propagating unknown

inputs		output
3	0	1
3	1	3
3	2	Q
3	3	3

b) propagating unknown

Figure 32 - 4-valued logic for a stable initialization

14. THREE-VALUED REPRESENTATION AND SIMULATION

In this section we will present three models of internal representation of three-valued variables and evaluation of logical functions with these variables. Of course, it is common to all 3 models the use of two bits to represent one 3-valued signal. However, they differ in how these bits are stored in the host computer words (the word formats), in how they are encoded and how logical operations are performed. In all models we will assume parallel simulation, i.e., we will represent many signals in the same host computer word (see section 11 for the meaning of these signals represented in parallel).

Model 1 [BRE 76][THO 75]

In the first model, the three logical values are encoded as in Fig. 33a, and one signal is stored in each two adjacent bits of the host computer word, as in Fig. 33b. If we have two parallel values A and B as in Fig. 33c, we see that we can directly evaluate C=A.B and C=A+B to obtain logical AND and OR, but when we try to evaluate the logical NOT using bit complementation, we obtain a invalid code 10 as complement of 01. One solution would be, after any complementation, to search for 10 bit pairs and change them to 01. However, this solution requires bit string manipulations and is time consuming. A better solution is to represent the signals as in Fig. 34a, in which the first bit of each bit pair is in the first half-word, and the second bit in the second half-word. This representation does not affect AND and OR operations, but now the NOT operation is simpler, requiring only a word complementation and an interchanging of both half-words, as we see in Fig. 34b.

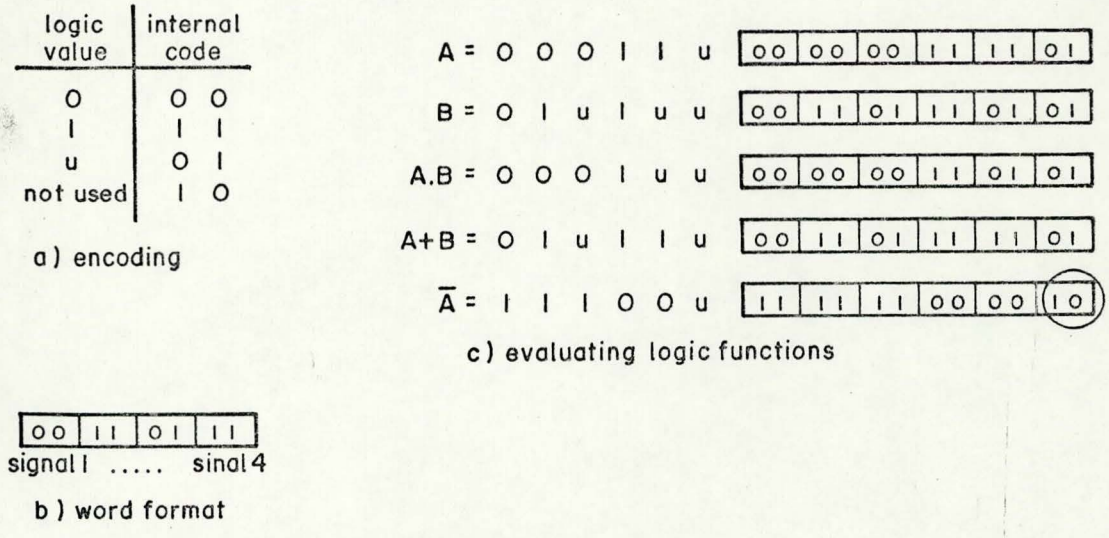


Figure 33 - 3-valued logic, model I

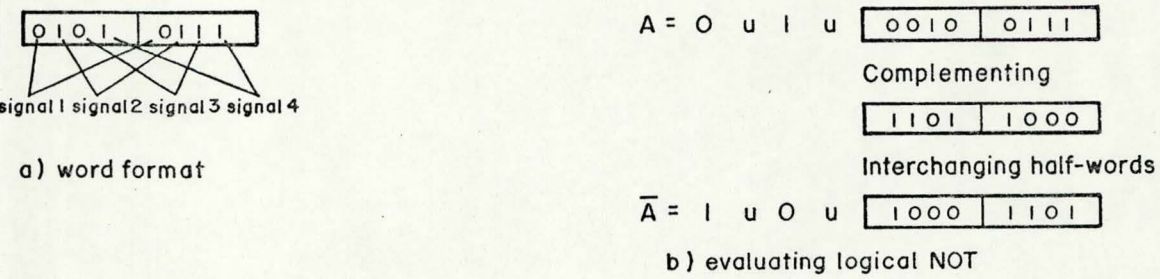


Figure 34 - 3-valued logic, alternative to model I

Model 2 [TOK 78] [THO 75]

In the second model, each parallel value A is stored in two words of the host computer, the Indeterminate Flag Word Ai and the Value Word Av, as in Fig. 35. If one bit in Ai is 0, then the corresponding bit in Av carries the logic value 0 or 1. If the bit in Ai is 1, then the bit value is indeterminated, no matter what is the value of the corresponding bit in Av. As an alternative, we can use 4-valued signals [TOK 78]: 0, 1, "up" and "down". In this case, if the bit in Ai is 1, then the

corresponding bit in Av indicates "up", if equal to 1, or "down", if equal to 0.

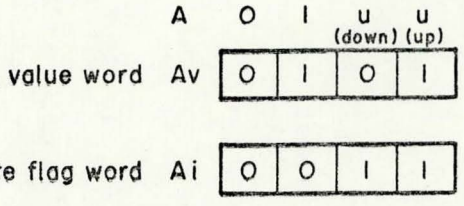


Figure 35 - 3-valued logic, model 2 encoding and word format

Logical function evaluation is a little more complicated than in model 1. The following equations must be used (see examples in Fig. 36):

AND: $Cv = Av \cdot Bv$

$Ci = Ai \cdot Bi + Ai \cdot Bv + Bi \cdot Av$, i.e., the result is unknown if both values are unknown, or if one is unknown and the other is equal to 1.

OR: $Cv = Av + Bv$

$Ci = Ai \cdot Bi + Ai \cdot \overline{Bv} + Bi \cdot \overline{Av}$, i.e., the result is unknown if both values are unknown, or if one is unknown and the other is equal to 0. In the example, the term Ai.Bi seems to be redundant, but it must remain if we represent the unknown by 11 and not by 01, as in the example.

NOT: $Cv = \overline{Av}$

$Ci = Ai$

Model 1 has easier AND and OR operations, while model 2 has an easier NOT operation and an easier translation from the external (user) to the internal representation (and vice versa).

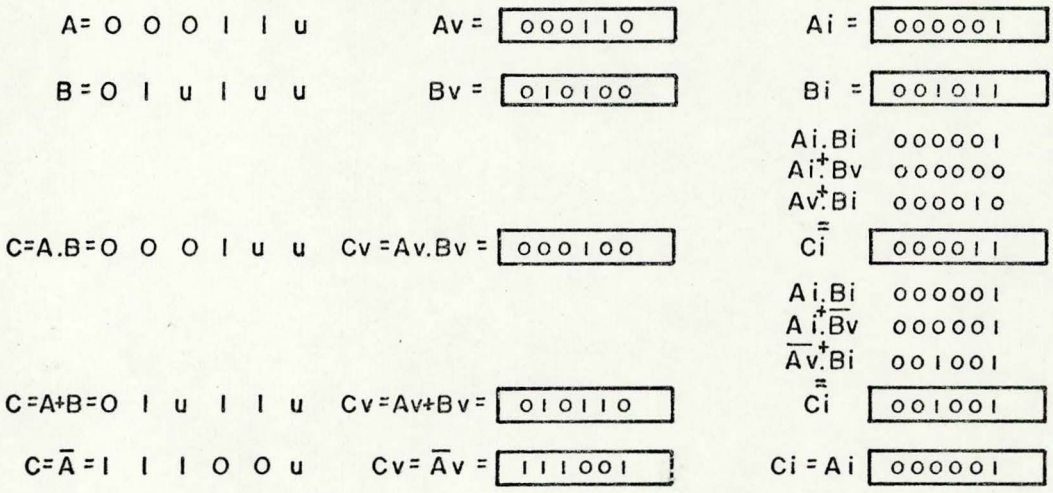


Figure 36 - 3-valued logic, model 2, evaluating logic functions

Model 3 [ALI 78]

Model 3 also uses two words to represent a parallel value, with corresponding bits in the words representing one signal. The enconding is shown in Fig. 37. This enconding allows a simple logical evaluation, according to the following equations:

AND: c1 = a1 . b1, c0 = a0 + b0
OR: c1 = a1 + bi, c0 = a0 . b0
NOT: c1 = a0 , c0 = a1.

Fig. 38 shows the correctness of these expressions.

value	a^0	a^1
0	1	0
1	0	1
u	1	1
not used	0	0

Figure 37 - 3-valued logic, model 3 encoding

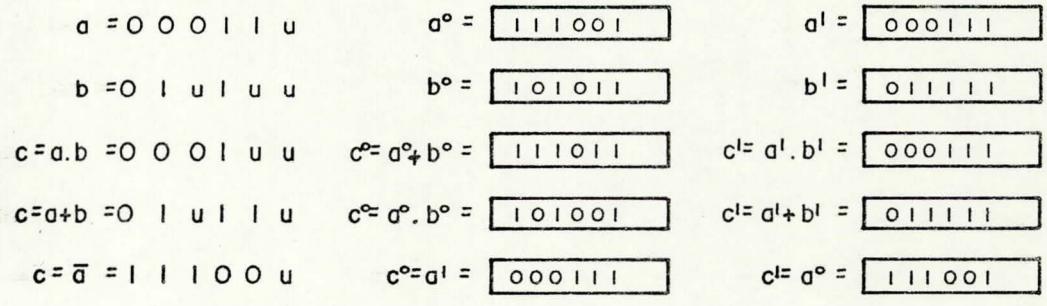


Figure 38 - 3-valued logic, evaluating logic functions

Apart from being the model with easier logical evaluation, this model allows also a very easy calculation of any logical expressions. If we have a general expression $y = f(x1, x2, \ldots, xi)$, we obtain y1 from y and y0 from $\overline{y}$ simply replacing in the expressions for y and $\overline{y}$ each xi by xi1 and each $\overline{x}i$ by xi0, as we can see for an EXOR operation in Fig. 39. It is easy to see that the evaluation expressions for the operators AND, OR and NOT were obtained from this general rule.

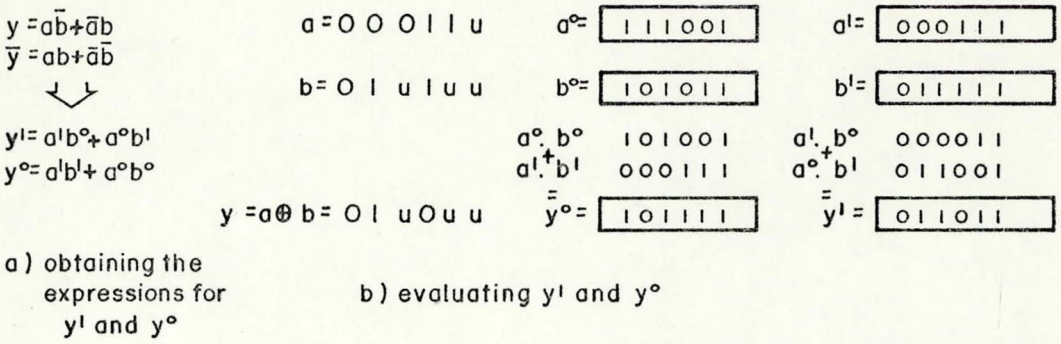


Figure 39 - Evaluating the EXOR function with model 3

15. TABLE STRUCTURES AND SIMULATION ALGORITHMS

We have seen that gate level simulators which make accurate modeling of timing conditions are in general event-driven and have two basic requirements: a scheduling (or time flow) mechanism and a table structure, which reflects the network interconnections and element descriptions. In general this table structure must contain the following information about each gate (we assume that each device, or gate, has only one output signal):

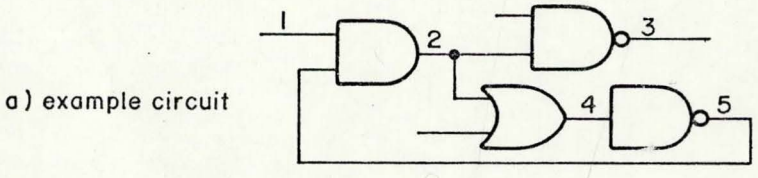
- element type - this acts as an index to the element evaluation subroutine (see section 12);
- logic value of output signal, one bit for 2-valued simulation, 2 bits for 3-valued simulation;
- propagation delay associated with the output - for more sophisticated timing models, see section 10.3 for improvements in the table structure;
- number of inputs - if we consider gates with different number of inputs as instances of the same gate type, then this number is a parameter needed by the evaluation subroutine;
- pointer to the element fan-in list;
- number of fan-out elements;
- pointer to the element fan-out list;

The simulation algorithm will be determined by the nature of the table structure, as we shall see in the next sections.

15.1 Simple data structure [SZY 75]

An immediate way of implementing the table structure with the requirements before listed is shown in Fig. 40. The Circuit Description Table - CDT - has one entry for each gate. The index in the table is the internal name of the gate and by this index is the gate referred in the fan-in and fan-out lists

of other gates. Each entry contains the following information about the corresponding gate: type, output logic value, number of inputs, fan-in list pointer, number of outputs, fan-out list pointer, output propagation delay. The Interconnection Table - IT - contains the fan-in and fan-out lists which are pointed out by the CDT. If we use parallel simulation, the output value is stored normally in a separated Output Value Table, and no information is needed in the CDT about this table if we use the same indexing to identify the gates. Primary inputs, like signal A in Fig. 40, can be considered as dummy gates, which have an output value and a fan-out list, but no inputs or propagation delay.



a) example circuit

index	type	output value	# inputs	fan-in list	#outputs	fan-out list	delay
1	primary input		—	—	1	•	
2	AND		2	•	2	•	
3	NAND		2				
4	OR		2				
5	inverter		1				

2
1
5
3
4

Circuit Description Table

Interconection Table

b) portion of the corresponding table structure

Figure 40 - Simple table structure

The simulation algorithm is depicted in Fig. 41, and has 4 basic steps [TOK 78]: 1) time advancing and output updating value, in which events scheduled to the current time cause updating of output values in the CDT; 2) output

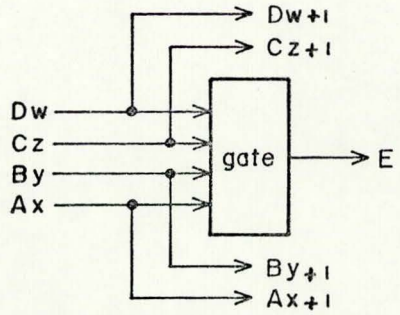
```
Step 1 - Time advancing and output value updating
   clear Propagation-Stack;
   repeat Current-Time ← Current-Time + 1
   until any scheduled event exists in the current time slot
   for every scheduled event in the current time slot
   do begin set the value of the scheduled event for output i into
               CDT (i, output value);
            push the output name i into Propagation-Stack;
      end

Step 2 - Output value propagation
   clear Evaluation-Stack;
   for every output name i in Propagation-Stack
   do begin with CDT (i, fan-out list pointer), and CDT (i, number of
               outputs) find IT (j), IT (j+i), ... IT (j+number of
               outputs);
            push IT (j), ..., IT (j+number of outputs) into Evaluation-
               -Stack;
      end

Step 3 - Element evaluation
   clear Scheduling-Stack;
   for every gate name j in Evaluation-Stack
   do begin with CDT (j, fan-in list pointer) and CDT (j, number of
                  inputs) find the input values (CDT (k, output
                  value)) to gate j;
            with CDT (j, type) and the input values evaluate the gate
                  (find its future output value);
            if future output value ≠ CDT (j, output value)
            then push gate name j and its future output value
                  into Scheduling-Stack;
      end

Step 4 - Event scheduling
   for every gate name j in Scheduling-Stack
   do schedule (gate name, future output value) as an event in the Time
               Queue at Current-Time + CDT (j, propagation delay)
```

FIGURE 41 - Simulation algorithm

propagation, in which we find the gates potentially affected by output changes in the current time step; 3) element evaluation, in which we evaluate all gates potentially affected to determine which of them will really change value in the future; and 4) event scheduling, in which these new calculated output changes are scheduled as events in the Time Queue. The algorithm was clearly separated in these 4 steps only for didatic purposes. Steps 2, 3 and 4 could be carried as a single step and thus we would not neet the Evaluation-Stack and the Scheduling-Stack.

15.2 Descriptor based table structure [ULR 69]

In this table structure approach, each element in the circuit is represented by a descriptor, as in Fig. 42. This descriptor contains the following information about the element: pointers to other destinations (Ax+1 to Dw+1) of the element input signals; pointer to output signal destination; input logic values; output logic values; element type; output propagation delay. The input and output pointers form a cyclic chain for each signal in the circuit, as can be seen in Fig. 43 for a simple circuit. In this example some devices are flip-flops and thus have two output signals. Numbers in parenthesis in Fig. 43a represent memory cells where the respective pointers in Fig. 43b are stored. Take for example element 1. Its output 1 pointer (in memory cell number 1) points to input 1 of element 1 (memory cell 4), which in turn points to input 2 of element 3 (memory cell 14), which points to input 1 of element 4 (memory cell 17), which finally points back to output 1 of element 1. Thus, we have a cyclic chain (shown in Fig. 43b), which passes through all inputs which receive the output 1 signal of element 1. Similar chains exist for all other output signals (for example, also is shown the chain for output 2 of element 2).

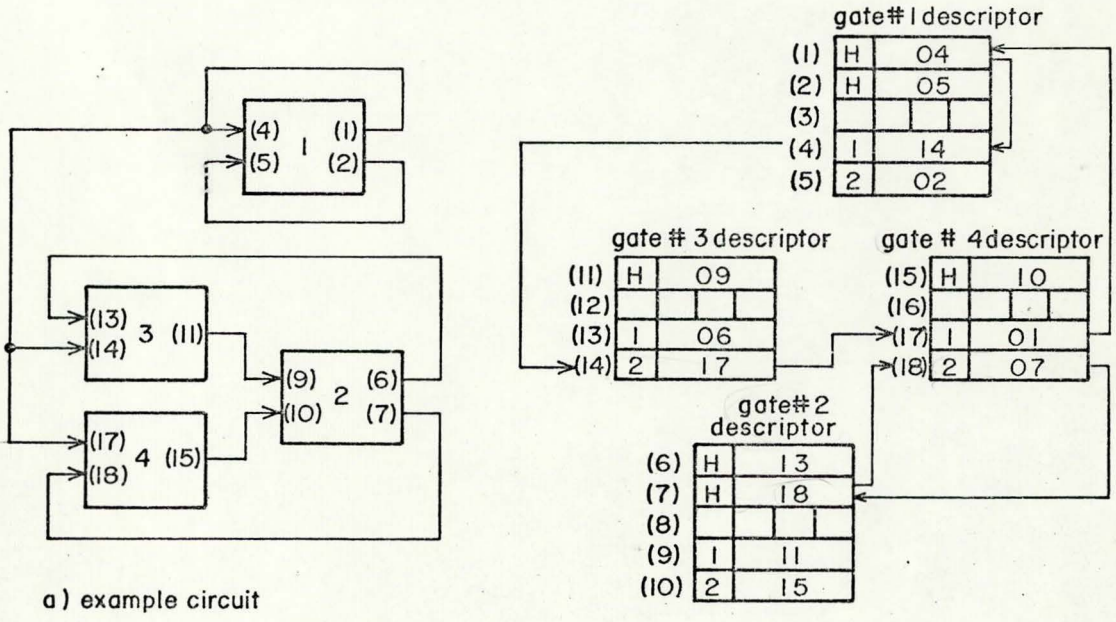


a) a 4-input element

Header	→ E1		
input logic values	output logic value	type	delay
input # 1	→ Ax+1		
input # 2	→ By+1		
input # 3	→ Cz+1		
input # 4	→ Dw+1		

b) data descriptor

Figure 42 - Data descriptor organization



a) example circuit

b) corresponding table structure

Figure 43 - Descriptor based table structure

This table structure introduces the following important simplifications in the simulation algorithm steps of Fig. 41:

Step 4 - Suppose that some output signal AO must be scheduled to time tx. The scheduling is simply performed by exchanging the contents of the first cell of the signal cyclic chain with the contents of the established time slot, as can be seen in Fig. 44a and 44b. Now suppose another signal BO must also be scheduled to tx. The same scheduling process is used, and we see the result in Fig. 44c. The scheduling process cuts temporarily the cyclic chains and transforms them in part of open-ended extendible lists for the time slots of the Time Queue.

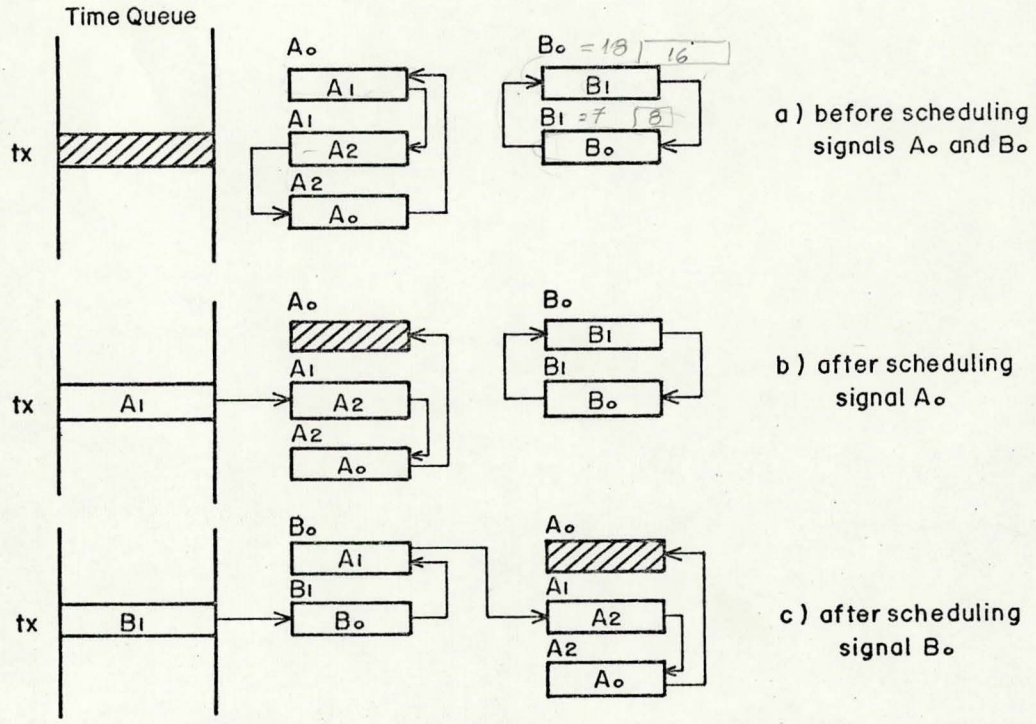


Figure 44 - Scheduling mechanism in the descriptor based table structure

Step 2 - With the list obtained at the time slot, the output value propagation step is automatically performed, because all possible gates affected by the signal changes are now contained in this list.

Step 1 - If we have 2-valued simulation, the output value updating consists in inverting all signals (outputs and inputs) which are found in the list formed at the time slot.

Step 3 - Element evaluation is performed for all destinations found in the list at the time slot. We have stored in each element descriptor a redundant information about the input values. These values could be found if we search the chains, but the redundancy simplifies element evaluation, avoiding the use of a fan-in list.

Acknowledgments

This work was done in 1981 during a three years stay at the University of Kaiserslautern, Germany. I appreciated the support of the University of Kaiserslautern and of the CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico.

REFERENCES

[ALI 78] ALIA, G., CIOMPI, P. and MARTINELLI, E. "LSI Component Modeling in a Three-Value Functional Simulation". Proceedings of the 15th Design Automation Conference, Las Vegas, Jun 1978, pp. 428-439.

[BEN 79] BENING, L. "Developments in Computer Simulation of Gate Level Physical Logic". Proceedings of the 16th Design Automation Conference, San Diego, Jun. 1979. pp.561-567.

[BRE 76] BREUER, M. A. and FRIEDMAN, A.D. Diagnosis & Reliable Design of Digital Systems, Computer Science Press, California, 1976.

[CHA 71] CHAPPEL, S.G. and YAU, S.S. "Simulation of Large Asynchronous Logic Circuits Using an Ambiguous Gate Model", Fall Joint Computer Conference Proceedings, 1971. pp.651-661.

[CHA 74] CHAPPEL, S.G., ELMENDORF, C.H. and SCHMIDT, L.D. "LAMP: Logic-Circuit Simulators", The Bell System Technical Journal, Vol. 53, No. 8, Oct. 1974. pp.1451-1476.

[EVA 78] EVANS, D.J. "Accurate Simulation of Flip-Flop Timing Characteristics". Proceedings of the 15th Design Automation Conference, Las Vegas, Jun. 1978. pp.398-404.

[GOR 78] GORDON, G. System Simulation. Prentice-Hall Inc. Englewood Cliffs, New Jersey, 1978.

[HAR 67] HARDIE, F.H. and SUHOKIE, R.J. "Design and Use of Fault Simulation for Saturn Computer Design". IEEE Transactions on Electronic Computers, Vol. EC-16, No. 8, Aug. 1967. pp.412-429.

[KNU 68] KNUTH, D.E. The Art of Computer Programming. Vol.1, Addison-Wesley, Reading, Massachussets, 1968.

[KOC 69] KOCHLER, D. "Computer Modeling of Logic Modules Under Consideration of Delay and Waveshaping", Proceedings of the IEEE, Vol. 57, No.7, Jul.1969. pp.1294-1296.

[MAG 77] MAGNHAGEN, B. "Practical Experiences from Signal Probability Simulation of Digital Designs", Proceedings of the 14th Design Automation Conference, New Orleans, Jun. 1977. pp.216-219.

[SCH 72] SCHULER, D.M. "Simulation of NAND Logic" Digest of Papers of the 6th Annual IEEE Computer Society International Conference (COMPCON'72), San Francisco, Sept. 1972. pp.243-245.

[SES 62] SESHU, S. and FREEMAN, D.N. "The Diagnosis of Asynchronous Sequential Switching Systems". IEEE Transactions on Electronic Computers, Vol. EC-11, No. 8, Aug. 1962. pp.459-465.

[SZY 70] SZYGENDA, S.A., ROUSE, D. and THOMPSON, E.W. "A Model and Implementation of a Universal Time Delay Simulator for Large Digital Nets", Spring Joint Computer Conference, AFIPS Conference Proceedings, 1970. pp.207-216.

[SZY 73] SZYGENDA, S.A. and LEKKOS, A.A. "Integrated Techniques for Functional and Gate Level Digital Logic Simulation". Proceedings of the 10th Design Automation Workshop. Portland, Jun. 1973. pp.159-172.

[SZY 75] SZYGENDA, S.A. and THOMPSON, E.W. "Digital Logic Simulation in a Time-Based, Table-Driven Environment. Part 1: Design Verification", Computer. Vol. 8, No. 3, Mar. 1975. pp.24-36.

[THO 75] THOMPSON, E.W. and SZYGENDA, S.A. Digital Logic Simulation in a Time-Based, Table-Driven Environment. Part 2: Parallel Fault Simulation". Computer, Vol.8, No. 3. Mar 1975. pp.38-49.

[TOK 78] TOKORO, M. et al., "A Module Level Simulation Technique for Systems Composed of LSI's and MSI's". Proceedings of the 15th Design Automation Conference, Las Vegas, Jun. 1978. pp.418-427.

[ULR 69] ULRICH, E.G. Exclusive Simulation of Activity in Digital Networks", Communications of the ACM. Vol 12, No. 2, Feb. 1969. pp.102-110.

[ULR 72] ULRICH, E.G., BAKER, T. and WILLIAMS, L.R. "Fault--Test Analysis Techniques Based on Logic Simulation". Proceedings of the 9th Design Automation Workshop, Dallas, Jun. 1972. pp.111-115.

[ULR 76] ULRICH, E.G. "Non-Integral Event Timing for Digital Logic Simulation". Proceedings of the 13th Design Automation Conference, San Francisco, Jun. 1976. pp.61-67.

[WAG 84] WAGNER, F.R. "Hazard Detection in Logic Simulation". Internal Report. No. 13, Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação. Nov. 84