Proceedings of the 1980 Winter Simulation Conference
T.I. Ören, C.M. Shub, P.F. Roth (eds.)

SIMULATION OF STRUCTURED HARDWARE DESIGNS

Charles M. Shub
Computer Science Department
University of Vermont

The need for simulating structured hardware designs at the clock pulse level is described. The conceptual components of such a simulator are delineated both for the Algorithmic State Machine which controls the hardware and the hardware components which are controlled. The minimal design restrictions which are made are justified not only in terms of practicality and simplification of the simulator, but also in terms of good structured design practice. The concepts are extended to multiple parallel controllers and loosely coupled controllees. Several examples of simulating a simple hardware design which plays blackjack are presented. Preliminary results and trends in the simulation of a parallel architecture LISP machine are also delineated. Suggestions are made for further improvements and enhancements.

INTRODUCTION

Simulators exist for simulating hardware at the gate level. Simulators also exist for simulating computers at a wide variety of levels of detail from the instruction level on up to much coarser levels of detail. In hardware design, a common technique involves separating the design into a Controller and a Controllee. With such a design methodology one achieves a nice conceptual separation between the task of choosing or deciding what must be done and the task of logically performing the desired operations. Normally, in the design phase, the design of the Controller is expressed in an Algorithmic State Machine flow diagram or, as it is more commonly called, an ASM chart. This ASM chart describes the sequence of command signals to be generated given a sequence of control inputs [5].

In a pragmatic sense, an ASM chart can be viewed as a description of a Finite State Automaton with a set of inputs and a set of outputs. Unlike the Finite State Automaton, however, the inputs are usually related quite strongly to the previous history of outputs by the controllee.

The designer, in developing a design for a piece of hardware involving a controller and a controllee, normally has no nice mechanism for testing his design for correctness. His alternatives usually are to actually implement or to analyze by hand. The first alternative is not desirable for a variety of reasons including:

1) Time and expense involved, particularly if the final implementation will be made using LSI technology.

2) Errors which are found will probably get patched in the hardware itself rather than at the design level.

The second alternative is also not desirable, mainly because of the possibility for errors in the analysis, the complexity of the task, and the difficulty of adequately testing all possibilities.

To further compound the problem, there have recently been several efforts in parallel hardware designs involving two or more quasi independent controllers driving weakly related controllees.

The need then is clear. The designer needs an easy to use, inexpensive mechanism for testing his design prior to actual implementation. The mechanism must allow for modifications at the design level as well as the ability to simulate at the design level.

80CH1617-0/80/0000-0305$00.75 © 1980 IEEE

MODELING A CONTROLLER

The ASM chart is, in a formal sense, the flow diagram of a Mealy-type Sequential Machine [1] as the outputs depend upon both states and inputs, though pragmatically it should be thought of as a Moore-type machine as the outputs essentially occur during the state. The machine is synchronized by a clock pulse. We identify the following four points in the hardware clock cycle:

A) Clock Pulse Edge (E)
  1) The hardware is assumed to be totally stable.
  2) Conditions can be tested to determine (based on input signals to the controller) which path of a state is applicable.

B) Clock Pulse Edge + (E +Δ)
  1) The hardware is not stable.
  2) Controller signals have begun propagating into the controllee effecting the controllee.
  3) Changes starting in the controllee will not affect the choice of path made at clock pulse edge time.

C) Stabilization Time (S)
  1) The hardware has become stable and will remain stable until the next clock pulse edge.
  2) All changes in the controllee have occurred.

D) Signal Off time (E - Δ)--The state signals can be turned off at this point without having any effect on stability at the next clock pulse edge (normally clock pulse edge - Δ).

This identification scheme imposes only one restriction on the design process. This restriction, which is also a good design habit as it reinforces the separateness of the controller and controllee, is that a controller output in any state cannot be directly connected as an input signal to the controller to be used by the controller to decide which path to take in the next state. In other words, the controller itself has no memory of which signals it generated during the previous clock pulse.

The action of a controller can then be simply described:

1) At E, the controller enters a state and selects the flow path from that state based upon controllee signals.

2) At E +Δ, the controller starts to emit its signals from this state.

3) The controller waits until S, the time that the signals it has emitted have caused the controllee to go through its changes.

4) At E - Δ, the controller ceases to emit its signals for this state.

5) Go to step 1.

The action of a controller can then be very simply modeled in a variety of fashions. Consider the controller as a process or (to use GPSS terminology) a transaction. One can then use simulation terminology to describe the controller action. Figure 1 gives a flow diagram for controller actions in such terminology.

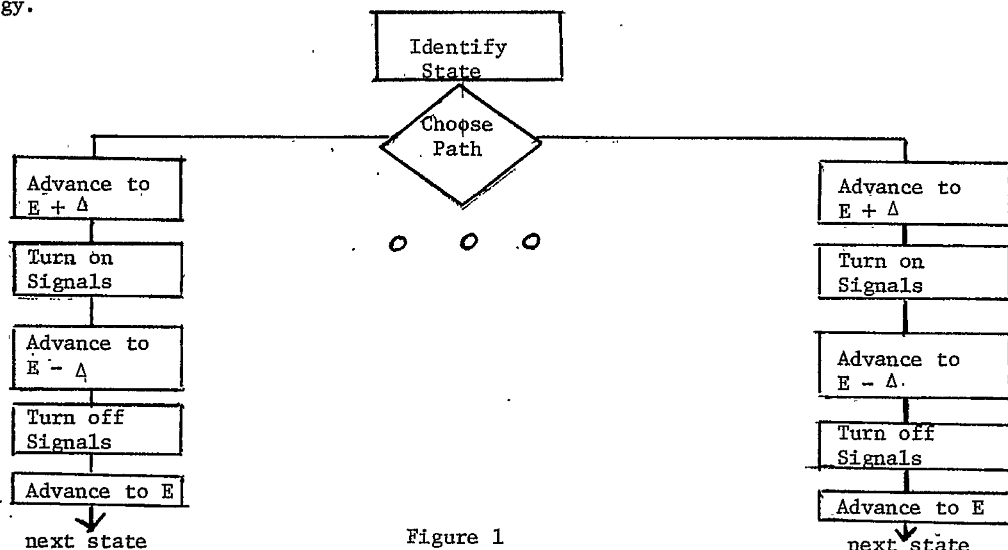

Figure 1

A Controller

MODELING THE CONTROLLEE

The second major step is modeling the changes which take place in the controllee. These changes always occur as a result of the changes in signals presented to it, either from a controller or from some external source. As is common in all design endeavors, we make certain simplifying assumptions. The first such assumption involves that of asynchronous signals received from external sources to the controllee such as, for example, console switches and pushbuttons. We assume that any designer using pushbuttons as an input has taken, or will take, the steps necessary to debounce and synchronize (using standard techniques) all pushbutton signals. With respect to console switches, we insist on a design philosophy where the controller must issue a "load" signal of some sort to "read" the switches, and, futthermore, that such a signal is issued only in response to some sort of "ready" input to the controller. As an example, see [5].

The second assumption we make is that the controllee does not have a hardware loop in it. If it did, then the restriction we have already placed on the controller could be violated at higher clock rates as well as make the design clock rate dependent which is also undesirable.

Analysis of what happens in a controllee is rather straightforward. The effects of the command signals propagate through the controllee at a finite speed. The speed is, of course, limited by the finite switching time of the circuitry and is normally faster that the clock rate of the machine. Were it not, controllee restriction would be violated. The important thing to note is that the final stable states of the various devices in the controllee, if a loop free structure controllee is required, can be easily determined in a sequential fashion. Consider the controllee as a directed graph with the components representing nodes and the signal flow within the controllee representing the edges. View this directed cycle free graph as a partial ordering. Produce an equivalent total ordering [2], and then have the transaction set controllee outputs in the order of this total ordering.

In a number of cases, common controllee structures which appear to have loops within them are actually loop free. As an example, consider the design shown in figure 2.

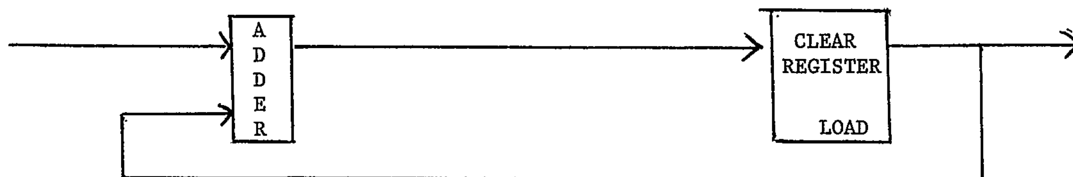

Figure 2. A Typical Controllee Portion

Essentially, the sequence of actions is as depicted in figure 3.

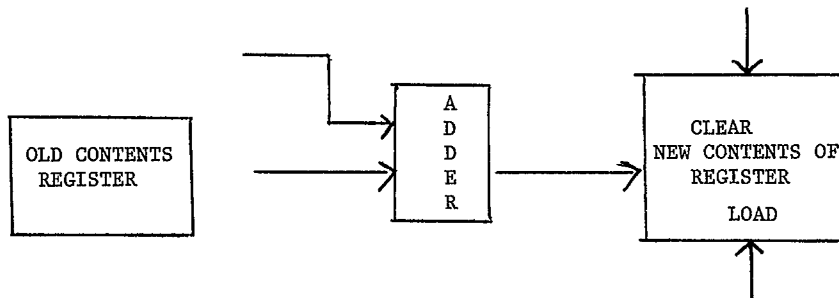

Figure 3. Loop Free Realization

The fact that a load or clear signal must be present to change the contents of the register dictates that the old register output is added to the other signal, and then the results of this addition are loaded into the register.

The controllee then can be viewed as a sequence of actions which start after $E + \Delta$ and end at S. This, along with an explanation of how to effect the transformations on each hardware component, is sufficient to allow simulation of the controllee. One models this sequence of actions in the same fashion as a controller by providing a process or transaction to effect the changes. The changes are summarized by describing for several devices the input to output transformation.

1) A Flip Flop
   A) Inputs Set Signal
      Reset Signal
      Toggle Signal

```
2) A Comparator
  A) Inputs     Two Signals
  B) Output
       If Input Signals Are in the Proper Relation Then True
       Else False

3) An Arithmetic Logic Unit
  A) Inputs    Operand 1 Signal
               Operand 2 Signal
               Function Signal
  B) Output
       Operation 1 Function Operation 2

4) A Multiplexor
  A) Inputs    Signal 0
               Signal 1
                  :
               Signal N
               Selector Value
  B) Output
       The Appropriate Input Value

5) A Demultiplexor
  A) Inputs    Input Signal
               Selector Value
  B) Output
       Signal 0 If Selector Value Not 0 Then False Else Input Signal
       Signal 1 If Selector Value Not 1 Then False Else Input Signal
          :
       Signal N If Selector Value Not N Then False Else Input Signal

6) A Gate
  A) Inputs     Arbitrary Signal
  B) Output
       Appropriate Function Applied to Inputs
```

IMPLEMENTATION

With the components of the design of a controller and the controllee, the implementation of a computer program is rather straightforward. A simple hardware design which plays the game of blackjack, (see figures 4 and 5) was modeled and programmed first in GPSS and then in ALGOL. In both cases, the programming effort was minimal requiring about a day of effort to implement in GPSS and a short afternoon to recode in ALGOL. (Perhaps the level of effort was related to the order.)

In GPSS, the technique was to provide two transactions, the first of which would implement the controller and the second of which would implement the controllee. A clock pulse time of 10 GPSS time units was used. The sequence of coding for the controller tranaction is given in Table 5 and that for the controllee in Table 7. Note that the simplifying assumption has been made that times S, $E - \Delta$, and E are all the same GPSS clock point in time in the GPSS model. Such an assumption is valid because, at time S (in the program), the controllee will have completed the propagation of the signals thus producing the commands for the next state. The signals can then be turned off $(E - \Delta)$ immediately. In the simulation, $(E - \Delta)$ can be equated with E because of the sequential nature of transaction flow [4].

TABLE 6

GPSS Controller Coding Skeleton

```
 STATE    SAVEVALUE    CNTRL, <STATENO>, B    SAVE the state for printing
*
*         LOGIC TO SELECT A PATH
*
          ADVANCE      1                       Time E + Δ
          SAVEVALUE    SIG1,1,H              § turn on signals
              :                              §
          SAVEVALUE    SIGN,1,H              §
          ADVANCE      9                     § time S and E - Δ
          SAVEVALUE    SIG1,0,H              §
              :                              §
          SAVEVALUE    SIGN,0,H              § turn off signals
          TRANSFER     ,NEXT
```

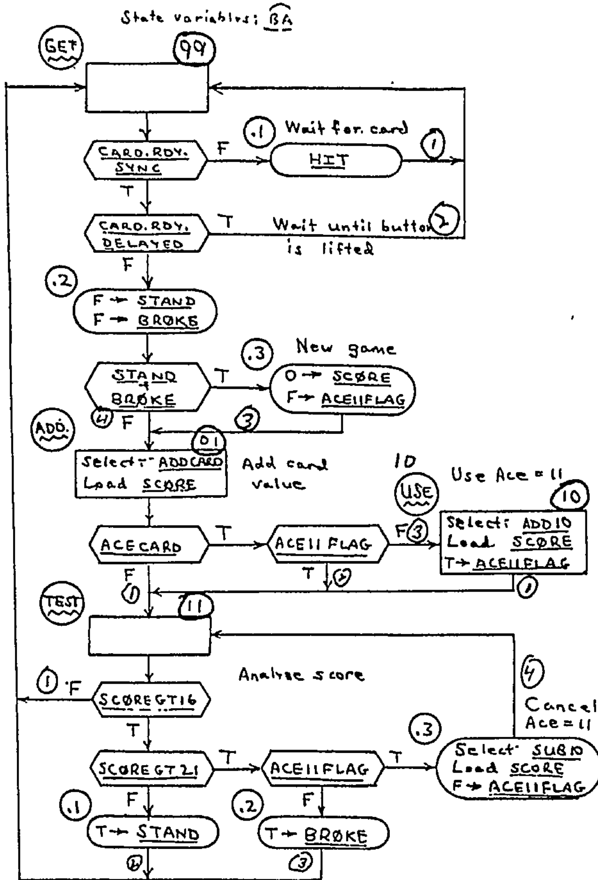

Figure 4

Black Jack Dealer ASM

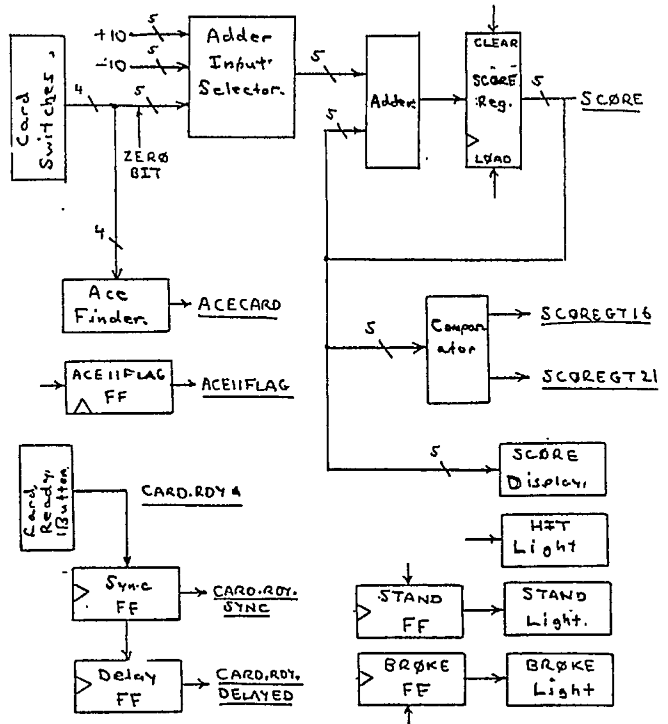

Figure 5

Black Jack Dealer Functional Architecture

TABLE 7

GPSS CONTROLLEE SKELETON

```
  LOOP      ADVANCE        10                       ANOTHER CLOCK PULSE
*
*           EFFECT CONTROLLEE ACTION
*
            PRINT          ,,X          display commands input to controller
            PRINT          ,,XH         display signal output from controller
            PRINT          ,,XB         display controller state
            TRANSFER       ,LOOP
```

Run in interactive mode, the user can effect toggle switches and pushbuttons at any point.

The major drawback of the GPSS implementation is that the nature of GPSS does not permit convenient display of outputs on a periodic basis. For that reason, the simple controller was implemented in ALGOL. Table 8 gives the skeleton of the ALGOL program.

TABLE 8
ALGOL Implementation

```
Procedure Display;
     BEGIN  Comment Display States and Interact With User;
     END;

Procedure Controllee;
     BEGIN  Comment Implement Controllee Action;
            Display;

     END;

State:  If Condition for this path then
          BEGIN
              Turn on Signals;
              Controllee;
              Turn off Signals;
              Go to Next State;
           END
```

In the special case of a single controller, implementation in almost any procedure oriented language can be effected in a manner similar to that described for ALGOL [3]. Such an implementation depends upon the sequential nature of flow which enforces the sequence:

1) Select the path.
2) Turn on the applicable signals.
3) Effect the controllee changes.
4) Display what is necessary.
5) Interact with the designer.
6) Turn off the signals.
7) Go to the next state.

SYNCHRONOUS PARALLEL CONTROLLERS

With recent research in the area of parallel hardware, the design aid technique delineated above seems natural to use in parallel controller designs as well. This section considers the conceptual components of such an extension. Intuitively, one should be able to model several controllers in parallel providing (in GPSS terminology) a separate transaction for each controller. One would then be able to have each controller go through its loop essentially in parallel.

One must make the restriction that all controllers are driven by a common clock. If this restriction is not made, the situation is more complex. Hartmanis and Stearns [1] show formally that the connection of several parallel machines is, in fact, a machine whose state set is the cartesian product of those being connected. Given this, the modeling task is an almost trivial extension of modeling a single machine. The advantage is that the designer can conceptualize what he wants to so much more clearly by specifying the functions of each component separately rather than being concerned with keeping track of all the facets of the design at once.

From a simulation point of view, the crucial need is to be able to specify parallel processes. This requires, if one does not want to become overburdened with programming details, a language which allows for specification of several parallel processes. Simulation languages are well suited for this task. In GPSS, for example, one can provide a separate transaction for each controller. The only necessity is that all transactions make their decisions concerning path selection before any transaction starts turning on signals for the controllee transaction to process. Table 9 gives the sequence of operation.

In terms of the restrictions that are implicit on the overall design, the only one which can cause trouble is that of consistancy. This is best illustrated with an example. Consider a flip flop F. Suppose one parallel machine in state S_1 emits a signal to set F and another machine in state S_2 emits a signal to reset F. What happens to F when the first machine is in state S_1 and the second is in S_2? If one is designing on a whole machine basis, such an anomaly will not occur. Thus, the problem is one of specifying rules for design consistancy rather than coping with the problem. The task of simulating the controllee will point out that such inconsistencies do exist, and the design can be modified at that point, perhaps by decomposition into more parallel controllers. Such a direction is not always going to be proper though. In fact, it is a bonus of the design methodology coupled with the simulation that the designer has the ability to explore, through the simulation model, the relative complexity of actually implementing the separate controller or combining them on paper to realize the whole machine as a single controller.

TABLE 9
Parallel Controller Modeling in GPSS

```
Controller 1, Controller 2 . . . Controller N            Controllee

Select path
Delay to E + Δ
            Select path
            Delay to E + Δ. . .
                               Select path
                               Delay to E + Δ

Signals on
Delay to S
          Signals on
          Delay to S . . .
                               Signals on
                               Delay  to S               Perform Actions

Signals off
Next state
          Signals off
          Next state . . .
                               Signals off
                               Next state
```

ASYNCHRONOUS PARALLEL CONTROLLERS

Finally, we consider the task of modeling at the clock pulse level a number of independent controllers which are running asynchronously, perhaps at different clock rates. The designer should and must synchronize his machine via standard "hand-shaking" procedures [5,6]. The design then will involve each controllee having the appropriate "hand-shaking" hardware which serves to (from the point of view of the host machine) synchronize the "external" signal from the other machine to the clock rate of the host and for the host to emit what it considers a synchronized (to its own clock rate) signal which will be treated as an asynchronous external signal by the other machine. The design task is then rather straightforward, but the simulation becomes much more complex.

The simulator is faced with the dilemma of needing to specify an asynchronous interaction. The simulation modeler can use one of two alternatives. He can code with different delays representing different clock rates, or he can introduce random delays to model the asynchronous nature of the problem. Neither solution is entirely satisfactory. The problem is that in the simulation either technique can lead to a situation which violates the rules delineated in the previous sections. For this reason, we do not currently advocate the use of this technique for modeling several asynchronous machines.

SUMMARY

The author has presented a mechanism for modeling simple structured hardware designs consisting of either a single controller or several synchronized parallel controllers subject to the following minimal design restrictions:

1) No controller output can be used directly as a controller input.
2) The controllee designed will stabilize its response to controller signals within a time less than the clock rate of the machine.
3) The design is independent of the clock rate.

Preliminary investigations have shown that the resultant technique can be used to

1) Check out the paper design for correctness.
2) Allow for modification to be implemented as design changes rather than hardware patches
3) Provide for a structured functional level of design of separate pieces of the device independently by use of the parallel decomposition results in [1].
4. Develop with minimal effort a computer program to actually perform the simulation.

FURTHER EFFORTS

It is clear that this paper does not provide the pot of gold at the end of the designers rainbow. There are two major areas which should be pursued. First is the problem of simulation of several asynchronous machines as a unified design. While one can currently bypass the problem with brute force techniques, a more logical approach is necessary. Secondly, further effort can be expended into making the simulation program construction more palatable. The current efforts involve using Macro techniques in GPSS to program the design from design chart form to computer program. This

area of the simulation process can be subject to errors, particularly when the GPSS restriction on variable names are taken into account. In addition, the output could be arranged in a more palatable form. The author feels, however, that such efforts are more the province of a software engineer than a modeler

REFERENCES

1) Hartmanis, J. and Stearns, R.E. "Algebraic Structure Theory of Sequential Machines", Prentice-Hall, 1966.

2) Knuth, Donald E. "The Art of Computer Programming; Volume 1: Fundamental Algorithms", Addison Wesley, 1969.

3) McCracken, Daniel D. "A Guide to ALGOL Programming", Wiley, 1962.

4) Schriber, Thomas J. "Simulation Using GPSS", Wiley, 1974.

5) Winkel, David E. "Digital Logic and Computer Architecture",(to be published).

6) Winkel, David E. Private Communication, March, 1979.