*Scientia Iranica,* Vol. 14, No. 6, pp 566–570


SCIENTIA IRANICA

*Research Note*

# A Hardwired Discrete Simulation Algorithm

**M.H. Nojumi**[1]

The architecture of a hardwired simulator for implementation of a discrete event-driven simulation of digital systems at the logic level is presented. In the design of this system, attempts have been made to utilize techniques of high performance computing to have a system capable of simulating the digital circuits rapidly. The centralized event-driven simulation algorithm chosen here, has the advantages of being efficient and conceptually straightforward. The high reliability of the simulator has been taken care of through a collection of handshake signals between each two of the three main modules.

## INTRODUCTION

Simulation is, today, the most frequently performed operation in the computer aided design of systems. It enables designers to test the designed system for defects and to make necessary modifications and improvements without the need to go through the expensive process of constructing a prototype [1,2]. Advances in computer programming paradigms, like object oriented design, have seen parallel advancements and implementations of simulation algorithms [3].

The ever-increasing complexity of digital systems makes it necessary to develop faster simulators for testing the designed circuitry. One acceleration strategy is to employ more efficient algorithms, which are not easy to think of and come by. An alternative approach is to incorporate the simulation algorithm into the hardware, thereby having a "special-purpose" computer, specifically configured to perform simulation. The simulation algorithm will, thus, run faster, since software compilation and execution will not be required.

This work is based on the latter idea of having a "hardwired" simulator for discrete event-driven simulation of digital systems at the logic level. In its design, attempts have been made to utilize techniques of high performance computing to have a system capable of simulating the digital circuits rapidly. For its efficiency and conceptually straightforward algorithm, the centralized event-driven simulation has been chosen.

It is important to note that the focus of this paper is on describing a simulation paradigm at the conceptual and logic level for event driven simulation, rather than hardware details. For this reason, all physical implementation issues have been deliberately left out of this discussion.

The architecture of the accelerator consists of three main modules: Element Evaluation Unit (EEU), Event Scheduling Unit (ESU) and Event Processing Unit (EPU). These parts perform the three operations involved in centralized-time discrete event-driven simulation. In this way, the simulation algorithm has been partitioned into three main tasks, each allocated to one main module. These modules form a kind of pipeline and operate in parallel to accelerate the simulation process. The high reliability of the simulator has been taken care of through a collection of handshake signals between each two of the three main modules.

## HOST AND AUXILIARY UNITS

The Host is the interface between the simulation engine and the user. It receives a description of the system from the user, performs relevant compilation and preprocessing, if needed, and, then, delivers data to the simulator. It also receives simulation results from the simulator, performs post processing operations, if needed, and, then, submits results to the user in convenient forms (like graphs, simulation charts, etc.) by displaying them on a monitor or storing them on disk.

Two signal lines leave the Host and two enter it:

- "Start" signal leaves the Host and goes to EEU, ESU and EPU. This signal is activated at the start of the simulation and orders the units to start their operations;

1. *Department of Mathematical Sciences, Sharif University of Technology, Tehran, I.R. Iran.*

- "Stop" signal is sent by STI to various units, including the Host, when the simulation time reaches its predetermined final value, informing units that simulation has terminated;
- "CycleComplete" signal is sent by ESU to the Host to inform it that the current simulation cycle has been completed. Upon reception of this signal, the Host reads the current simulation time from the CTR and the values of the nodes requested by the user from EOT;
- "Continue" signal is sent by the Host to ESU. Having finished reading the necessary data from EOT and the CTR, the Host activates this signal, indicating for the ESU to continue.

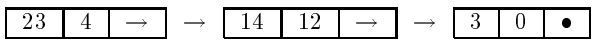
A possible architecture for the simulator includes the following data structures.

Circuit Topology Memory (CTM) is a memory module, in which the description table of the system, or equivalently, the "system's topology", is loaded by the Host. CTM can be an array of $2^{16} \quad 1 = 65535$ cells, each corresponding to one element in the system, making the simulation engine capable of simulating systems with up to 65535 elements, with one code ("0000Hex", for instance), denoting "no element". A element number can then be coded with 16 bits. During the processing of an event, the EPU determines the numbers of activated system elements by referring to this topology memory.

Assuming that elements of the system have, at most, 16 outputs, the cell in CTM, at a location with the address, $s$, is, thus, an array of 16 pointers to the fanout lists of elements with the number $s$. So, CTM$[s][i]$ is a pointer to the list of element numbers of elements that receive, as input, the output, $i$, of an element with the number $s$. The fanout list can be implemented as a linked list, with each node having the structure:

| Element # | Input # | Pointer to next Node |
|---|---|---|

For example, if output number 2 of element number 37 leads to input number 4 of element number 23, to input number 12 of element number 14 and to input number 0 of element number 3, then, CTM [37,4] is a pointer to the linked list:




Element Class Table (ECT) specifies the class to which each element belongs, which, in turn, specifies the functionality of the element. For instance, all $OR$ gates in the system belong to the "$OR$ class" (a collection of elements that performs the OR operation with various "generalized" interpretations of $OR$ in terms of logic "strengths", to be defined in any way the system designer wishes). During the compilation of the system, the Host loads the ECT with the class numbers of elements in the system. The Host may also read the ECT, perform some transformations and store the ECT, on disk, for future reference or future simulations. The 8-bit class number of an activated element is read from the ECT by the EEU for evaluation of that element, as described later.

Class Delay Table (CDT) contains the delay time of each class of elements in, for instance, picoseconds. With each 8-bit entry, delay time between 0 and 255 picoseconds can be recorded. A class with delay time longer than 255 picoseconds can be represented by a cascade of two or more classes. With a 8-bit class code, there are 256 possible classes, making the CDT a $256 \times 8$ bit memory unit.

During the compilation of the system by the Host, CDT is loaded with delays of element classes. The data in CDT can also be read by the Host for various computations or for storage on disk or for later retrieval for future simulations to be done without the need for recompiling the system to be simulated. EEU refers to CDT to get the delay of the evaluated element, which is used in the construction of new events.

Element Output Table (EOT) contains the signal value of each output of each element in the system. With a 16-bit element number and a 4-bit output number, the EOT is a $65536 \times 16$ dmatrix, with EOT$[r, c]$ (element of EOT at row $r$ and column $s$) being the 4-bit value of the output $c$ of element $r$.

In the processing of each event, EPU updates the changed output in EOT. After evaluating an activated element, EEU reads the previous values of the outputs of the element and compares them with the present values of the outputs, to determine whether an output has changed. The Host gathers the simulation results (that is, the values of the nodes requested by the user) by reading from the EOT at the end of every simulation cycle.

Element Input Table (EIT) contains the signal value of each input of each element in the system. With a 16-bit element number and a 4-bit input number, the EIT is a $65536 \times 16$ matrix, with EIT$[r, c]$ (element of EIT at row $r$ and column $s$) being the 4-bit value of the input $c$ of element $r$.

In the processing of each event, EPU updates the inputs in the fanout list of the changed output by loading the new values into the EIT, in locations corresponding to those inputs. EEU reads the input values of an activated element from the EIT, in order to "evaluate" that element (that is, to determine its outputs with the current inputs). For diagnostic purposes, the Host can read input values of various elements from the EIT. Before the start of the simulation, the values of main inputs of the system are loaded into the EIT by the Host.

In the processing of each event, EPU puts the

numbers of the "activated" elements (that is, elements with at least one changed input) in the Activated Element List (AEL), which is, then, read by EEU, one by one, for evaluation. The size of AEL depends on the activity level of the systems one wants to simulate.

It is important to recall that event-driven simulation (an algorithm selected in this architecture) is an efficient simulation algorithm for low-activity systems where, at each instance, a small percentage of system elements experience change in their inputs [2]. For simulation of high-activity systems, other simulation algorithms should be employed [4,5].

In the initialization process, the Host loads the AEL with the numbers of "zero-rank" elements; that is, those elements whose inputs are all main system inputs. EEU will start by evaluating these elements (initially activated elements). After reading an element number from AEL, EEU writes "0000" (the "unknown" logic level) in the same location, to make it ready for the next simulation cycle.

Evaluation Flag Memory (EFM) is an array which indicates whether the number of an element has entered in the AEL; that is, whether the element is activated, with EFM[$s$] = 1 indicating that the element with number $s$ is active (so $s$ is currently in AEL), and EFM[$s$] = 0 indicating that the element with number $s$ has not experienced change in any of its inputs and, hence, is inactive (so its evaluation is unnecessary).

Whenever EEU processes an event and determines the activated elements, it checks for each element $s$ and whether it has previously been entered into AEL in the current simulation cycle, by reading EFM[$s$], the "evaluation flag" of that element. If the flag is 0, EEU sets it to 1 and puts the number of that element in the AEL. If the flag is already 1, EEU does nothing since the element number is already in AEL, making the element marked for evaluation. This mechanism prevents multiple evaluations of an element with more than one altered input in a simulation cycle and, hence, accelerates simulation.

Whenever EEU reads an element number $s$ from AEL, it resets the corresponding evaluation flag, EFM[$s$], to 0, to prepare the EFM for the next evaluation cycle.

During the initialization process, before the start of the simulation, the Host loads the Final Time Register (FTR) register with the "final simulation time", as specified by the user.

Current Time Register (CTR) is an internal register of ESU, which holds, at any instant, the "current simulation time" of the system. It is read by the Host at the end of a simulation cycle and is then advanced by ESU to the next simulation time read from the timing wheel. It also leads to STI for being compared with the content of FTR.

Simulation Termination Indicator (STI) is a comparator that constantly compares the contents of FTR (final simulation time) and CTR (current simulation time). When the current simulation time reaches the final time specified, STI activates its output: The "Stop" signal which goes to the Host, EEU, EPU and ESU, informing them that simulation has terminated.

Timing Wheel Pointer (TWP) is a pointer inside the ESU that always points to the current slot on the timing wheel, corresponding to the current simulation time. At the end of a simulation cycle, TWP is increased to point to the next "slot" on the timing wheel and the slot previously pointed to by it is disposed of.

## ELEMENT EVALUATION UNIT

EEU performs the element evaluation; that is, determination of logic values of outputs of elements, based on the current logic values at their inputs, for all activated elements whose numbers are in AEL. A possible architecture of EEU includes a counter for addressing AEL; for reading the numbers of activated elements; a register for storing the number of the activated element read from the AEL; a bank of registers used to store the signal values of the inputs of the active element to be evaluated, based on these input values and the class of the element; a bank of registers used to store the previous signal values of the outputs of the evaluated element, read from EOT for comparison with the new values of the outputs obtained from the evaluation of the element; a bank of registers used to store the new signal values of the outputs of the evaluated element; a register for storing the delay of the element to be evaluated, (read from CDT); a register for storing the class number of the element to be evaluated (read from ECT), and an "event register", into which the newly constructed event is stored.

For every changed output, an event is created and sent to ESU, via an "event bus". To shorten the width of the event data structure, the difference between the event time and the current simulation time (the event delta time), which is equal to the delay of the element, is sent in the event, rather than the event time.

### EEU Algorithm

After receiving the "Start" signal from the Host, EEU repeatedly executes a loop until it receives the "Stop" signal from the STI. Communication between EEU and EPU involves a handshake with two lines; Evaluate and Acknowledge. The communication between EEU and ESU involves a handshake with three lines; NewEvent, No-Element and Accept.

In each iteration, EEU waits for the Evaluate signal from EPU. Having received 1 on this line, EEU sets the Acknowledge line to 1 and waits for Evaluate

to be lowered to 0 by EPU. It then lowers Acknowledge. EEU then reads the numbers of activated elements, one by one, from AEL and, for each number, performs the following:

1. The number of the activated element is used as the address to EIT, EOT, ECT and CDT, to read the input values, the previous output values, the class of the element and the delay of the element;
2. The 4-bit class number of the element is applied to the control inputs and the values of the element inputs are applied to the data inputs of the Arithmetic Logic Unit inside EEU and the element is evaluated;
3. The new and previous values of each output of the element are compared. For each altered output, EEU constructs an event and sends it to the ESU by placing the content of its event register on the "event bus" to ESU and activating the handshake line, NewEvent, to inform ESU that a new valid event is on the bus. EEU then waits until it receives 1 on the Accept line from the ESU, meaning that ESU has taken the event from the bus. EEU then lowers the NewEvent line back to zero;
4. EEU resets the evaluation flag of the element in EFM to 0, and loads "0000Hex" (indicating "no element") at the location in AEL, from which it had just read the element number.

The above operations are performed for each element number in AEL until "0000Hex" is encountered in AEL, indicating the end of the list of activated elements. EEU then activates the NoElement line, to inform ESU that there is no other activated element. It then waits until it sees 1 on Accept, meaning that ESU has received 1 on the NoElement line, at which point EEU lowers the line NoElement to 0 and then starts all over again by waiting for 1 on the Evaluate line.

## EVENT SCHEDULING UNIT

ESU performs event scheduling. A possible architecture for ESU includes a pointer to the first slot of the timing wheel; a register for holding the current simulation time; a pointer to the current time slot, which will be the next event dispatched by ESU to EPU; a register to store the event pointed to by the timing wheel, which will be the next event sent to EPU for processing; a register to store the newly constructed event received from EEU; a register to store the time of a new event, which is the sum of the content of CTR and the delta time specified in the event; and auxiliary pointers, used in the insertion of the newly received event in the appropriate place on the timing wheel.

### ESU Algorithm

After receiving the "Start" signal from the Host, ESU repeatedly executes a loop until it receives the "Stop" signal from the STI. Communication between ESU and EEU involves a handshake with three lines, Accept, NoElement and NewEvent. Communication between ESU and EPU involves a handshake with three lines, NextEvent, NoEvent and Ready. In each iteration, ESU does the following:

1. The first slot in the timing wheel, pointed to by TWP, points to the linked list of events pertaining to the current simulation time. The events on this list are transmitted, one by one, to EPU in the following manner. For each event, ESU waits until it receives 1 on the Ready line, sent by EPU to inform ESU that it is ready to get an event. Having received 1 on Ready, ESU places the currently scheduled event on the event bus to ESU and sets the NextEvent line to 1, thereby, informing EPU that a valid event is now on the bus. ESU waits until it receives 0 on Ready, meaning that EPU has taken and has read the event from the bus. ESU then lowers the NextEvent line;
2. Having sent all the events scheduled for the current simulation time, ESU then waits until it sees 1 on the Ready line and, then, sets the NoEvent to 1, to inform EPU that there is no other event to be sent. ESU then waits to see 0 on the Ready line, i.e., the acknowledgment of EPU that it has received the NoEvent signal. Having seen 0 on Ready, ESU lowers the NoEvent line;
3. ESU starts its transaction with EEU. It waits until it gets 1 on either the NewEvent line or the NoElement line. If it receives 1 on NewEvent, ESU reads the new event on the bus sent by EEU and then sets the Accept line to 1, to inform the EEU that it has received the newly transmitted event. ESU then waits for 0 on NewEvent, at which time it lowers the Accept line;
4. If the newly received event has zero delta time (in which case, it corresponds to the evaluation of an element with no delay), then, ESU sends it to the EPU for immediate processing, without scheduling it on the timing wheel. If the newly received event has non-zero delta time, then, ESU schedules it on the timing wheel at the proper slot;
5. Upon reception of 1 on the NoElement line, ESU raises the Accept line and waits to see 0 on NoElemen, at which time, it lowers the Accept line and adjusts the TWP to point to the next slot on the timing wheel;
6. ESU sends the "CycleComplete" signal to inform the Host that the current simulation cycle has

been completed. It then waits for the "Continue" signal from the Host, at which time, it lowers the "CycleComplete" line and loads the CTRd with the time of the slot on the timing wheel currently pointed to by the TWP (the new current simulation time) and starts the above steps all over again.

## EVENT PROCESSING UNIT

EPU performs processing of events generated by activated elements. A possible architecture for EPU includes: A counter for addressing AEL in the process of storing the numbers of the activated elements in AEL; a register to store the event sent by ESU and a register to store the element and input numbers of each node in the fanout list of the output when an event is being processed.

### EPU Algorithm

After receiving the "Start" signal from the Host, ESU repeatedly executes a loop until it receives the "Stop" signal from STI. Communication between EPU and ESU involves a handshake with three lines, NextEvent, NoEvent and Ready. Communication between EPU and EEU involves a handshake with two lines, Evaluate and Acknowledge. In each iteration, EPU does the following:

1. It repeatedly executes the following until it receives 1 on the NoEvent line from ESU: It sets the Ready line to 1 and waits until it receives 1 on either the NextEvent line or the NoEvent line. If it receives 1 on NoEvent, EPU exits from this inner loop, otherwise, with 1 on NextEvent, EPU reads from the bus the event sent by the ESU and, then, lowers the Ready line to acknowledge to ESU reception of the event. EPU then waits for the NextEvent line to be lowered by ESU. Having seen 0 on NextEvent, EPU starts the processing of the event it has just received. This involves updating the corresponding values in the EOT and EIT tables. For each of the elements on the fanout list of the changed output that has generated the event, EPU enters the number of that element in the AEL, if the evaluation flag of that element in the EFM is 0, and sets the evaluation flag of that element to 1. If the evaluation flag of that element is already 1, no action is performed on AEL and EFM. For this process, the fanout list of each output is extracted from the CTM;
2. Upon exit from the above inner loop, EPU lowers the Ready line and waits for 0 on the NoEvent line, at which time, it raises the Evaluate line, to inform the EEU that it can start evaluating elements;
3. EPU then waits to see 1 on the Acknowledge line, at which time, it lowers the Evaluate line and starts the above steps all over again.

## CONCLUSION

This paper aims at describing, at the conceptual and logic levels, a possible architecture of a hardwired simulator for performing the discrete event driven simulation algorithm. Physical implementation has been deliberately not discussed, in order to emphasize underlying architectural ideas and discrete algorithm paradigms.

The author believes the ideas outlined in this paper can be employed for efficient design of a large class of discrete algorithms and can be used in a wide variety of applications where discrete event driven simulation is useful for system testing and evaluation [6].

## ACKNOWLEDGMENT

The author wishes to thank the Research Council and the Department of Mathematical Sciences of Sharif University of Technology for academic and financial support.

## REFERENCES

1. Banks, J., Nelson, B.L. and Carson, J.S., *Discrete-Event System Simulation*, Third Edition, Prentice Hall (2000).
2. Severance, F.L., *System Modeling and Simulation: An Introduction*, John Wiley and Sons (2001).
3. Tyszer, J., *Object-Oriented Computer Simulation of Discrete-Event Systems*, Kluwer Academic Publishers (1999).
4. Fishman, G.S., *Discrete-Event Simulation: Modeling, Programming and Analysis*, Springer-Verlag (2001).
5. Train, K.E., *Discrete Choice Methods with Simulation*, Cambridge University Press (2003).
6. Walke, B., *Mobile Radio Networks: Networking and Protocols*, John Wiley and Sons (2002).