VLSI DESIGN
1996, Vol. 4, No. 1, pp. 59-68
Reprints available directly from the publisher
Photocopying permitted by license only

© 1996 OPA (Overseas Publishers Association) Amsterdam B.V.
Published in The Netherlands under license by
Gordon and Breach Science Publishers SA
Printed in Malaysia

Behavioral Simulation and Performance Evaluation of Multi-Processor Architectures

AUSIF MAHMOOD
Department of Electrical Engineering, Washington State University at Tri-Cities, 100 Sprout Road, Richland, WA 99352

(Received June 14, 1989, Revised January 9, 1990)

The development of multi-processor architectures requires extensive behavioral simulations to verify the correctness of design and to evaluate its performance. A high level language can provide maximum flexibility in this respect if the constructs for handling concurrent processes and a time mapping mechanism are added. This paper describes a novel technique for emulating hardware processes involved in a parallel architecture such that an object-oriented description of the design is maintained. The communication and synchronization between hardware processes is handled by splitting the processes into their equivalent subprograms at the entry points. The proper scheduling of these subprograms is coordinated by a timing wheel which provides a time mapping mechanism. Finally, a high level language pre-processor is proposed so that the timing wheel and the process emulation details can be made transparent to the user.

Key Words: *Simulation, Behavioral Simulation, Performance Evaluation, Process Simulation, Hardware Processes, Parallel Architecture Simulation*

1 INTRODUCTION

The design process for a VLSI multi-processor architecture normally starts by first partitioning the design into major building blocks. The behavior of each block is described in a language and an initial simulation is carried out to analyze the correctness of results. This may involve verification of the algorithm, throughput analysis, identification of potential bottlenecks etc. Note that this becomes a non-trivial task for multi-processor architectures employing distributed control such as data flow [15] and wavefront array processors [12].

For describing multi-processor architectures at the behavioral level, constructs for describing concurrency and multi-process functionality are required. The behavioral language should be able to describe the hierarchical and symmetrical relations between hardware units. The capability for description of hierarchical relationships corresponds to a block structure type of organization in the language. Symmetrical relations e.g., handshake and interfaces between processors can be modeled by process level constructs. These include constructs for scheduling a process, its termination, suspension and resumption. Multi-process functionality demands that there be constructs for concurrent scheduling of processes. Since different processes may take a different time to complete, a mechanism for synchronization of processes needs to be provided.

A time delay mechanism is required for modeling communication delays, arithmetic operation times etc. so that accurate execution times can be generated for benchmark programs, and a performance evaluation of the architecture can be made.

Concurrent programming languages e.g., concurrent Pascal [7], concurrent C [5], concurrent Prolog [20, 24], Ada [21, 26], and Modula-2 [27] etc. meet most of the requirements for a behavioral simulation language except for a time delay mechanism. Concurrent Prolog has been used to describe VLSI circuits at the functional level [23]. The main drawback in using a concurrent language is the difficulty of associating time delays with hardware processes in a straight-forward manner (e.g., see [23] for problems related to implementation of a global clock).

General simulation languages such as SIMULA [3], GPSS [14], SIMSCRIPT [11], incorporate the necessary constructs required for a behavioral simulation including a simulated time scale for modeling delays. Some special

hardware description languages e.g., SLIDE [16, 17] have also been designed to describe interfaces between processors. The drawbacks in using above languages include higher cost, less availability, portability problems, and difficulty of learning a new language.

The hardware description language VHDL [9, 19] (which is based on Ada high level language) is a hierarchical language that supports design descriptions from behavioral to structural levels. This flexibility in VHDL comes at the price of enormous size (about five times the size of standard Pascal) which makes its simulators expensive and the language difficult to learn. Since most parallel architecture designers or parallel algorithm designers do not need any structural level capabilities in the simulation language, VHDL is overly complex and an expensive choice in this respect.

Popular programming languages such as Pascal and "C" have been used in the behavioral simulation of computer architectures. Although these provide maximum flexibility in the simulation of a design and the execution of benchmark programs, they lack two important constructs needed for the simulation of parallel architectures i.e., process level constructs and a time mapping mechanism. However, it will be shown in the next section that these can be easily incorporated. The techniques will be presented to emulate the concurrent processes involved in a multi-processor architecture such that an object-oriented description of the design is maintained. Hence, an object-oriented high level language such as "C++" [22] is preferred but the techniques presented in this paper are general and can be used with any high level programming language.

2 PROCESSES IN A HIGH LEVEL LANGUAGE

The theory of processes has received a great deal of attention in the last decade due to increased interest in parallel programming. Much of this work has been devoted to the important issues of process communication and synchronization. There are two basic methods for these [1]; one uses explicit signaling, and the other is an indirect method which updates shared variables. Implementations of these methods have resulted in mechanisms such as monitors [7, 8], interface modules [27], event counts and sequencers [18], semaphores [4], conditional critical regions [6], messages [6], entries [21, 26], and managers [10]. Each of these mechanisms assumes a different philosophy about the relationships between processes. For describing hardware, there is no single mechanism that is completely suitable. It is important that the design description should closely reflect the underlying hardware design, and so such a mechanism should be adopted. For communication between processes, either explicit signaling or updating of shared variables can be used depending upon the design of the hardware. As an example, handshake is better described by explicit signaling whereas a bus arbitrator might use a status flag to indicate whether the bus is in use. The process synchronization mechanism used in our implementation is known as a timing wheel [13, 25]. It may function as a monitor or an event sequencer depending upon how it is used in the simulation of a design.

2.1 Modeling Hardware Processes

For behavioral simulation of multi-processor architectures, interfaces between processors and distributed control in the design can be modeled by hardware processes. A hardware process is a sequence of actions which is controlled independently of other sequence of actions [17]. For example in a multi-processor system, the process of computation in one processor is carried out concurrently with the process of computation in another processor. At a given time in an architecture, several processes may be active corresponding to computation, transmission and receiving of data etc. Since each process may take a different amount of time to complete, a time mapping mechanism is needed to keep track of the completion of a process and the start of another. Also, by using a time mapping mechanism, execution time of a benchmark program can be measured accurately for performance evaluation purposes.

In a high level language such as Pascal or "C", there are no predefined constructs for handling concurrent processes. However, these can be incorporated by representing a process by a subprogram (procedure or a function), and their concurrent scheduling can be emulated by a time mapping mechanism. The statements inside the subprogram correspond to the sequence of actions implied by the process. A process is activated when the corresponding subprogram is invoked. The termination of a process is equivalent to reaching the end of subprogram i.e., execution of a return or end statement.

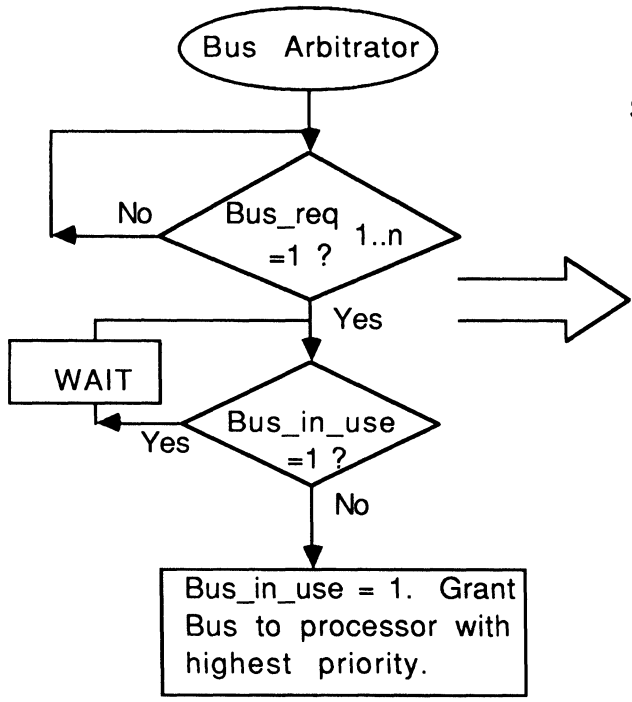
Suspension of a process may occur at the start of a process or some time later. This may happen if the resources needed for the process execution are tied up, or if a higher priority process is given control of the resources currently being used by the original process. If a process needs to be suspended immediately after its invocation, the effect of its suspension can be simulated by a subprogram which checks for its enabling condition at the beginning. If this condition is not satisfied—meaning that the process needs to be suspended, the subprogram's execution is restarted later by using the time mapping mechanism.

The effect of suspension of a process that might be suspended in the middle requires that it be resumed at exactly the same point where it was suspended. This effect can be simulated by representing the process by multiple subprograms. Each suspension point in the process corresponds to the end of a previous subprogram and the start of a subsequent subprogram. The new subprogram will check the suspension condition at its beginning and may be suspended until this condition is satisfied. In this way, the suspension of a process may result in delayed execution of the remaining subprograms representing it.

There are two possible implementations for resuming a suspended program. One approach is to have the suspended subprogram check for its resumption by reinserting a call to itself in the next time unit. The subprogram will check for its suspension condition in every time unit and when it becomes false, it will continue its execution. This method is particularly useful when many processes are suspended because of the lack of availability of a resource. Figure 1 shows an example of an arbitration controller for a bus. If a request from a processor for the bus is issued when the bus is already being used by another processor, it cannot be granted until it is released by the processor using it. A natural way of representing the waiting of the arbitration controller until bus is released will be to have it continuously check for its availability by reinserting a call to itself in every time unit.

The second approach for resuming a suspended process is to have the process releasing the resource enable the suspended process(es) on this resource. This approach is efficient and more effectively models the hardware interaction in cases such as handshaking protocols. Figure 2 depicts an example of a process of preparation of data and a responsive handshake before data is to be transmitted. The process will be suspended as long as an acknowledge is not received from the data receiving medium. This process can be modeled by two subprograms; one corresponding to the preparation of data and the other for transmitting data after an acknowledge is received. The suspension of the process until an acknowledge is received will then be equivalent to scheduling the second subprogram upon receiving this signal i.e., the second subprogram will be invoked by the data receiving medium when it is ready to accept data. Explicit signaling is used in the invocation of subprograms in this example as it reflects the design intent more closely.

The mechanism used here for suspension of a process achieves the synchronization of processes i.e., it is similar in concept to rendezvous in Ada or concurrent C. In Ada, processes contain synchronization and communication points called entries which may be called by other processes (in concurrent C, these points are called transactions). In describing hardware, it is more meaningful to call these suspension points as they normally correspond to temporal waiting of a process. The access




```
Subprogram Bus_Arbitrator
  begin
    if Bus_in_use
      then              --wait
          Insert a call to Bus_Arbitrator
          in next time slot
       else             --grant bus
          Bus_in_use = 1
          Grant Bus to processor
          with highest priority
          Insert a call to this processor
          in next time slot
  end
```

FIGURE 1 Representation of Bus_Arbitration Process by its Equivalent Subprogram.

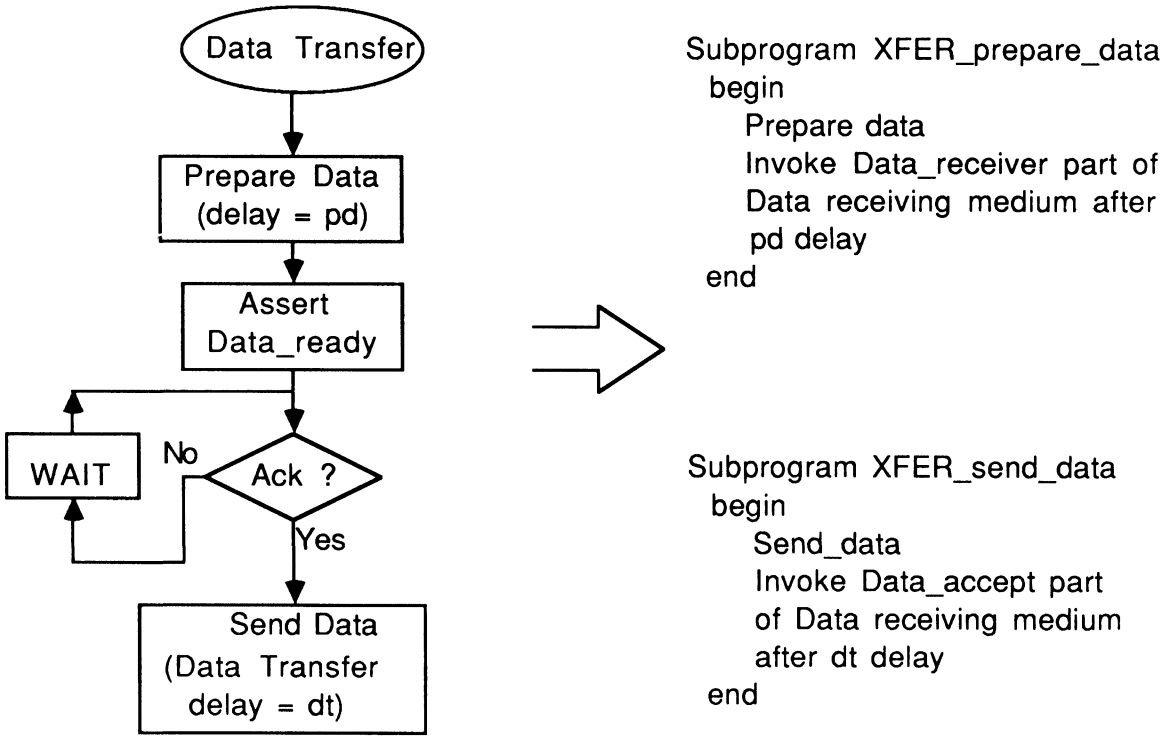


FIGURE 2 Representation of a Data Transfer Process (handshake) by two Subprograms.

to these suspension points by other processes is achieved by representing the hardware processes with more than one subprogram. Each subprogram corresponds to a synchronization or suspension point. The temporal waiting is achieved by the time mapping mechanism. Thus by using multiple subprograms for a hardware process and the time mapping mechanism, suspension of processes can be easily implemented.

For scheduling processes at different times, a time mapping mechanism is required. A multi-stack implementation known as timing wheel is preferred to a single event queue as it allows efficient scheduling of concurrent processes. The basic simulation algorithm involving hardware processes is presented in the following subsection. The algorithm is similar to an event driven logic simulation algorithm [2, 13, 25] except that it has been modified to incorporate processes.

```
While (time <> end_simulation) and (not termination
_condition)
  {
    { For each slot in the timing wheel
      {
        For each partition in a slot --update phase
          {
            .Update global flags according to the code
            specified in the partition.
          }
        For each partition in a slot --schedule phase
          {
            .Schedule the process (subprogram) in the
            partition (the execution of this process may
            further result in a process being inserted in a
            future time slot)
            .Move to next partition
          }
        .Move to next slot in the circular timing wheel
        .Increment time
      }
  }
```

The timing wheel is divided into time slots. Each time slot contains partitions which correspond to parallel events in this time. The simulation algorithm is divided into two phases. The first phase scans the current time slot to update any global flags. The second phase schedules the subprograms that are called in the time slot. It is important to update the flags first, because otherwise a subprogram may be executed before a relevant condition is updated causing incorrect scheduling of activities.

The simulation process starts when some subprograms are initially inserted in a timing wheel slot. The main program in the high level language implements the simulation algorithm and only scans the timing wheel. It schedules any subprograms contained in the current time slot. The execution of a subprogram may result in insertion of another subprogram in a future time slot.

2.2 Object-Oriented Description of Hardware Design

The description of a hardware design may be divided into two parts i.e., datapaths and controllers. In the initial design phase, different controllers and datapath functions in the design are identified. Delay time associated with each datapath function may be approximated and incorporated into the states of the controller. Uninterruptable states are integrated into a single state with a delay value that is the sum of the delays of the individual states. A controller may then be represented by multiple subprograms depending upon the number of suspension points in it according to the techniques described in the previous section. All identical controllers operating on different datapaths need to be represented by *only one set* of subprograms. An identifying parameter may be used in the subprogram's description to distinguish between invocations of different controllers of the same kind. For example, in the representation of a two-dimensional systolic array, a position co-ordinate (x,y) may be assigned to each processor and passed as parameters to the subprogram(s) representing a processor. A timing wheel emulates concurrent activation of multiple controllers by inserting the corresponding number of calls to these controllers in a time slot. This scheme meets similar goals as a "class" and "object" concept in object-oriented programming. The set of subprograms representing a controller which operate upon the data structure (datapath) form a "class" as shown in figure 3. Activations of these subprograms then correspond to the instances of a class i.e., objects. Hence, by using one set of subprograms for identical controllers, the description of large processor arrays will be extremely concise and easily extendable. If an object-oriented language is not used, then all memory devices, registers, status flags etc. in every datapath are represented by global variables in the high level language. The global variables may be indexed (dimensioned) according to the corresponding position coordinates of the processor as mentioned earlier. This way each instance of a controller can refer to its own datapath by matching the parameters to the datapath index.

3 AN EXAMPLE OF BEHAVIORAL SIMULATION

A simulation of a data flow machine is presented in this section. A data flow machine is a multiple processor architecture involving distributed control. For purposes of simplicity in demonstrating the concepts of behavioral simulation, a small static data flow machine [15] with 16 memory cells and 4 processors is shown in Figure 4. A separate control network is not included in the architecture, and both operation results and acknowledge signals are routed through the distribution network. The design of the architecture uses a queue of enabled memory cells and a queue of enabled processors. The arbitration controller sets up the interconnection between the memory cell at the front of the cell queue and a processor

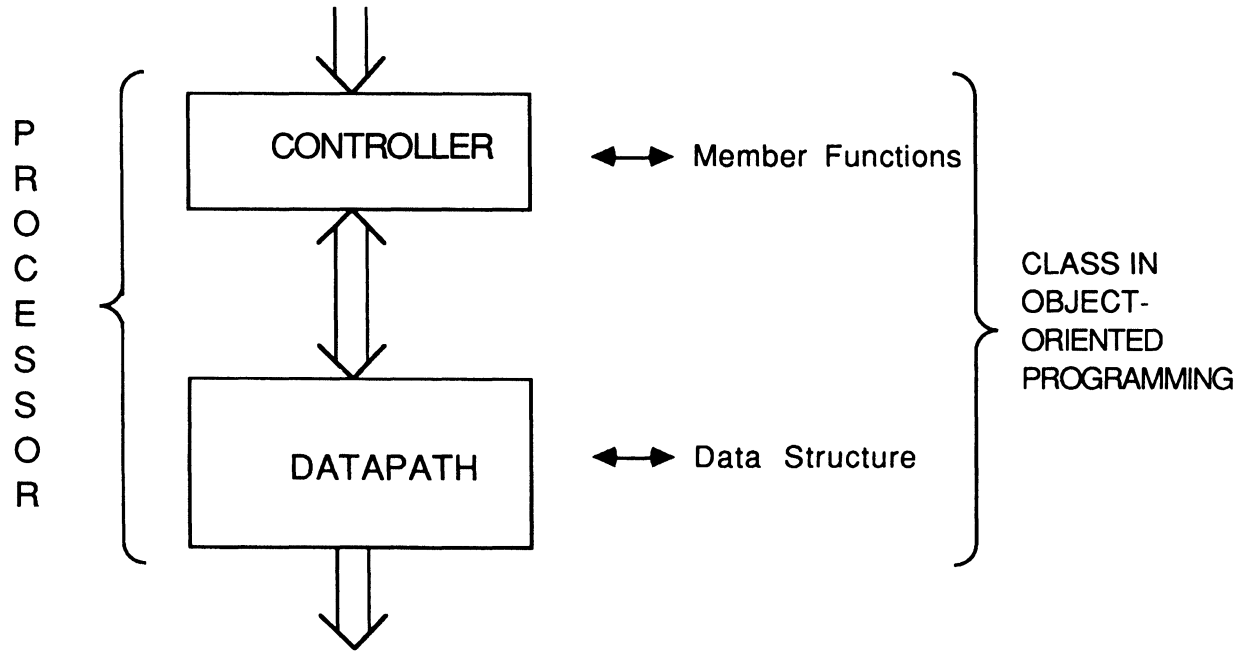


FIGURE 3 Object_Oriented View of Hardware Description.

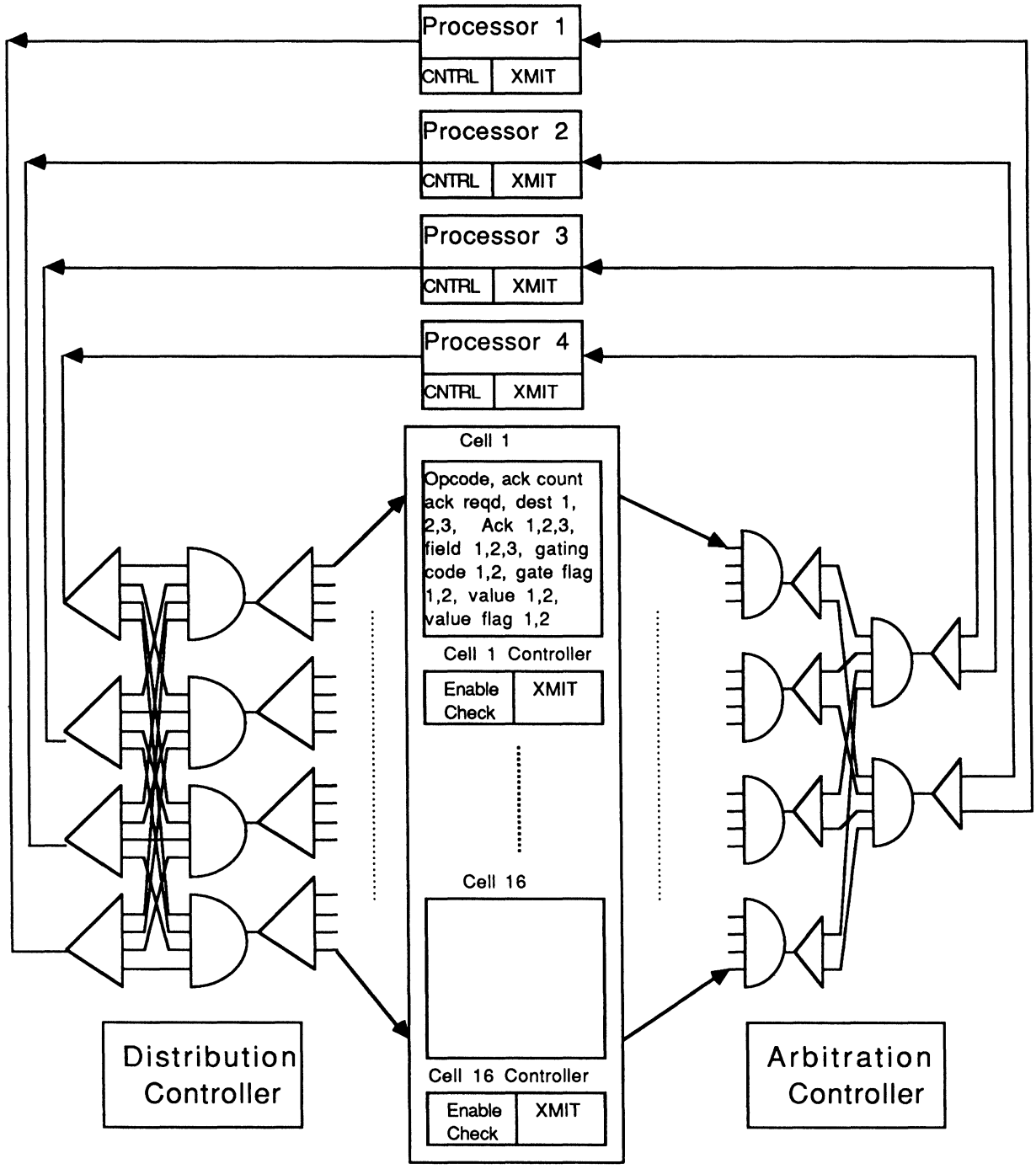


FIGURE 4 A Static Data Flow Machine with 16 Cells and 4 Processors.

at the front of the processor queue. Three global flags are used to indicate availability of processors, arbitration, and distribution networks.

In a static data flow machine, an instruction contained in a memory cell is enabled when it has received both the operands and the required number of acknowledge signals. A controller is associated with each cell that checks for the enabling condition and transmits data to a processor after the instruction is enabled. For the architecture shown in Figure 4, there are 16 controllers for the memory cells, one for the arbitration network, four for the processors, and one for the distribution network. The simulation of this architecture can be viewed as interactions of these controllers (processes).

The simulation process may start when a cell controller performs a check for an enable condition because

the cell has received an input or an acknowledge signal. If the cell is enabled, it will cause activation of the arbitration controller to set up a path from the cell to an available processor. After the path is set up and the data are sent to a processor, the processor computes the results and will invoke the distribution controller. The distribution controller sets up the interconnection between the processor and a memory cell. The data and acknowledge signals are transmitted from the processor to memory cells and the process continues as this in turn will activate the cell controllers.

Processes corresponding to memory controllers and processors can be suspended if the arbitration and distribution networks are busy. For this reason, the memory cell controller is split into two subprograms. The first subprogram checks for the enable condition and if it is enabled, it places a call to arbitration controller in a future time slot depending upon the time consumed in this checking. The arbitration controller will then invoke the second subprogram of the cell controller after the arbitration network and a processor are available. After computing the results, the processor may have to wait for the availability of distribution network. For this reason the processor is also represented by two subprograms. Figure 5 shows the control actions needed for a cell controller and its representation by subprograms.

There are a total of six subprograms corresponding to the different controllers in the simulation of architecture of figure 4. These include two subprograms for the memory cell controllers, two for the processors, one for arbitration controller, and one for distribution controller. The timing wheel accomplishes concurrent activation of different memory cell controllers and processors by inserting the corresponding number of calls into a time slot and passing a cell number or a processor number as a parameter. Hence extensibility in the hardware can be

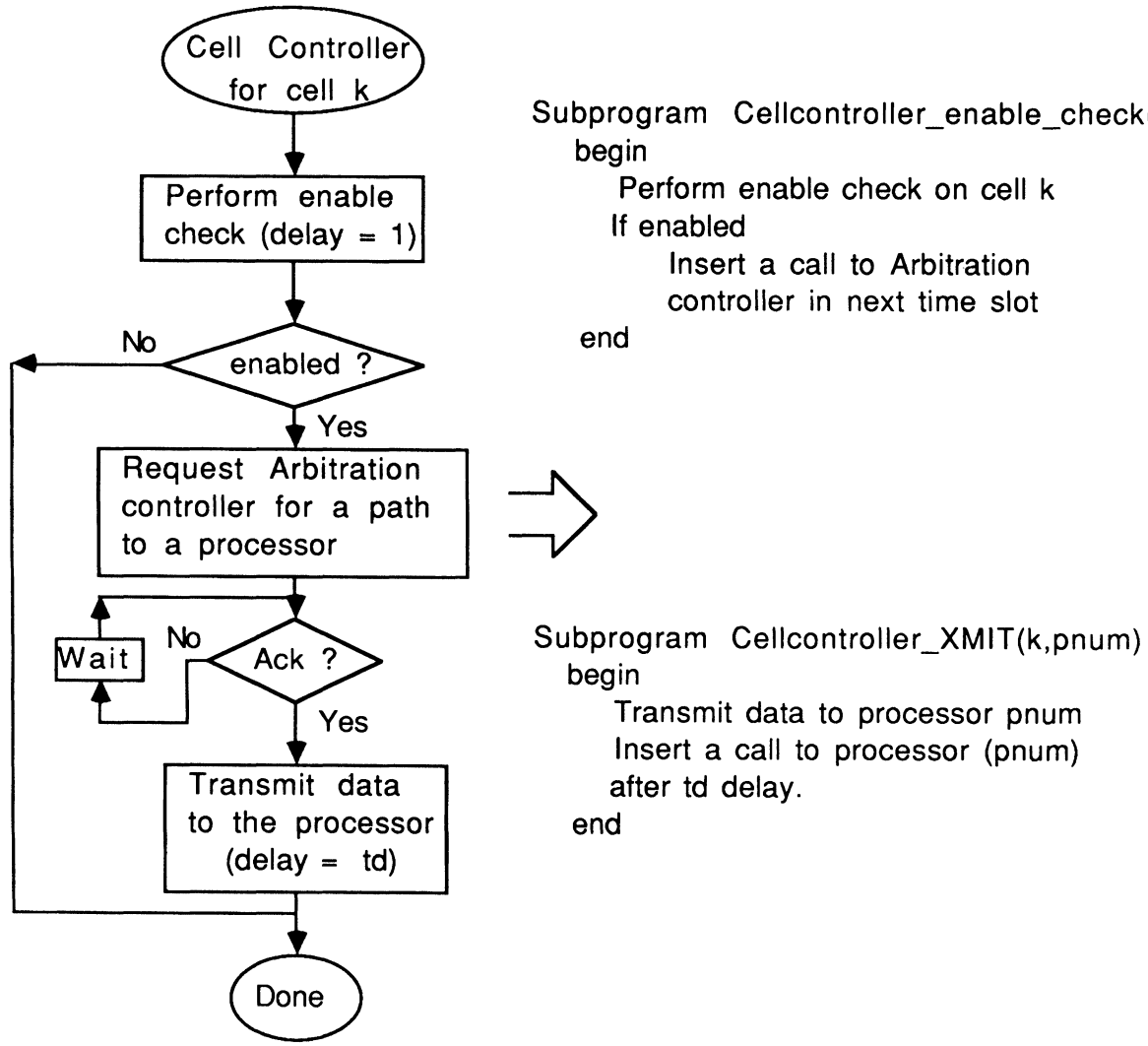


FIGURE 5 Representation of Cell Controller Process by its Equivalent Subprograms.

TABLE 1
Interactions Between Processes During Simulation of a Static Data flow Machine

TIME	TIMING WHEEL CONTENTS	SUBPROGRAM ACTION
i	—	A cell k gets a either data, acknowledge, or a control signal (gate flag). This results in a call to CELL_CONTROLLER_ENABLE_CHECK(k) in the next slot.
i+1	CELL_CONTROLLER_ENABLE_CHECK(k):	Check enable condition for cell k by examining gating code, gate flag and value flag for each input. Also ack_count is compared to ack_required. If enabled, then place cell k in CELL_QUEUE and insert a call to ARBIT_CONTROLLER in the timing wheel in next time slot i.e., it is assumed that the checking for enable condition took one time unit.
i+2	ARBIT_CONTROLLER :	Check availability of arbitrator and a processor by examining ARBIT_AVAILABLE_FLAG and PROCESSOR_AVAILABLE_FLAG. If both are not available then wait i.e., insert a call to itself (ARBIT_CONTROLLER) in next time slot (may be they will be available then).
i+3	ARBIT_CONTROLLER : (assuming a processor and ARBITER were busy)	If a processor and arbitrator are still busy, then wait i.e., call itself in next time slot.
i+-	ARBIT_CONTROLLER : wait	
i+8	ARBIT_CONTROLLER :	Check availability of arbitrator and a processor. If both are available, then get cell number from front of CELL_QUEUE and processor number from front of PROCESSOR_QUEUE. Set ARBIT_AVAILABLE_FLAG = 0. Decrement PROCESSOR_AVAILABLE_COUNT and if this count is zero, set PROCESSOR_AVAILABLE_FLAG = 0. Set up interconnections from cell number to processor number in the arbitration network. Place a call to CELL_CONTROLLER_XMIT (cell number, processor number) in next time slot so that this cell can start transmitting data to the processor.
i+9	CELL_CONTROLLER_XMIT (cell number, processor number) :	Transmit data values, opcodes, destinations for results, acknowledges to the processor. Indicate number of destinations as DCOUNT, acknowledges as ACOUNT so that processor knows how many times it has to request the DISTRIBUTOR_CONTROLLER to send data to cells. Absorb gate flags, value flags (if gating code is not C) and reset ack_count. Call PROCESSOR 4 time units from now (assuming ARBITRATOR delay = 4). Also release ARBIT_AVAILABLE_FLAG 4 time units from now by putting a release code in the timing wheel.
i+13	RELEASE ARBITRATOR :	ARBIT_AVAILABLE_FLAG = 1.
i+13	PROCESSOR :	Evaluate the opcode. Prepare ACOUNT number of (acknowledge, destination) pairs. Prepare DCOUNT number of (result, destination) pairs. Place (ACOUNT+DCOUNT) number of calls to DIST_CONTROLLER in timing wheel after opcode delay from now. processor_xmit_count = ACOUNT + DCOUNT.
+23	DIST _CONTROLLER :	If DIST_AVAILABLE_FLAG = 0, then wait i.e. insert itself in next time slot.
i+--	DIST_CONTROLLER : wait	
i+26	DIST_CONTROLLER :	If DIST_AVAILABLE_FLAG = 1, then set up connections in the distributor from the processor to the cell. Set DIST_AVAILABLE_FLAG = 0. Call PROCESSOR_XMIT in next time slot so that processor can transmit data.
i+27	PROCESSOR_XMIT :	Send results to value fields of cell. Set value flag of cell to 1. Send control signals to gate flag field of cell. Increment ack_count of cell if an acknowledge was sent. Insert call to CELL_CONTROLLER_ENABLE_CHECK 3 time units from now. Decrement processor_xmit_count (processor_xmit_count was set to (ACOUNT+DCOUNT)). If processor_xmit_count is zero, then increment PROCESSOR_AVAILABLE_COUNT and put this processor in PROCESSOR_QUEUE indicating that it is available 3 time units from now. DIST_AVAILABLE_FLAG to 1 in 3 time units from now. (delay of distributor is assumed 3 time units).
i+30	RELEASE DISTRIBUTOR :	DIST_AVAILABLE_FLAG = 1.
i+30	RELEASE PROCESSOR :	PROCESSOR_AVAILABLE_COUNT is incremented and processor is placed in PROCESSOR_QUEUE.
i+30	CELL_CONTROLLER_ENABLE_CHECK :	

easily handled e.g., the number of subprograms in the simulation remains the same if the number of memory cells or processors are either increased or decreased. The registers in each memory cell are declared as a global array of structures (records). The index of the array corresponds to the cell number.

The description of each subprogram and the contents of the timing wheel during a simulation cycle are shown in Table 1. The main program scans the timing wheel and implements the simulation algorithm of section 2.1. One additional subprogram is required to do the proper insertion of a call and parameters to a subprogram in the timing wheel.

4 CONCLUSIONS

The behavioral simulation of multi-processor architectures involves process level constructs. Techniques were presented to emulate processes by representing suspension points in a hardware process with multiple subprograms and using a timing wheel mechanism. The advantage of a timing wheel formulation is that it allows measurement of execution times and also is helpful in implementing the "class" concept for processes. This way the total number of subprograms required in the simulation of an architecture depends upon the different types (classes) of processors and is independent of the total number of processors in a design. Hence the design description is extremely efficient and easily extendable to different number of hardware resources.

The criticism of the approach presented in this paper is that it requires a user to understand the splitting of a process into its equivalent subprograms. The techniques presented are straight-forward and are based on the concept of processes in a concurrent language i.e., suspension points in a hardware process are equivalent to entry points in Ada, or transaction points in concurrent C. Still, using multiple subprograms to represent a process affects the readability of the description in relation to its hardware design. However, by developing a small set of special keywords and constructs for describing the delay and suspension of a process, the description of hardware designs can be made elegant. This will require a language pre-processor to convert the processes in the user program to their equivalent set of subprograms. This way the timing wheel and splitting of subprograms can be made transparent to the user. The small set of constructs needed in the high level language pre-processor is listed below.

```
<PROCESS> ::= CONTROLLER "name"
              (<parameter list>) ;
              BEGIN <process statements> END
  <process statements> ::=
                {<HLL statement>} |
                {<WAIT statement>} |
                {<SIGNAL statement>} |
                {<CONTROLLER STATE>}
  <HLL statement> ::= "High Level Language
                      Statement"
  <WAIT statement> ::= WAIT (<WAIT TYPE>) ;
    <WAIT TYPE> ::= "n time units" | status_flag |
                    process_name. variable-
                    _name
  <SIGNAL statement> ::= SIGNAL ( <SIGNAL
                         TYPE> ) ; |
    SIGNAL ( <SIGNAL TYPE> ) AFTER "n time
    units" ;
  <SIGNAL TYPE> ::= process_name |
                    process_name. variable_name
  <CONTROLLER STATE> ::= BEGIN_STATE
                         <process statements>
                         END_STATE WITH
                         DELAY "n time
                         units"
```

As an example, using the above constructs the controller of figure 1 is easily described as,

```
Controller Bus_Arbitrator () ;
    begin
      WAIT (Bus_in_use) ;
      determine_priority (pnum) ;
      Bus_in_use = 1 ;
      SIGNAL (processor pnum) AFTER 1 ;
    end
```

A pre-processor for C++ is currently being developed.

By using a popular high level language for behavioral description and simulation, the user does not have to learn a new language. Since multi-processor architectures are mostly designed to run programs in a high level language, the use of a high level language to simulate the architecture and the benchmark's execution provides maximum flexibility for design verification and performance evaluation.

References

[1] Andrews, G. R., "Synchronizing Resources," ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, October 1981, pp. 405–430.

[2] Breuer, M. A. (editor), "Digital System Design Automation: Languages, Simulation and Data Base," Computer Science Press, Inc., 1975.

[3] Dahl, O., and K. Nygaard, "SIMULA- An Algol Based Simulation Language," Communications of the ACM, vol. 9, no. 9, Sept. 1966, pp. 671–678.

[4] Dijkstra, E. W., "Cooperating Sequential Process," Programming Languages, F. Genuys (editor), Academic Press, New York, 1968, pp. 43–112.

[5] Gehani, N. H., and W. D. Roome, "Concurrent C," Software Practice and Experience, vol. 6, no. 9, Sept. 1986., pp. 821–844.

[6] Hansen, B. P., "Operating System Principles," Prentice Hall, New Jersey, 1973.

[7] Hansen, B. P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, vol. SE-1, no. 2, June 1975, pp. 199–207.

[8] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," Communications of the ACM, vol. 17, no. 10, October 1974, pp. 549–557.

[9] "IEEE Standard VHDL Language Reference Manual," IEEE inc, New York, NY, April, 1989.

[10] Jammel, A. J., and H. G. Stiegler, "Managers versus Monitors," Information Processing 77, B. Gilchrist (editor), Elsevier North-Holland, New York, 1977, pp. 827–830.

[11] Kiviat, P., R. Villanueva, and H. Markowitz, "The SIMSCRIPT II Programming Language," Prentice Hall, 1969.

[12] Kung, S. Y., S. C. Lo, S. N. Jean, J. N. Hwang, "Wavefront Array Processors—Concept to Implementation," Computer vol. 20, no. 7, July 1987.

[13] Mahmood, A., "An Extensible Multi-Level Logic Simulator with Model Abstraction Capabilities," Progress in Computer Aided VLSI Design, ed. G. Zobrist, vol. 1, Ablex, 1989.

[14] Maisel, H., and G. Gnugnoli, "Simulation of discrete Stochastic Systems," Science Research Associates, 1972.

[15] Myers, G. J., "Advances in Computer Architecture," John Wiley and sons, 1982, pp. 463–494.

[16] Parker, A. C., and A. H. Altman, "The SLIDE simulator: A Facility for the Design and Analysis of Computer Interconnections," Proceedings of the 17th Design Automation Conference, 1980.

[17] Parker, A. C., and J. J. Wallace, "SLIDE: An I/O Hardware Description Language," IEEE Transactions on Computers, vol. c-30, no. 6, June 1981, pp. 423–439.

[18] Reed, D. P., and R. K. Kanodia, "Synchronization with Eventcounts and Sequencers," Communications of the ACM, vol. 22, no. 2, February 1979, pp. 115–123.

[19] Shahdad, M., et. al., "VHSIC Hardware Description Language," Computer, vol. 18, no. 2, February, 1985. pp. 94–103.

[20] Shapiro, E. Y., "A Subset of Concurrent Prolog and its Interpreter," ICOT Technical Report, TR003, Tokyo, February, 1983.

[21] Shumate, K., "Understanding Concurrency in Ada," McGraw Hill, New York, 1988.

[22] Stroustrup, Bjarne, "The C++ Programming Language," Addison-Wesley, 1987.

[23] Suzuki, N., "Concurrent Prolog as an Efficient VLSI Design Language," Computer, vol. 18, no. 2, February, 1985. pp. 33–41.

[24] Suzuki, N., "Experience with Specification and Verification of Complex Computer Using Concurrent Prolog," Logic Programming and its Applications, Von Cavegham & Warren, Editors, Ablex Publishing Co., 1985.

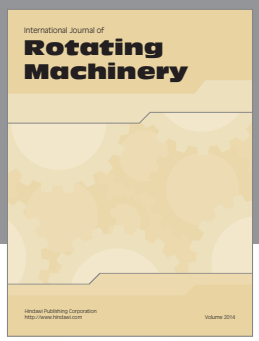
[25] Ulrich, E. G., "Exclusive Simulation of Activity in Digital Networks," Communications of the ACM, vol. 12, no. 2, Feb. 1969, pp. 102–110.

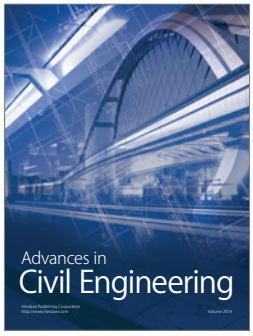
[26] U.S. Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD 1815A, Jan. 1983.

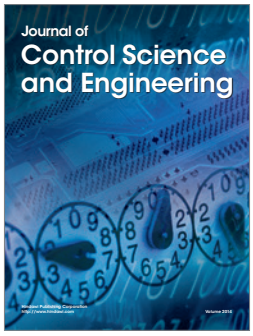
[27] Wirth, N., "Programming in Modula-2," Springer-Verlag, New York, 1982.

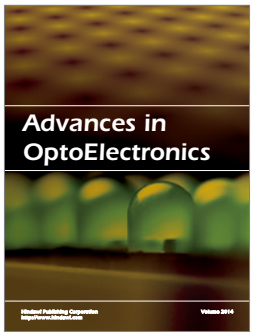
Biography

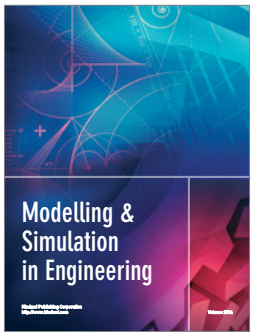
AUSIF MAHMOOD is currently an assistant professor in the school of Electrical Engineering and Computer Science at Washington State University at Tri-Cities. He holds a B. S. in electrical engineering from university of engineering and technology Lahore, Pakistan and M.S. and Ph.D. degrees in electrical engineering from Washington State University. His research interests are in CAD for VLSI, computer architecture and parallel processing.

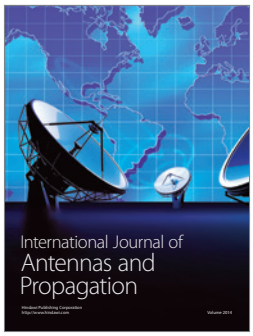


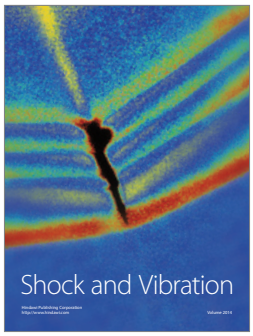


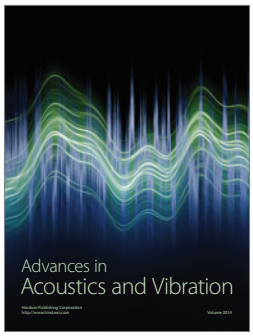












Hindawi

Submit your manuscripts at
http://www.hindawi.com