8/31/89

LOGIC LEVEL AND FAULT SIMULATION ON THE RP3 PARALLEL PROCESSOR

Stephen H. Unger
Computer Science Department
Columbia University

CUCS 479-89

1. Introduction

Logic level simulation for circuits of the sizes currently being designed is indeed a formidable computational task. Chips are being built today containing over a million gates, with storage elements and RAMs. Ordinary logic simulation of systems of this size can take many hours of computation time on the fastest computers. Fault simulation for such large chips, is out of the question for anything but the largest supercomputer. Certainly this is a task justifying the use of parallel processing.

The 64 processor RP3 can serve as a useful test bed for experimenting with parallel processing techniques applied to the problem of logic level simulation and related DA algorithms, such as fault generation and simulation. I have been doing just this since last summer. I have developed a number of algorithms for general logic simulation and for fault simulation. These have been implemented with programs running on the RT under Mach. This permits testing the workability of the programs, since it simulates a parallel processor environment. But it gives no direct information as to how fast the programs would run on an actual parallel processor. I have run programs on the RP3, but my examples are not large enough to yield much meaningful data on speed.

In the following sections, I will describe the simulation algorithms that I have developed, and indicate what I propose to do in the future.

2. Combinational Logic Simulation- Overview

My goal is to develop a scheme that allows the work to be distributed among the processors in such a manner that:

(1) Each can do a substantial amount of work without having to wait for information from the other processors.

(2) The processors can share fixed information, principally the descriptions of the circuits being simulated.

The first requirement is necesssary if the processors are to be usefully employed, i.e. if the useful work that each does is to be substantially greater than the overhead necessary to distribute and coordinate tasks.

The second requirement is important in keeping the memory requirements within reasonable bounds. If large circuits are to be simulated, it would be undesirable to require that each processor have a copy of the circuit description.

The approach that I have taken is to simulate on a level basis. That is, gates fed only by primary inputs (or latches) are on level 1, and a gate is on level i if the highest level gate feeding it is on level i-1. Given all of the primary input signal values, it is then possible to evaluate the outputs of all of the level 1 gates, independently of one another. Once all of the level i gates have been evaluated, it is then similarly possible to evaluate all of the gates on level i+1 independently of one another. Using a parallel processing system, we can attack the levels in sequence. At each level, the gates are partitioned equally among the processors. When every processor has completed the evaluation of the gates assigned to it, the gates on the next level are evaluated.

A single description of the circuit exists, and is accessible to all of the processors. The description consists of a list of gates (with their types) and, for each gate, pointers to the gates that feed it and to the gates it fans out to. Similarly, the current signal values at each gate output (as well as the primary input signal values) are also globally available. There is never an occasion for two processors to attempt to change the same signal value, nor is there any possibility of a processor changing a signal value that another processor must read.

The efficiency of this scheme may be improved (by how much is not clear) by evaluating only those gates for which at least one input has changed since the last time that gate was evaluated. (It is assumed

that we are concerned with a sequence of primary input states.) If the input sequence can be so ordered that only a small fraction of the input signals change at each step, then the overhead involved in keeping track of which gates have active inputs would be small compared to the amount of evaluation work saved. Evaluating this possible enhancement is one of the objectives of the proposed research.

3. Sequential Circuit Simulation- Overview

It is useful to be able to simulate systems with memory, i.e. systems that have RAMs, and/or storage elements such as latches, where the outputs of these devices constitute some of the input variables. (It is assumed in the following discussion that the systems simulated are clocked or synchronous.) In order to extend the process outlined in the previous section to cover systems with feedback, it is necessary to introduce a phase of the simulation in which some of the output signals are transmitted to the feedback inputs. It is also necessary to arrange for the storage of state values of storage elements. If RAMS are to be allowed as circuit elements, then additional special storage is needed for the contents of memory.

In order to permit the accurate simulation of systems with multi-phase clocking, it is also necessary to incorporate a convenient way of specifiying the sequence of clock signals, and to allow this sequence to be altered during the simulation. This can be done by allowing suitably coded clocking specifications to be incorporated in the stream of input signals. Such clocking specifications are injected only when the clocking sequence is to be changed. My simulation programs include such features. A special application is to allow the simulation of systems using LSSD. In particular, one of the primitive elements I have implemented is a scan latch.

It is assumed that clocking intervals are sufficiently long (or long-path delays are sufficiently small) that the outputs of the circuits reach stable conditions before the arrival of the next clock pulse. It is also assumed that short-path delays do not cause any problems. The general approach that I am using can be extended to give information on signal path delays, and indeed some of my earlier programs implemented this, but I am postponing further work along this line until later.

The simulator allows for unknown logic values, the high impedance state (tri-state logic), and a variety of logic elements as primitives (including MUX's, RAMS, and LSSD scan latches.)

4. Fault Simulation- Overview

The basic fault simulation problem is to determine which members of a given list of faults are detected by a given sequence of test vectors. My basic fault simulation program works as follows: The first input (test vector) is applied, and the outputs of all gates in the valid circuit are determined and recorded. Each member of the fault list is then injected into the circuit, one at a time. For each fault, the simulator, starting at the site of the fault, determines which gate values are affected. If any of them are among the set of signals specified as "observable outputs", the fault is checked off as detected, and is deleted from the list. After the entire list has been processed in this manner, the next test vector is applied, and the process is repeated. This continues until either the test sequence ends or the list of undetected faults is empty. Note that, during the simulation of the faulted circuits, only those gates with at least one input changed as consequence of the fault are evaluated. In practice, this means that only a small fraction of the gates (for a circuit of any size) need be evaluated.

This scheme has been implemented both for a uni-processor and for an RP3 type multi-processor. In the multiprocessor version, the fault list is partitioned into n equal lists, one for each processor. After the valid circuit has been evaluated for a test vector (using the parallel processing technique outlined in Sections 2 and 3 above), each processor, in parallel, then processes the circuit for each of the faults on its list (one at a time). Faults not detected are placed on one of the n members (in rotation) of a second set of fault lists, to be processed for the next test vector. Each processor works entirely independently of the other processors on the faults it has been assigned. Processors share the circuit description, but maintain private copies of the signal values for the faulted circuits, so that they do not interfere with one another other than to read common information. (Another, less frequent, form of contact may occur during the placing of as yet undetected faults on fault lists.)

The most complex aspect of the program has to do with the simulation

of sequential circuits, particularly those including RAMs. Various lists of previous values of stored signals must be properly maintained.

5. Results To Date

The ideas discussed above have been implemented in programs written in C, using the Mach C-threads system. They are running successfully on both the RT and on the RP3. The examples used thus far are relatively small circuits, the largest involving about 350 gates (most of them are an order of magnitude smaller). These include sequential circuits using small RAMs and also LSSD circuits. The results on the RP3 for the two largest circuits run are as follows:

Circuit 38: 178 gate ALU (combinational logic) 21 inputs. 50 random tests detected 418/1254 faults.

Number of Processors	Time
1	699
2	376
4	221
8	164
16	128

Circuit 30: A small circuit (designated DCDEX) designed for the RP3 by Rory Jackson. 354 gates (including 20 latches), 24 inputs. 45 random tests detected 396/1380 faults.

Number of Processors	Time
8	332
16	219
32	185

The efficiency of the algorithm in utilizing multiple processors is obviously a function of the size of the circuit being processed. For example, if the number of gates per level is relatively small, then there is not much work for each processor to do relative to the overhead involved in getting them into play. Thus, I would not expect this program to be really effective on circuits with fewer than many thousands of gates.

At the input end, John Heaven has written some programs for converting circuit descriptions generated via the SCALD graphics system to a form that can be used by my programs. This would allow us to integrate my simulator into the current design environment at Hawthorne and to handle larger circuits. He has also done some preliminary work (on the uni-processor version) to improve the input interface, and also to reduce memory requirements. I have not yet incorporated these ideas into my programs.

6. Proposed Further Work

I would like to continue along the following lines:

(1) Thoroughly test the present version of my program on the RP3. Include tests on real logic chips, such as those used in the RP3 itself.

(2) Make measurements on the program to determine its speed and where the bottlenecks are. Then determine how to eliminate them.

(3) Modify the program to make it more efficient. For example, it would certainly be useful to control memory allocation to ensure that the variables used exclusively by a processor are assigned to its local memory.

(4) Incorporate into my program some of the new features mentioned above (at the end of Section 5), particularly those pertaining to the input interface.

(5) Simulate some large, real, circuits to get some good speed measurements. This depends on the enhancement of my program so that it can fit in with the input interface referred to above.

(6) Look into the problem of test vector generation.

(7) Consider incorporating timing measurements.

(8) Consider allowing simulation of multiple faults.

(9) Consider applications to asynchronous sequential circuits.

(10) Develop some ideas about program checking and debugging on an RP3 type machine.

(11) Determine what characteristics of the RP3 are impeding faster operation of my program, and how might these be feasibly improved.

(11) Evaluate various RP3 features and generate some ideas for other new hardware and/or software features that would be useful and practical. (For example, how can the wait operation best be implemented?)

(12) Consider how to extend the ideas outlined here to systems using built-in-test.

(13) Since this is a research project, fruitful ideas for further work are likely to develop along lines not now evident.

7. Running Programs on the Current Version of the Simulator- Details

At present, circuit descriptions are in the form of a set of assignment statements, specifying the gates and the inputs to each gate. These descriptions are in files designated, for example as, ckts/38t.c. (All are in the directory ckts, the integer refers to the specific circuit, and the letters, such as "t", refer to variations in the description forms for various versions of the simulator. For example, versions 13 and 14 of the simulator work with circuit descriptions of type "t".)

The circuit description includes variables ninpts (the total number of inputs- including feedback inputs), ngts (the total number of gates - including latches), nmems (the number of feedback inputs), and ncl (the number of clock signals). Circuit elements are, in general, multiple-input, single-output devices, described by a structure labelled "gate". (Input and clock signals are also specified as gate structures.) The gates are organized in a 1-dimensional array, g[], with indexes running from 0 to ngts - 1. Inputs are in an array i[] with indexes ranging from 0 to ninpts - 1, and clock signals are in an array c[] with indexes ranging from 1 to ncl. (This is true for the latest 2 versions of the simulator; in earlier versions all indexes start at 1.) For gate g[i], the type is given by g[i].fn. For example, the statement g[3].fn = 'b' specifies gate 3 as a NAND-gate. The statement g[3].fanin = 4 specifies that g[3] has 4 inputs. The statement g[3].inp = new = gen_ptr_array(4) creates a pointer to an array of 4 pointers that will contain pointers to the 4 inputs. These are specified by further statements such as: *new = &g[2] (indicating that the first input is from g[2], *(new + 1) = &i[17] (indicating that the second input is from i[17], etc. A feedback input i[22] receiving its signal from a latch g[6] would be specified by the statements: i[22].inp = new = gen_ptr_array(1), and *new = &g[6].

Circuit descriptions are compiled separately into object code and linked later to the compiled simulator programs. A shell script,sim, is used to compile and run. It calls on various make files to ensure that up-to-date versions are produced. Sim has 4 parameters. They indicate whether the program is to run on the RT or on the RP3, which version of the simulator is to be used, how the faults are to be supplied, and what circuit is to be tested. Thus, the command sim rt 14 m 38 would run version 14 on the rt with circuit 38. The parameter m specifies that "most" faults are to be generated, meaning that both stuck-at-0 and stuck-at-1 (abbreviated here as @0 and @1) will be generated at the outputs and inputs of each gate, but there will be no duplication of the same fault at the output of a gate with fanout 1 and the input of the gate it feeds. Other options are "a", which does not eliminate the redundancy described above, "d", which does not include a fault if all tests for some other fault also test for it, and "l", which calls for a user supplied list of faults. (These are taken from a file tflts/38, where 38 is the circuit number.)

In all cases, the test vectors are taken (for circuit k) from file tsts/k. A preface file, called tpref/k (pref/k for versions of the simulator prior to 13), corresponds to each circuit. The first line of this file, is a y or n, indicating whether or not detailed printouts are desired. The next line specifies the number of processors to be used. Next comes a list, terminated by a line with a "." of gates whose outputs are to be printed (if the first line is a "y"), and, then comes a similar list, also terminated by a ".", of the observable gates (i.e. observable for fault detection purposes). The sim shell concatenates tpref/k, tflts/k (for the "l" case), and tsts/k into a file called inpt. The executable program is placed in the file simprog. Where the RT was specified, sim causes simprog to be executed, with inpt redirected to it and the results redirected to res/k (where k is the circuit number). When the first argument given to sim is rp3, it is necessary for the user to ftp simprog and inpt to the RP3 and then telnet the appropriate command to call for execution. In my RP3 file, I have a shell called rrp3 which causes simprog to be

executed with inpt as the input, and the result redirected to outpt.

The shell sim expects to find a make file for each combination of its first 3 arguments. Thus, for example, there is a makefile mk-rt-14m associated with the command sim rt 14 m k, (k is a circuit number). In order to ensure proper recompilation when different circuits are to be processed, there is a dummy file and a circuit number file associated with each make file. For example, dum/rt14m and cn/rt14m are associated with mk-rt-14m. dum/rt14m is touched whenever the circuit designation is changed (as indicated by the contents of cn/rt14m) so as to force the recompilation of the file that becomes simprog. Constants are in the file constants, and definitions are in the file glbldef1.h. (Versions prior to 13 use glbldef.h, and still earlier versions use other files. All these are in the directory simulators/.)

8. Fault Simulation of Combinational Logic Circuits- Closer Look

There is a block of storage with the structure "flt" associated with each fault (see the file glbldef1.c for definitions of the key stuctures such as gate and flt). For combinational logic, the only information that is contained in this block is the description of the fault: namely the type, the specification of the terminal involved, and the value (at which the faulted node is stuck). For example, a @0 fault at input i[4] would be described by specifying the type as an input fault (type 1), the index of the input (in this case 4), and the value (in this case 0). A @1 fault at input 2 of gate g[5] would be described as a stuck fault at a gate input (type 2), gate index (5), gate terminal (2), and value (1). A @0 fault at the output of gate g[9] is similarly specified (the type is 3 for such a case). Where faults are presented to the simulator in lists generated by the user (in a file tflts/k) by giving the type, value, terminal and index in that order. For example,the preceding 3 faults would be entered as: 100,2, 212,5, and 300,9, respectively.

Each such description, referred to as a fault descriptor, is pushed onto one of nproc stacks, where nproc is the number of processors to be used by the simulator. The fault descriptors (the abbreviation "fault" will be used where the meaning is clear) are assigned to these stacks in round robin fashion. There are actually two fault stacks associated with each processor (i.e, processor n has the 2 stacks pointed to by pointers topfstk[0][n] and topfstk[1][n]). The initial set of faults is distributed among the 0-stacks. When a processor has simulated the results of the first input for a fault on its 0-stack, if the fault has not been detected, then it is, again in round robin fashion, pushed onto the 1-stack of some processor. (Otherwise it is discarded.) For the next input, the processors work on the faults in their 1-stacks, pushing undetected faults onto 0-stacks, etc.

In the routine fwork, processor n pops an element off its active fault stack, pointing to it with faultptr[n], and then calling the routine procfault to process it. Procfault calls injflt to analyze the fault description, and set the stage for procfault to call the routine checkfault. Among other things, it injects the false value for an input stuck fault (if it differs from the valid value) and calls the routine updlevf to determine which gates receive signals from the faulted input and must therefore be evaluated. Those to be evaluated are placed on an "active gate stack" for the appropriate logic level, called (for processor n) actgts[n][lvl]. It uses the routine updlevf for this purpose.

Checkfault determines the low end (fltlev) of the range of logic levels that must be processed and calls on the routine eval to evaluate, in proper order, the appropriate gates. It analyzes the results of each evaluation to determine (calling on updlevf for this) what other gates must be evaluated, and whether the fault has been detected. It passes the results of its efforts back to procfault. Procfault records the results (printing out if specified), and calls pushfault to place the fault on the appropriate stack if it has not been detected. It then calls on restorez to restore to the valid values any gate outputs in processor n's copy of the gate outputs (signal designated, for example, as g[16].z[n]) that it changed as a result of the fault simulation. Restorez also cleans up other data changes made during the simulation of the current fault. (In order to facilitate the restoration process, a stack called "changed" is maintained by checkfault of all gates whose outputs it has changed.)

Appropriate statistics are gathered at each stage, and, when all processors have worked through the members of their fault stacks, the next input is entered to the valid circuit, valid gate values determined (and distributed to each processor), and fwork called again to oversee the checking of the faults now on the alternate stacks. The program terminates when there are no more inputs or when all

faults have been detected.

9. Fault Simulation of Sequential Logic Circuits- Closer Look

If a circuit has memory, in the form of storage elements such as latches, or FFs, then the fault simulation process must take this into account by keeping track of faulty states of such devices. Suppose that, as a result of some input acting on a cirucuit with a stuck fault at some gate terminal, no observable output is changed (so that the fault is not detected), but the states of one or more latches are affected (i.e. are different from their valid circuit values as a result of the existence of the fault). Then it is possible that a subsequent input, in conjunction with these false signals may propagate a false signal to an observable output, hence revealing the fault. In order to simulate this behavior, the fault descriptors, introduced in the preceding section, include pointers (fltchstk) to stacks listing latches (references to latches also apply to FFs) whose values become false due to the original fault.

The checkfault routine pushes onto a stack, called newltchstk, latches whose values have been changed from their valid values in the manner outlined above. For faults not yet detected, procfault attaches newltchstk to the fault descriptor. Injflt copies the stack of latches with false signals (if such exists as part of the descriptor of the fault currently being processed) onto a stack called oldltchstk. The latches involved are pushed onto the appropriate actgstks. Injflt then treats the corresponding feedback inputs (if any) as though they were terminals with stuck faults. Checkfault takes into account the existence of such latches when determining fltlev (lowest level of gates to process). Procfault and restorez free memory allocated to these stacks when no longer needed.

Suppose that, during a faulty circuit evaluation of a latch, it is found that the clock input to that latch is 0. Then the output of that latch should be the same as the output it had during the previous input. But how is that value to be found? If the latch is on oldltchstk, then the entry on that stack will contain the required value. But if the latch is not on oldltchstk, the proper value is the valid output of the latch during the previous input. But the present value of the valid output of the latch (which is the result of the present input) may be different from the past value (the clock input for the valid circuit may be a 1, as opposed to the 0 for the faulty circuit). In order to take care of this situation, the data structure for a gate includes oldz, the value of the valid output after the last input, and a stack, chltchstk, of latches whose values were changed by the current input is maintained during the simulation of the valid circuit. The routine updtoldz is part of this process. (I believe that oldz can be eliminated and the old latch value can better be kept on chltchstk.)

10. Fault Simulation of Logic Circuits Containing RAMs

Since RAMs (usually of modest size) are sometimes included in logic circuits, it is useful for the simulator to be able to treat them as gates during simulations. This does, however introduce some complexity into the process of fault simulation.

For the valid circuit simulation, a memory location is reserved for the contents of each memory location of each RAM. These are maintained in a straightforward way during the valid circuit simulations. (RAMS are treated as multi-input, single output gates, in a manner similar to the way latches or NOR-gates are handled.) For reasons similar to those motivating the need for the stack of changed latch values (see preceding section), it is necessary to maintain a stack (chmemstk) of RAM locations whose values have changed (along with the old values).

In order to handle fault simulation of such circuits, some further additions to the data structure are needed. Each fault descriptor must contain a pointer (fmstk) to a stack of faulty memory locations (the index of the RAM, the local address within that RAM, and the faulty value are all stored on that stack for each false value in a RAM, whether due to a stuck fault in memory or to the consequences of other faults). During fault simulation, injflt generates a pointer (oldfmstk) to this stack, whose contents may be altered during the simulation. In addition, a new stack (newfmstk) of memory locations that acquire false values for the current input is produced by checkfault. If the current input does not detect the fault, then oldfmstk and newfmstk are concatenated in procfault, and the result attached to the fault descriptor via fmstk, so it becomes the oldfmstk for the next input.

When faults are entered by list (the l option) memory faults are entered in a special form (see the file crfltlst6.c). For example, m12,3,0 specifies a memory fault in which location 3 of the RAM g[12] is @0. This description is converted by the program to a form analogous to that for the other fault types. Its internal form beginning with a 0 for the type, would be 000,12, and there would be a pointer fmstk to a stack of false memory values that would have, as its first (bottom) entry, a structure (mvalstk) with components indicating false value (mfval), gate index (gtindex), address (mloc), and a pointer to the next item on the stack (nxt- initially NULL).

All RAMs with false memory contents are placed on the actgtstk by injflt. When eval is evaluating the output of a RAM during a faulty circuit evaluation, a number of situations must be taken into account.

(1) If the operation is neither write nor clear, then the function chkvmw is called to determine if the valid circuit simulation changed any memory bits in this RAM. If so, a false memory value listed on oldfmstk might have been corrected or changed, in which case an element of oldfmstk is deleted or changed (if it does not correspond to a stuck fault). If the memory location is not on oldfmstk, then a new false memory entry must be created and put on newfmstk.

(2) If the faulty circuit simulation is executing a read operation, then rdfm searches oldfmstk for the location involved; if it is there, then the output is the associated false value. Else it is the valid contents of them memory location.

(3) If the operation is write, and the value t to be written differs from the valid circuit value stored at the specified memory address, then prmfch is called to see if that location is on oldfmstk. If it is, then prmfch updates the value on that stack if necessary. Otherwise prmfch adds a new item to newfmstk. If t is equal to the valid contents of the memory location, then another function, prmnfch, is called to search oldfmstk for an entry at this location and to delete it if it exists (and is not a stuck fault). In both cases, chkvmwb is called to determine if the valid circuit simulation wrote at diffeent location of the same RAM. If so, then it may be necessary to add a new fault to newfmstk.