The EVE Companion Simulator

D. K. Beece1, R. Damiano1, G. Papp1, R. Schoen2

1 IBM Watson Research Center, PO Box 218, Yorktown Heights, NY 10598
2 IBM Application Business Systems, Hwy 52 & 37th NW Rochester, MN 55901

Abstract

This paper describes the EVE Companion Simulator, ECS, a four-valued gate level simulator which complements a simulation methodology that uses the Engineering Verification Engine, EVE. ECS is, like EVE, a zero- and unit-delay cycle simulator, but unlike EVE, ECS is a software simulator which runs on a general purpose computer. The basic simulation paradigm of ECS is event driven to exploit latency. ECS uses the same design description interface as EVE, enabling an EVE model to be run on either the hardware accelerator or the software simulator. The major use of ECS is in the interactive debug of large models, where the model is run on the hardware accelerator, a checkpoint is taken and then loaded into the software simulator. In this paper, we give an overview of the ECS architecture and simulation technique, and describe how ECS is used in conjunction with EVE.

(1) Introduction

Simulation plays a central role in virtually all verification methodologies in VLSI. Often, the computer and human effort spent in simulation is a major contributor to the total design effort[7]. This large cost and the impact of simulator performance on design schedules, particularly for very large models, has led to the increased use of hardware accelerators[2,3,4,5]. Typically, these machines support only one user and are loosely coupled to a host that is used for programming, load and control. The result is very high performance for large models which require only relatively simple run time interactions. Unfortunately, interactive use, while possible, is often not cost effective, as these machines are so expensive that their use can only be justified if they are highly utilized. Thus, the vast majority of applications on these machines are batch.

There are two basic techniques which can be used during the design process to debug a failing simulation test case: iterative batch or interactive. In an iterative batch approach, the same test case is run many times in batch mode, gathering data on more and more nets that might indicate the cause of the problem. The analysis of that data is done off-line. This approach is efficient for a hardware accelerator, but is not an efficient way for the designer to spend his time. In interactive mode, the test case is run just before the failure is believed to occur. Simulation then proceeds in small time increments, carefully observing any net as needed to understand what is causing the problem. When a hardware accelerator is run in an interactive mode, it is often stopped while the designer looks at net values. It is an efficient way for the designer to debug the problem, but it leaves the costly hardware accelerator idle for periods of time, unavailable for other jobs. In many cases, the the cost of the interactive job is so high that schedules simply preclude running them in the hardware-only design methodology.

Software simulators do not suffer the restrictions of single-job scheduling. However, except for the interactive simulation application, the cost to a design project of running a simulation job on the hardware accelerator is usually less than a software simulator running on a general purpose computer. In this paper, we describe a software simulator which has been specifically developed to be used in conjunction with a hardware simulator. This software simulator, which we call ECS, for EVE Companion Simulator, gives the designer the flexibility to choose how to run a given simulation job as efficiently as possible, maximizing the productivity of both the hardware and human.

With a companion software simulator, there is now a third debug method available to the designer, called checkpoint mode. Using this technique, the simulation job is periodically checkpointed. Then if the test fails, the last good checkpoint is used to initialize the same model on the software simulator and the debug continues there. The hardware is available for other jobs, yet the designer gets an efficient way to debug the failure.

In order to effectively couple a software simulator to a hardware accelerator, the representation and interfaces for both must be the same, so that it is functionally transparent how the job actually executes. In addition, the software simulator must have sufficient performance and capacity so that it can run any model which is run on the hardware. Given the large memory sizes available on todays high-end processors, the capacity of the companion simulator we have devised is sufficient to run jobs which contain several million gates and a large number of array constructs.

In sections 2 and 3 we describe the ECS simulation algorithm in some detail. In section 4, we illustrate how
ECS complements a simulation methodology which employs an EVE. Performance and capacity are discussed in section 5.

(2) ECS Overview

ECS is an event-driven 4-valued cycle simulator and, as in EVE, the simulator primitives can model both zero-delay and unit-delay logic. The basic ECS model is made up of EVE primitives, either 4-input, 1-output gates or array (memory) descriptors, exactly as they are described for EVE.

A commonly used approach for gate evaluation[2] gathers the values of the respective inputs for the gate from a value table, and after performing the operation, applies the new output value back into the value table. ECS does not gather inputs. Rather, the outputs are spread to all successor gates. This spreading of the outputs serves the dual purpose of propagating values and scheduling events, thus reducing execution time and storage. In order to further reduce the execution time, latches can be "owned" by a clock. If the clock is disabled, no latch events owned by that clock will be executed.

The events are scheduled through the concept of cycle phases. All the logic to be simulated is ranked ordered, so that every gate is assigned a rank. The algorithm itself uses several tables based on this rank ordering and on connectivity information. A brief overview of these tables is given in the rest of this section.

The next section describes how these tables are used.

(2.1) Cycle Phase Table: The cycle phase table provides the basic state changing mechanism for scheduling. In essence, it is similar to a time wheel[7] but the rank is used to determine the offset into the array. Since there are additional phases beyond those associated with zero-delay logic, we use the term phase instead of rank. Thus, there is one entry for each potential phase, including zero-delay phases (as given by the rank ordered model), clock phases, unit-delay phase, etc (see section 3.2). During simulation a stack of executable events is assigned to each phase.

(2.2) Event Table: Events are grouped together into a table which corresponds essentially to the instructions on EVE. In addition to these gate events, other events are included to represent unit-delay buffers which will be referred to as slow events, array simulation, simulation optimization processes, and special events to facilitate interaction with the symbol table and with the test case. Each entry in the event table has a fixed length area coupled with a variable sized area used to address the fan-out events. The fixed area contains a type field to classify the event (zero delay, clock, latch, etc), along with the current input and output values, function table index, plus additional information needed for test case functions and the specialized EVE DeMorgan operations. In addition, the event table contains an index into the cycle phase table which is used to schedule the event into the appropriate stack.

(2.3) Function Table: In analogy to EVE, the function of each gate is specified in the event table by an offset into the function table. The concatenation of the gate's current inputs with this offset provides a very efficient lookup structure. DeMorgan translations are also stored in the function table. The actual DeMorgan translation not only performs the function modification but also shifts the result 0, 2, 4, or 6 bits to provide a quick store into the appropriate successor input.

(2.4) Array/Port Table: ECS models arrays in exactly the same way as does EVE. Each array is accessed via ports, which are assigned a priority for ranking. Each port consists of an enable, a direction (read or write), address and data lines. In addition, for a write port, a set of mask lines is provided to support partial update, in essence a read-modify-write port. These array/port specifications are managed through two data structures, an array descriptor table and a port table. The array table contains information pertaining to the general array structure such as the number of rows and the length of an individual row. The array table also indicates if the array is dense or sparse (see section 3.4). The symbol table entry for the array contains a pointer to this table. Each array table has an associated list of port tables, which correspond to the read and write ports for the array.

(2.5) Symbol Table: While not needed for simulation proper, a symbol table must be used to provide the connection between test case commands and the simulation model, for example to spread or gather a value from a user exit and the run time controller. Minimizing the storage required for a symbol table is an important task, especially for those symbols that share some commonality such as subscripted buses. To achieve this, the event table is arranged so that all gates that share a subscript are sorted and contiguous in the table. This permits the use of a single pointer to locate all the common events.

(3) Event Scheduling

The basic underlying simulation algorithm is an event-driven spread algorithm, using the cycle phase stack. During simulation each of the cycle phases is examined and the scheduled events on the stack executed.

(3.1) Initialization: Each top-of-stack pointer in the phase array is initialized with one element denoting a change-of-phase. One of the actions associated with this change-of-phase condition is to advance to the next stack in the phase array. When the simulation algorithm finishes execution of a stack, these change-of-phase events are re-applied to the top of the stack, hence they remain initialized on all subsequent cycles.

(3.2) Simulation Cycle: At the start of each simulation cycle, the slow events, which represent delayed
updates from the previous cycle, are executed in what is called the unit-delay phase. During this phase, the outputs of gates evaluated on the previous cycle are spread to its successors which need a unit of delay on input. Normally, all zero-delay spreading from the gate was performed directly after the gate was computed during one of the zero-delay phases. However, if a gate needed to spread to one or more inputs with a unit of delay, then a unit-delay event was created, initialized with the output value, and pushed on the unit-delay stack.

The next set of phases correspond to the various zero-delay ranks of the model’s gates. The gate events on these stacks cause the evaluation of a gate and the spreading of the gate’s output value, if changed, to its successors. The last event on each stack denotes the end-of-phase condition, causes the next phase to be entered.

The zero-delay phases are followed by the clock phases. Before discussing how these are scheduled, we first describe the notion of clock ownership of latches and update events.

Update events are similar to gate events, the fundamental difference is that they are controlled by a clock. The output associated with an update event remains unchanged until the controlling clock becomes active. Thus, once a latch owned by a clock is scheduled, it remains scheduled until the clock is enabled. This control reduces the number of times that these events are executed. Typically, a clock is disabled with a 0 value. Without the guard, all latches would execute every time their clock dropped from a 1 to a 0 with no change in their output. Likewise, if a clock rises from a 0 to a 1, then all latches would be executed without a clock guard, even if no change has occurred in the other inputs. Thus, if latches can be identified as special gates, assigned to clocks, then most can be rank ordered as zero-delay gates, without the need for an additional slow (unit-delay) event. The exception is a latch which directly feeds another latch, which is resolved through a slow event. Since latches account for between 5% and 10% of the gates in a typical simulation model, these factors combine to produce significant overhead, slowing the simulation, especially in cases where the overall simulation activity is small.

During the clock phases, the outputs of those gates which are associated with an active clock are spread. For EVE, these updates or latches are formed by having a primitive gate which have only slow fanouts, that is, have a unit of delay on all successors. Each clock event stack, corresponding to each clock phase, contains a single clock event that controls the corresponding latches. When this event is executed, a comparison is made between the current clock value and those values that disable the clock. If the clock is enabled, the simulator moves to the next phase, which contains a pointer to the list of update events (described next) which have been assigned to that clock. If the clock is disabled, the next phase is skipped.

The end-of-cycle phase is the last phase in a cycle. This stack can have break events on it. These events specify conditions under which the dispatch of events should be suspended and control should be returned to the run time controller or some user exit routine. EVE provides an analogous function, via hardware interrupts. By using break events, the status of simulation can be accessed before the specified number of cycles have been run.

The very last event on the end-of-cycle phase is called the end-of-cycle event, which checks for two terminating conditions. The first check is for breaks; if any break occurred the control returns to the run controller unconditionally. The second check is a comparison of the cycle time against the number of cycles the simulator has been requested to run. If there are cycles remaining to be run, the start of cycle phase is re-executed. Otherwise, control is returned to the run time controller.

(3.3) Spread Algorithm: The simulation loop simply runs through the phase array of stacks, executing all elements on each stack. If it finds a gate event, the new output of the gate is computed using the input string, function index, and DeMorgan translation. The new output is then compared to the current output. If they are the same, there is no further processing necessary for this event. Otherwise, they are different and the current output value is changed to the new value and the new value is spread to all successor inputs while scheduling the events associated to these inputs for later execution.

The spread of the new output value proceeds as follows. A loop is entered to spread the new value to the appropriate input strings in the fan-out array. A fan-out end flag signifies that the fan-out is the last one for the gate. The fan-out position index indicates whether the new output value needs to be placed over the first, second, third or fourth input (right to left) value in the input value byte for the associated fan-out event. The fan-out DeMorgan rule provides a translation rule (typically no-operation or complement) of the output value before the input value is replaced. The fan-out flags act as an index into a fan-out branch table. The branch table contains the address of the appropriate code needed to compute the DeMorgan translated output value and correctly place the value into the fan-out event’s input byte.

The fan-out event with the new input must now be scheduled for execution at some time in the future. However, it is quite possible that the changed input value will not affect the fan-out events output value. Therefore, a look-ahead evaluation of the fan-out event is performed. If the new output differs from the current,
then we schedule the fan-out; otherwise we move to the
next fan-out or event on the stack. We have found that
on the average 50% of all potential fan-outs need not
be scheduled and since scheduling is expensive com-
pared to evaluation, this look-ahead technique signif-
ically optimizes the algorithm. If a fan-out event needs
to be scheduled, then we can use the fan-out event’s
phase offset which contains an offset into the cycle
phase array. This offset is a negative number and rep-
resents an offset from the end address of the cycle
phase array. The phase array element address is de-
termined and the new event is pushed on the stack.
The fan-out event’s cycle phase offset is then overwrit-
ten with the stack chain pointer. This process has the
added advantage that the positive/negative nature of
the phase offset field tags the event as scheduled or
not. The negative phase array offset is regained after
the event is executed. If a fan-out member is tagged
as the end of the fan-out list, then a special dual fan-out
routine is called that duplicates the normal fan-out code
selected through the fan-out flags but leaves the loop
after execution.

If the output needs to be spread with a unit of de-
lay, then a member of the fan-out array has a pointer to
a unit-delay event. An event has at most one member
of its fan-out list associated to a unit-delay event, which
simply spread the delayed output to the appropriate
inputs.

(3.4) Array Processing: Arrays are processed
through special events which gather information into
ports (read and write port events) and which spread
the information into (write row) and out of (read row) the
array. As with EVE, a concept of mask and enable have
been incorporated into this processing.

The gather events simply spread the correspond-
ing enable, address, data, or mask value into a contig-
uous area used for processing the arrays. If any of
these values change on a given cycle, then a write or
read port event, the actual trigger for the array access,
is scheduled. The write and read port events have an
associated port table which contains pointers to all in-
put data areas (address, mask, input data, and enable).
If the enable for the port event is 0, then the event is
skipped. Otherwise, the access is done by processing
a list of row events, with a special end-of-sub-stack
event used to indicate that the row has been totally
processed.

The write access is done via write row events
which take the mask into consideration before writing
to any byte of the row. One subtlety occurs when a
write port event uses the same port address as that of
a read array event previously executed. In this case,
there is an implicit read event that must be executed.
ECS checks all read ports whenever a write event oc-
curs to schedule any corresponding read array events.
Notice that because of rank ordering, these read events
may occur on the next cycle.

The read access is done through read row events,
one for each element of the array row. Each row read
event processes the element and spreads the outputs
appropriately, if the new value differs from the old. A
trace value can be reported if desired.

There are two flavors of arrays provided, dense
and sparse. A dense array has two bits of storage for
each array value. This provides the best performance
results, but can be a problem when the arrays are
large. Thus, a second type of array, called sparse, is
used. These arrays are assigned a linked list which
contain only those rows that are accessed (initialized,
written to or read from) during simulation.

(4) ECS in an EVE Methodology

As was discussed in section 1, the software com-
panion for EVE needs to have the same simulation rep-
resentation as that used on the hardware, to facilitate
the direct conversion of checkpoints, as well as have
sufficient capacity and performance to be able to simu-
late EVE models in reasonable time.

EVE models are derived from IBM design[2,8]
languages, either BDL/CS or BDL/S. BDL/CS is a reg-
ister transfer language which must be converted by a
synthesis process into the gate and array primitives
needed for EVE. BDL/S is a structural description
which must be expanded to run on EVE. Both lan-
guages are supported by existing software simulators
within IBM, but neither of these simulators was a viable
candidate for the EVE companion simulator.

For BDL/S, the issue was one of capacity and per-
formance. The BDL/S simulator[9] used in IBM sup-
ports a large number of different delay modes and
complex test case functions. The scheduling and stor-
age overhead to support these functions is substantially
more than in ECS, which duplicates the only scheduling
algorithm supported on EVE. The BDL/S simulator
needed to be enhanced to provide sufficient perform-
ance at the large capacity to support EVE size models.

ECS essentially is that enhancement.

The BDL/CS problem is more subtle. BDL/CS is
normally simulated[8] by translation to polish strings,
which are then evaluated, a fundamentally different si-
mulation representation than that used for EVE. The
EVE representation has many more internal nets, many
of them latches used to hold state information. Thus,
while it is possible to convert EVE checkpoints to re-
store into the BDL/CS polish string simulator, the re-
verse process is substantially more difficult. ECS does
have comparable performance to the BDL/CS software
simulator, but at the expense of more storage: polish
strings are substantially more compact than gates.

The characteristics of ECS make it an ideal soft-
ware companion to EVE. Most importantly, there is
net-for-net equivalence in that every net and array in
the EVE model exists in the ECS model because they
are compiled from the same source. In addition, ECS also supports four-valued simulation and the same underlying timing model (cycle simulation), so the checkpoint data can be mapped directly between ECS and EVE. This is very important in order to get one model to the same simulation state as it was in the other when the checkpoint was taken. It also means the simulation will behave the same from that point whether on ECS or EVE. Also important is the fact that the same simulation control commands exist in ECS as in EVE. Therefore, the designer can treat ECS just like being on EVE. Finally, as discussed in section 5, the performance and capacity of ECS is sufficient to simulate large portions of time to get to a failing condition.

ECS can be utilized in all the major phases of the design verification, including initial model bring-up, architecture verification, and regression testing. However, the primary use of ECS is in architecture verification, when the speed of EVE is utilized to execute all needed tests. The failed tests are debugged on ECS, either from the start of the test case or from EVE checkpoint data. EVE is restricted to working on a single job at a time, but since ECS runs on a general purpose architecture, many jobs can be running simultaneously.

ECS is also used during the initial bring-up of new models. During model bring-up, few cycles are simulated so the speed difference between EVE and ECS is minimal. In many cases, it is more efficient to stay completely in ECS until the design has passed certain reset, flush and/or initialization tests and component interconnections have been verified. Until this occurs, EVE is available to run other simulation jobs.

When the design is in regression test, few design changes occur. If the design is changed, all the tests must be re-run as quickly as possible to verify that the change fixed the problem and did not introduce other errors. For this application, the performance of EVE is critical and having more EVES to run the tests concurrently would enable faster completion of the regression test. In this case, ECS is used to supplement the EVE hardware during this peak usage period. It also provides a back-up for EVE when the EVE system is unavailable.

(4.1) Model Compilation: The model build for ECS is a subset of that used for EVE. In order to use EVE, all design source data is converted into an intermediate representation called a NODES file, which is the basic language interface for the EVE compiler[2]. These NODES file(s) are processed for EVE in a multiple step process. ECS has a separate, substantially simpler compiler, which basically parses the NODES file, rank orders the gates and array/port descriptions and outputs the necessary tables.

(4.2) Run Time Interaction: The run time control of the ECS model supports exactly the same test case files as EVE. Except for hardware debug commands, all the simulation related functions are supported by ECS. This allows exactly the same run time system to be used for both simulators, a consistent interface which can be used either interactively or in batch mode. For example, the EVE test case language has a "load" command which is used to load a specified model onto the hardware. In the ECS version, the load command loads the ECS model and initializes, e.g., the phase array. Designer provided commands further initializes the model through the setting net and array values.

The test case command used to change a net value must not only change the gate event output value, it must also change the "spread value" on all fan-outs. It also assigns the fan-out events to the appropriate phase for later execution. Special events, such as those needed to process equates (symbol table synonyms) and other linkage with the symbol table, are created by the compiler to aid the run controller to access the net values.

(4.3) Results Processing: ECS gathers trace data and displays it in exactly the same format as EVE.

(5) Capacity and Performance
We conclude with a discussion of measurements we have done to characterize the capacity and performance of ECS. Our discussion is based on three models, ranging in size from 5000 to 50000 gates with arrays. The data is shown in the table below and described in the remainder of this section.

<table>
<thead>
<tr>
<th></th>
<th>totals</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>gates</td>
<td>5051</td>
<td>34392</td>
<td>49898</td>
<td></td>
</tr>
<tr>
<td>arrays</td>
<td>0</td>
<td>13</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>events</td>
<td>6996</td>
<td>51754</td>
<td>64249</td>
<td></td>
</tr>
<tr>
<td>fan-outs</td>
<td>11836</td>
<td>81516</td>
<td>141111</td>
<td></td>
</tr>
<tr>
<td>compile time¹</td>
<td>3</td>
<td>22</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>storage²</td>
<td>0.4</td>
<td>2.8</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>cycles</td>
<td>519025</td>
<td>42847</td>
<td>141</td>
<td></td>
</tr>
<tr>
<td>events³</td>
<td>32.28</td>
<td>35.85</td>
<td>0.52</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mixed asm/C</td>
<td>33.24</td>
<td>40.70</td>
<td>1.57</td>
</tr>
<tr>
<td>run time¹</td>
<td>33.15</td>
<td>40.28</td>
<td>0.53</td>
</tr>
<tr>
<td>sim time¹</td>
<td>84.32</td>
<td>76.64</td>
<td>2.07</td>
</tr>
<tr>
<td>sim time²</td>
<td>84.23</td>
<td>76.16</td>
<td>1.03</td>
</tr>
<tr>
<td>run time³</td>
<td>677.0</td>
<td>496.0</td>
<td>16.0</td>
</tr>
<tr>
<td>sim time⁴</td>
<td>677.0</td>
<td>496.0</td>
<td>8.0</td>
</tr>
</tbody>
</table>

¹3860-E CPU sec (VM/CMS)
²Megabytes
³in millions
⁴RT-125 elapsed sec (AIX)
The compile time from EVE NODES format to ECS model is fast and scales linearly. For an EVE compile, the model must be partitioned among the different processors, the switch must be scheduled, etc. In contrast, the ECS compiler only needs to rank order the gates, which is actually one of the early steps in the EVE model compile. ECS models are thus much faster to build than EVE models.

Storage size scales linearly, even with the addition of arrays. The data in the table can be extrapolated to show that a model consisting of a million EVE gates takes approximately 40 megabytes of storage.

We have two versions of ECS, one written completely in C and one with the dispatch kernel written in System/370 assembler. The first version is portable and can be run in a variety of environments. The second version, which replaces approximately 500 lines of C with 1300 lines of assembler, was written to optimize performance on our mainframe. The results of the mixed C/Assembler version on the 3090 and the pure C version on an RT/PC are shown in the table.

Since the algorithm is a spread algorithm, we have included totals for the events and event fan-outs created by the compiler. The difference between the event and gate totals comes from the addition of the unit-delay (slow) events and the events needed to spread and gather array data with the gate events. It should be mentioned that in the M1 and M2 models, the majority of the gate inputs have a unit of delay and all latches have been expanded to simple logic functions, with explicit feedback, referred to as an EVE unit-delay representation. In contrast, M3 has special latch gate primitives and only the inputs latch successors have a unit of delay; all other logic is driven with zero delay. This kind of model is called an EVE zero-delay representation.

One reason for the variation in the cycle time for the three models comes from the fact that the first two models come from a unit-delay representation while the last is from a zero-delay representation. For any zero-delay model, the simulation cycles represent the logic cycles described in the model. On the other hand for a unit-delay model, many simulation cycles must be run to correspond to an equivalent logic cycle. For example, in the M2 model, the ratio of simulation to logic cycles is 67:1, meaning it took 67 cycles in the simulator to make one logical cycle.

The table shows both total run time and simulation time. The run time includes model setup and initialization time in addition to the simulation time. In most cases, the difference between simulation and run time is negligible. For M3, the test case driver accounts for a large proportion of the time due to the small amount of cycles run. Typically, this overhead comes from the initialization of arrays. For M3, the mixed assembler/C version is about twice as fast as the pure C version. In addition to the normal overhead associated with compiled code, the efficient use of branch tables in the assembler version accounts for most of this difference.

The mixed assembler/C version can easily process a million events per second on the 3090. The pure C version running on an IBM RT/PC is only 10 to 20 times slower than this, comparing elapsed RT/PC time to 3909 CPU time. Of course, the most important performance metric is throughput, which depends on the loading of the host. Typically, on our hosts this loading makes the ratio of elapsed to CPU time somewhere between 1 and 10. Thus, we believe that for reasonably sized models and a loaded host, the RT provides an attractive alternative.

References


