Final Thesis

The Counting Algorithm for simulation of million-gate designs

by

Klas Arvidsson

LITH-IDA/DS-EX--04/046--SE

2004-05-12

Linköpings universitet
Institutionen för datavetenskap

Final Thesis

The Counting Algorithm for simulation of million-gate designs

by

Klas Arvidsson

LiTH-IDA/DS-EX--04/046--SE

2004-05-12

Supervisor:	Bengt Werner
Examiner:	Erik Larsson

Abstract

A key part in the development and verification of digital systems is simulation. But hardware simulators are expensive, and software simulation is not fast enough for designs with a large number of gates. As today's digital designs constantly grow in size (number of gates), and that trend shows no signs to end, faster simulators handling millions of gates are needed.

We investigate how to create a software gate-level simulator able to simulate a high number of gates fast. This involves a trade-off between memory requirement and speed. A compact netlist representation can utilize cache memories more efficient but requires more work to interpret, while high memory requirements can limit the performance to the speed of main memory.

We have selected the Counting Algorithm to implement the experimental simulator MICA. The main reasons for this choice is the compact way in which gates can be stored, but still be evaluated in a simple and standard way.

The report describes the issues and solutions encountered and evaluate the resulting simulator. MICA simulates a SPARC architecture processor called Leon. Larger netlists are achieved by simulating several instances of this processor. Simulation of 128 instances is done at a speed of 9 million gates per second using only 3.5MB memory. In MICA this design correspond to 2.5 million gates.

Content

1 Introduction

1.1 Motivation

Today’s digital systems are growing larger and larger in terms of complexity and the number of transistors that can fit on one chip. With the concept of system-on-chip entire systems can fit on a single chip. As entire platforms of processor with cache, ASIP, ASIC, memories etc. now fit on one chip, simulation grows important as crucial part in several important design steps. Many subcomponents are found as IP-packages, and early simulation of the system can help detecting problems and making better design decisions. Effects of exchanging a part of a system with a new or updated component can be studied in a simulator, without building a prototype.

Design verification and software development is made easier by letting the engineer see what actually happens inside the chip. Simulation also speed up system development time by enabling co-design. Software development can be started already with an early description of the system, for example an instruction set, which can be simulated before anything is actually built.

This leads to a need for fast simulators capable of handling a high number of gates. One might argue that there are lots of simulators already, capable of simulating almost any abstraction level with high observability, such as ModelSim for VHDL simulation, and we have SPICE if we really want details. But are they fast, and do we need all that observability for all purposes? The answer to both questions is no. With that kind of observability they are not fast, and no, in most cases we do not need to see all glitches and timing issues, timing analysis are better and faster for this [Jennings 91].

But software or hardware LCC (Levelized Compiled Code) simulators then? Yes, but software LCC simulation does not handle very large designs, or is slow for large designs, and hardware simulators are typically very expensive.

In the past decade there has been a tremendous increase in computer performance and complexity. Approximately ten years ago, a workstation would be a 66MHz 486 with 8 MB memory. Today, the equal amount of money would give several GHz and almost hundred times the memory, a memory equal to old times hard drive sizes. Still, not much seems to have happened when it comes to software simulator development. In fact most work found in the area is from the late 1980 or early 1990 decade.

1.2 Purpose

1.2.1 Goal

As briefly mentioned above simulators are practical tools during software and hardware development. This work is intended to investigate how to construct a simulator that can simulate large netlists, and how fast it can be made on a regular workstation. We will implement and evaluate MICA, a Multi-queue Interpretive Count Algorithm simulator. In a previous work a simulator named EMIL was developed. That simulator serves as a reference to verify and compare the result of this work against, and some parts of EMIL will be reused in this work. To get a clearer picture of EMIL, I would like to refer the reader to [EMIL 2002]. Only a brief description is given in section 2.1.

1.2.2 Limitations

The original plan was to implement MICHEL (Multi-queue Interpretive and Compiled Hierarchical Event-driven Levelized simulator), where selected parts of the netlist could be compiled for higher speed, while other parts was to use a compact interpreted representation. This plan also involved optimizing compiled code for the x86 architecture. But lack of time canceled the compiled part, and instead we got MICA, still assuming the x86 architecture.

1.3 Problem Definition

When building a simulator for small to average sized designs we are faced with several problems, such as[1]:

- Which abstraction level should be simulated?
- What signal and timing models should be used?
- Is event-driven or oblivious simulation the best choice?

The answers depend on what we want from our simulator; how much details of the simulated circuit we want to monitor. Secondly, they depend on how long we are prepared to wait; how fast simulation we need. This is a tradeoff between observability and performance. Most often, what we would like is all information in no time. Thus we have a fourth question:

- How do we make it faster?

[1] The reader not familiar with the different concepts such as event-driven, oblivious, zero-delay etc. that is mentioned in this section is referred to section 2.2 for further explanations.

EMIL tries to answer these questions, and as a continuation of EMIL many answers remain the same in this work. We will, like EMIL, use event-driven gate-level simulation with zero-delay timing and only two signal levels. For many purposes, such as evaluating a component together with a full system, or as part of design verification, the observability given by this suffices. Instead, the fourth question, simulation speed is what really matters.

But with a large design, this grows more complex. First we have the obvious expectation that more gates will lead to longer execution time. If 10^3 gates took 10^3 time units we would naturally expect 10^6 gates to take 10^6 time units. Secondly, more gates will naturally need more memory. Modern computers have huge amounts of memory, so by size this is not a problem, but by performance it is. The larger the memory the slower it is, and this breaks that first and obvious expectation. Having a data structure too large to fit in L1 cache will lead to slower execution, and having an algorithm with bad locality for memory accesses will make it even worse. Thus, the fourth question can be reformulated:

- How to simulate large designs fast?

And this is the question this work will investigate. Let us take EMIL as an example and try to picture it with a large design.

1.4 Imagine EMIL with an X million gate design

Let us first examine some data from [EMIL 2002] in Table 1 and Table 2. The EMIL benchmarks was run on a 500 MHz Pentium III processor[1].

	Total	*Each cycle*	*Each second*
Execution time	30s	0.6ms	-
Simulated cycles	50000	-	1667
Gate evaluations	64M	1278	2M
Changed signals	22M	445	739K
Real instructions	5.3G	106K	176M
Memory used	348KB		
Gate activity	14.8%		
Total gates	8637		

Table 1 EMIL benchmark statistics

	Size	*Latency (ns)*	*Latency (cycles)*
Level 1 data	64KB	6	3.0
Level 2 cache	512KB	45	22.5
Memory	256MB	147	73.5

Table 2 Cache statistics

EMIL cache misses was reported to be low, almost no L2 misses and only about 5% L1 misses, synchronous gates excluded (all searched all cycles). This suggests that the simulation netlist structure fit in the L2 cache, and the most frequently used gates fit in the L1 cache. The first suggestion is clearly true, since the memory used for the simulation netlist is only 384 KB and the L2 cache is 512 KB. Making the simplified assumption that it is the same 15% (the gate activity) of the gates that evaluates each cycle we calculate the critical netlist size to about 58 KB ($384 \times 15\%$). With a small margin this fits nicely in the L1 data cache, so our assumption seems true to at least some extent. If it were mainly different 15% of the gates that was evaluated from cycle to cycle, more L1 misses would occur.

[1] This information was not mentioned in the EMIL report, but is from the author of EMIL and confirmed by the cache data in Table 2; 3 cycles divided by 6 ns equal 500MHz.

Now, let us imagine EMIL with an X million gates design. This mean a simulation netlist size of $\mathrm{X}M \times 384K/8637 \approx 40\mathrm{X}$ million bytes. At the *same* speed and gate activity this would take $\mathrm{X}M \times 50000 \times 15\%/2M \approx \mathrm{X}$ hours per million gates. But $40\mathrm{X}$ MB will not fit in any cache, the critical size alone would be about $6\mathrm{X}$ MB, by far larger than any cache level, and since we now will need 73.5 cycles for every memory access instead of 3 cycles we will execute up to 24.5 times slower. Thus, we can expect many hours of simulation.

1.5 Gate-level simulation

Digital systems can be described in several different domains on different abstraction levels. One popular way of showing them are Gajski's Y-chart in Figure 1.

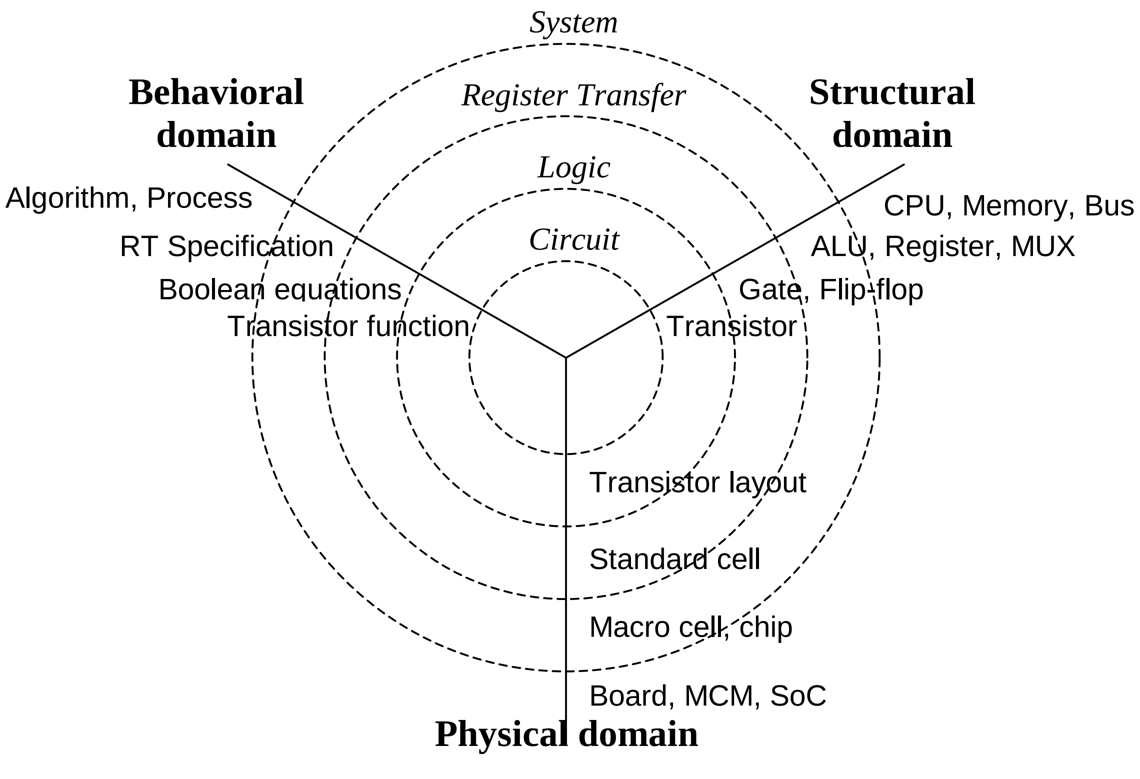


Figure 1 Gajski's Y-chart

There are three domains, behavioral, structural and physical, each with an abstraction hierarchy containing system, register transfer, logic and circuit level. Gate-level simulation takes a structural description of a system at the logic level and returns the behavioral response to any chosen input sequence.

The most immediate view of a gate is a unit implementing one of the basic boolean functions (AND, OR, NOT), but with increased abstraction to handle complexity the concept of a gate has become somewhat blurred. For example, a 32-bit adder can also be viewed as a gate, or super-gate, although it consists of 32 full adders, in turn consisting of several basic gates. So let us say that gate-level means a circuit description built with logic complexity from basic boolean functions up to units implementing more complex but still conceptually simple functions, such as flip-flops, multiplexers and full adders. Now we know what a gate-level simulator should take as input, but what should it be able to do?

We can try to define a gate-level simulator as a simulator *that takes a circuit design at gate-level and evaluates all gates in the circuit so that correct output is produced*.

However, this is not a good definition, since the important thing with a simulator is to be able to correctly view the internal state of a circuit at any point in simulation. Thus, a better definition would be:

> *A simulator that enable the user to correctly, at gate-level and any point in simulated time, view any part of the circuit's internal state.*

This is a better definition, since it defines only how we would like to use the simulator, and not how it should work internally. With this definition the simulator has the opportunity to cheat, and only evaluate gates that change, or only simulate the monitored parts of the circuit at gate-level, the rest of the circuit could be simulated faster on a higher level.

1.6 Computer architecture

Simulation speed depends on the implementation, the algorithms used and the computer architecture. If we want maximum speed they have to be coordinated. We have to use algorithms and coding that is adapted to the target architecture. This mean the properties of the target computer architecture must be known to make an efficient implementation. A brief overview of a typical x86 superscalar architecture is given in Figure 2. We assume the execution units are pipelined. To keep the instruction pool full instructions are prefetched from memory and decoded. When a control transfer (jump, call, ret) instruction appears the prefetcher tries to determine the new address at where to fetch instructions by prediction, often based on branch history statistics kept in a branch prediction table. If the prediction is wrong, all instructions following the jump instruction must be canceled and the correct instructions fetched. This typically costs nearly as many cycles as there are pipeline steps.

Thus, first conclusion is that a fast implementation should avoid jump instructions that are hard to predict correctly.

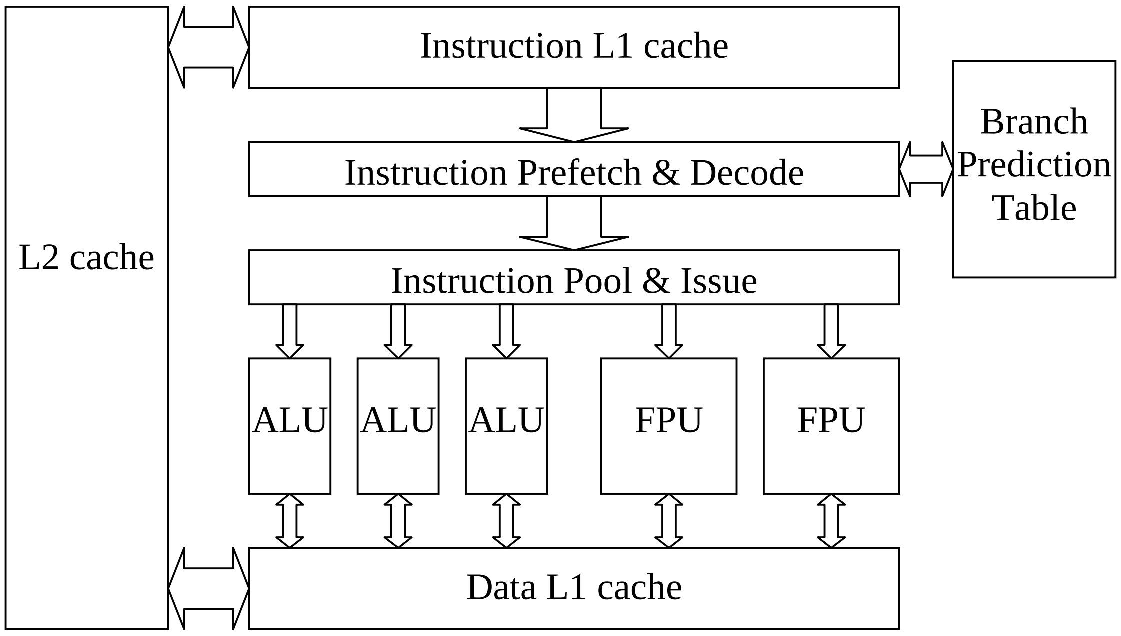


Figure 2 A typical computer architecture

Now, if either instruction or data is not in the L1 cache the execution must wait up to 73 cycles, according to Table 2 in section 1.4. The cache contents are updated based on usage history. So, secondly, a fast implementation either fit completely in L1 cache, or have high locality. And last, in order to use several of the available execution units in each cycle we need low interdependence between instructions.

1.7 Leon

Leon is a freely available implementation of a SPARC v.8 processor from Gaisler Research [Gaisler]. SPARC is a RISC processor architecture formulated by Sun Microsystems in 1985: “SPARC was designed as a target for optimizing compilers and easily pipelined hardware implementations. SPARC implementations provide exceptionally high execution rates and short time-to-market development schedules.” [SPARC]

In [EMIL 2002], Leon was chosen as test input “because it is a good example of hardware that one could want to simulate at gate-level in cooperation with a full system simulator”. In this work we stick to this choice also in order to compare the results against EMIL. We use exactly the same Leon (in the form of an EDIF netlist [EDIF]) to test and benchmark the simulator. For further information on how Leon was configured and synthesized the reader is referred to [EMIL 2002].

2 Related Work

2.1 EMIL

EMIL is an Event-driven Multi-queue Interpretive Levelized gate-level simulator [EMIL 2002]. The simulator engine in EMIL consists of a scheduler, a queue system and a dispatcher. A PERL script creates a specific evaluation function for each gate type from the description in the logic library (asynch_OR3 in Figure 3). The netlist is levelized, and the level is stored with each gate together with pointers to fan-in, fan-out and evaluation function in a gate structure, see Figure 3. The size of this structure varies depending on fan-in and fan-out. During simulation the dispatcher pick gates from the queue system and calls its evaluation function (with the gate structure as argument). The evaluation function in turn calls the scheduler if the gate output changed.

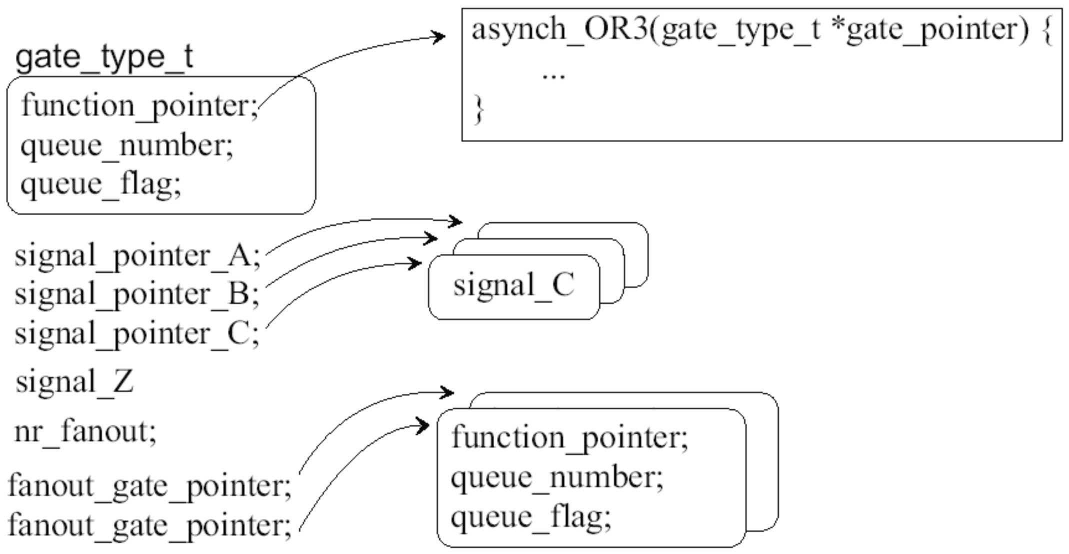


Figure 3 EMIL gate structure for an OR3 gate

This gives fast gate evaluation since each gate has a specific compiled function. It also gives a small code size compared to a traditional LCC simulator since only one instance of each gate function is created, and the number of gate types in the logic library limits the number of functions. However, it can be argued if EMIL really is interpretive, since it actually generate C-code for each gate type and compile it together with the simulator engine.

In EMIL, the problems occurring with synchronous gates when it comes to evaluation order and asynchronous inputs (see section 2.2.3, Gate ordering below) are solved by a special ordered synchronous queue, and a special asynchronous part scheduled and evaluated with the asynchronous gates.

2.2 Simulator concepts

Several simulator concepts have emerged as an effect of previous work in the area, mostly aiming at higher simulation speed but also for more precise simulation (exacter models) or reduced preprocessing time. These can be divided in Time and signal modeling, Algorithm choices, Gate ordering, Implementation strategies, and Netlist representation.

2.2.1 Time and signal modeling

In reality, a digital signal is nothing but an analog voltage pulse. It needs time to rise, to fall, and to propagate from one place to another. How this is described in a simulator is first a question of what resolution one need, which abstraction level to model. At gate level the voltage level is typically described by a discrete set of values, in VHDL {U, X, 0, 1, Z, W, L, H, -}. Modeling of more signal levels or more timing accuracy requires more calculations, and is as a result slower.

For timing there are three methods, nonunit-delay, unit-delay, or zero-delay. Nonunit-delay means that it takes a certain number of time units for a signal to propagate through each gate. How long is determined for each gate. In unit delay this time is fixed to one time unit for all gates, and in zero-delay no delay is used. The delayed models allow us to see glitches and timing issues of signals, but we need to simulate with a higher time resolution than clock cycles; we must simulate each time step any input changed. With zero-delay we can cancel all glitches and simulate a gate only when its inputs are stable for the cycle.

2.2.2 Algorithm choices

Traditionally two types of simulation algorithms exist, *event-driven* and *oblivious*. In event-driven simulation each changed signal generates an event, triggering evaluation of affected gates whose output signal possibly changes, in turn generating new events. Thus only gates involved in signal changes are ever evaluated. Oblivious simulation on the other hand evaluates all gates each cycle. It does not care nor remember (explaining the name oblivious) that nothing changed for a particular gate since last evaluation, but simply evaluate it again. The advantage with this method over event-driven simulation is that no event system is needed, no code determining new gates to schedule after each gate evaluation, and no code managing the scheduled gates. Each gate evaluation can be made small and fast, but we have to evaluate more of those.

The normal way to describe a gate is by giving the boolean function, where to find each input, and where to save the output. But this can be modeled differently. Maurer uses a method of describing gates originally discovered by D. Schuler to create the Inversion Algorithm [Maurer 97].

The method, named counting algorithm, uses the fact that most normal gates can be described by a count and a dominant value (D. Schuler from [Maurer 94]). When no input have the dominant value, the output is 0 (or 1) else it is 1 (or 0). For example, when no input of an AND is 0 (dominant), the output is 1, else 0. Or when no input of a NOR is 1 (dominant), the output is 1, else 0. So the algorithm simply counts the number of dominant inputs. The counting algorithm then evaluates gates each time an input changes:

```
if changed input is dominant then
  increment gate count
  if gate count is 1 then
    output = not output
  end if
else
  decrement gate count
  if gate count is 0 then      (no input dominant)
    output = not output
  end if
end if
```

This method has some advantages, and some drawbacks:

+ All inputs are represented by one count.
+ A single simple evaluation function.
- Requires correct initialization of gate output and count.
- Does not support all types of gates, only normal simple gates where all inputs have the same meaning to the gate.

Another way to go is by branching programs. Based on the boolean function a BDD (Binary Decision Diagram) is created for each output of a gate network, and a branching program is derived from the BDD. The method has the advantage of having a worst-case evaluation complexity proportional to the number of inputs and outputs, but the BDD size grows exponentially in the worst case (although overcome in the general case) [Ashar 95]. Other disadvantages are that only input and output values are known; intermediate nets have no observability, since no gates are simulated, and locality is poor. [Jiang 2003] presents a generalized method of this in the form of a Generalized Cofactoring Diagram (GCD).

2.2.3 Gate ordering

Consider the gates in Figure 4. Clearly we have to evaluate the gates in correct order, in this case A, B, C, D is one such order. If we evaluate A, B, D, C we might get wrong result, or will have to evaluate C again. The common way for a simulator to determine a correct order is by levelization.

Each gate is placed in a level depending on its depth in the circuit. All gates in one level are mutually independent, and can be evaluated in any order. Levelization is done by first assigning level 0 to gates whose inputs are known (connected to circuit in-ports or registers), in this case A and B receives level 0. Then gates whose fan-in gates all have a level are given the next higher level [SSIM 87]. Thus D cannot be given a level since its fan-in gate C doesn't have a level yet, but C is given level 1. Last D can be given level 2.

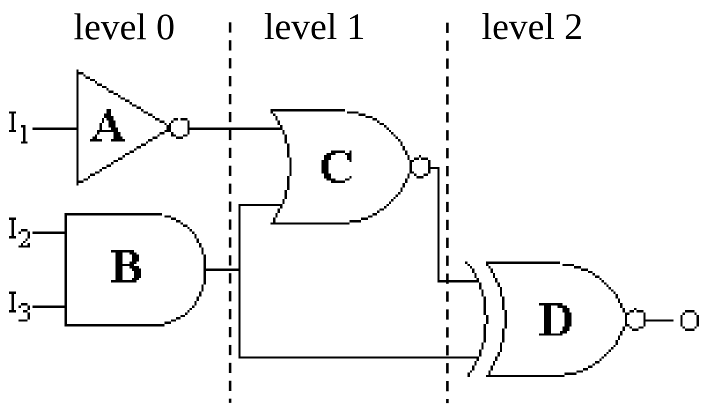


Figure 4 Levelization example

A problem with levelization occurs when a circuit has feedback paths (loops or sometimes also called *strongly connected* gates). A feedback path is characterized by the fact that the output of each gate in the path depends on the output of all other gates in the path. Consider if the output from D was connected to one of the inputs of B, creating an asynchronous loop B, C, D, B. B cannot be given a level since its fan-in gate D have no level and thus C and D cannot be given a level either.

In the work [LECSIM 90], such paths are handled by detecting and group the gates in a single gate. In gate-level most asynchronous loops are already grouped, forming different kinds of flip-flops. Loops including a synchronous gate (synchronous loops) are common, but not really a loop since the synchronous gate breaks the loop until next cycle.

A second problem when it comes to gate ordering is synchronous gates. Since all synchronous gates evaluate simultaneously, they must be evaluated in correct order. Starting evaluation with the first of two consecutive synchronous gates will produce a new output for the input to the second, but it should use the old value; in reality the new value have no time to propagate, since both gates receives the clock signal simultaneously. Starting with the second (last of the consecutive) gate, or inserting a buffer gate between, solves this [EMIL 2002]. Synchronous gates can also have an asynchronous input that has to be levelized and evaluated correctly.

Another ordering possibility, at least for oblivious simulation, would be to search the netlist depth-first from each out-port, something like:

```
mark all design inport nets as storage points
for all design out-ports
  recurse driving gate
end for
```

Gate recursion routine:

```
for each gate fan-in
  if net is marked as storage point then
    output load operation
  else if net have > 1 fan-out then
    recurse driving gate
    output gate evaluation operation
    output store operation
    mark net as storage point
  else
    recurse driving gate
    output gate operation
  end if
end for
```

The output from the algorithm would be an instruction sequence where each gate can read its input from the stack, and save output to stack, hopefully resulting in high locality. Detecting feedback loops would be straightforward, handling them some more trouble.

2.2.4 Implementation strategies

Talking about a programming language, such as C, there are a clear difference between the two concepts of interpreted versus compiled. An interpreted language such as LISP or PERL is executed directly by a program (interpreter), while a compiled language are put through a compiler and executed by hardware. Those two concepts occur also in the area of simulators, a netlist can either be read by the simulator and interpreted, or compiled with a simulation engine to execute directly. A compiled simulator tries to remove as much as possible of the runtime translation from netlist to executed instructions.

Using optimization techniques such as loop unrolling, direct addressing instead of indirect, and threaded code (arrange code segments in execution order to remove jumps, calls, and returns) can make a compiled simulator much faster. We could also imagine an evaluation routine evaluating two (or more) independent gates simultaneously, to increase instruction parallelism, and thereby hopefully use a superscalar architecture more effectively.

Attempts at condition-free simulation have also been made. The Inversion Algorithm [Maurer 94] does this by toggling the gate processing routine between two versions after each call, and later works [Maurer 2000] similarly by taking the address of labels. The immediate thought of this is nice, a function call is a fixed jump, and the processor can correctly predict fix jumps. *But it does not work*. What we get is a register indirect jump, the processing routine address is loaded to a register and the processor is told to jump to the address in that register. And this is a conditional jump, it depend on the address in the register. Further, since the address is toggled after each call, the branch prediction table will always contain the wrong address, and we get a costly miss-prediction *each* call.

Now, does it work to fix this by taking the address of a label and then use goto with that label address? Again, the thought is nice, but we have the same problem, and further, it is not supported in C (GNU extensions to gcc supports it however). Only if the actual goto instruction bits are replaced with other it *might* work, just some issues too many; it requires assembly, it is not portable, it is unreadable, does the processor really predict fix jump from instruction only? And are we allowed to modify the code section by hardware and OS?

Other ways of optimizing by code would be to pack many signals together and do one logic operation on all simultaneously. However, the overhead in packing, unpacking and repacking would be substantial at gate-level. At register transfer level it would be more natural.

Last, implementing an event-driven levelized simulator can use one or several queues to store scheduled gates. One queue is fast in determining the queue, but need to order scheduled gates by level to avoid reevaluations. Several queues (multi-queue) need more operations in determining the queue and determine when all queues are empty, but need no order among gates in the queues. If we distinguish full-queue (as many queues as levels), multi-queue (several queues, but some levels might share queue) and single-queue (only one queue), MICA would be a full-queue simulator.

2.2.5 Netlist representation

A netlist is often described hierarchically; a component is described in detail in one place and then used in several. A physical implementation of the netlist must flatten the netlist, by copying the description of each component to each place the component is used.

A simulator however, can choose to exploit the hierarchy to save space by creating one description of how to simulate the component and use this with different data each time the component occurs in the netlist, much like a programmer can create one function for a common task and then use it in several places with different data. Compared to a flat version this creates some overhead in calling the hierarchical components, but we save space. [Lewis 91]

[Maurer 99] distinguish a hierarchical component used only once as a partition. Such partitions can be used to reduce gate evaluations, and also be created in order to reduce the scheduling overhead in an event-driven simulator. The latter is done by [Blaauw 93] with a clustering algorithm. [DeVane 97] observes that gates that precede and follow synchronous gates need only be evaluated with these, and [Maurer 99] that existing partitions can be switched off some cycles, for example a CPU with both ALU and FPU need only simulate the unit actually used each cycle. (In my opinion the circuit designer should think of disabling such units when not used, since this will also save power, which is an increasing issue due to the heat generated and battery lifetime, but this is a sidetrack.) Partitioning is also used to overcome large BDD sizes in the creation of branching programs [Ashar 95].

2.2.6 Combinations

The different concepts can be combined in several ways. Some combinations are straightforward. Historically the compiled and oblivious concepts did go hand in hand, as did the interpretive and event-driven. The LCC (Levelized Compiled Code) is traditionally considered the fastest type of software simulator, such as [SSIM 87]. As the need for faster simulators increased new combinations where sought. The fast compiled code concept, and the event-driven idea reducing the number of gate evaluations required was combined in several simulators [SLS 88], [COSMOS 87]. [LECSIM 90] also added the levelization technique to this combination to minimize gate reevaluations, while [Lewis 91] has explored the effects of utilizing netlist hierarchy as well as the effects of caches.

3 Approach

3.1 Solution strategies

As we have seen there are a lot of different approaches to simulation, and how to make it fast. To return to the question in section 1.3, how do we simulate a high number of gates fast? Obviously we want to evaluate as few gates as possible with as few instructions as possible, using a minimal data structure to store the netlist. We have some options:

- Use alternate evaluation methods to reduce or simplify gate evaluations.
- Store only the information absolutely needed in simulation, as tight as possible.
- Exploit hierarchy; use the same subcomponent description in several places, but with different data.
- Compile the most frequently used gates to increase speed for those.
- Use algorithms that increase data locality.

Some of these options contradict, i.e. storing only absolutely needed data in a tight way will most likely mean some extra instructions packing and unpacking bits, and compiled gates typically takes more space. However, looking back at Table 2 (section 1.4) we note that we can spend up to 19 instructions avoiding each L1 cache miss, and up to 70 instructions avoiding L2 misses, meaning some extra work involved in reducing data size might be worthwhile.

This can be formally expressed. Assume we evaluate a total of *x* gates. Each evaluation need *g* cycles, and *w* cycles are wasted each time a gate is not in cache. Let *m* be the miss rate, the total percentage of gates not in cache. The total cycles needed for evaluation is now $y = xg + xmw$. Suppose we can achieve the new miss rate $m_{new} = m(1-d)$ where *d* is the percent miss decrease, by adding *i* cycles per evaluation. Then we get $y_{new} = x(g+i) + xm_{new}w$. Of course we want:

$$y_{new} \le y$$

$$x(g+i) + xm_{new}w \le xg + xmw$$

$$i + m_{new}w \le mw$$

$$i \le mw - m(1-d)w$$

$$i \le mwd$$

That is, adding 2 instructions per gate having 20% miss rate and wasting 20 cycles per miss we need to decrease the misses by at least 50%, which would be hard. But say we could move from memory to the L2 cache (50 cycles save) by 30% memory size decrease. This means that adding 3 instructions per gate will be to our benefit if the original miss rate is more than 20%. And if we can do it without adding instructions we can 'save' 3 instructions per gate.

3.2 Netlist representation

This work focuses on exploring the three top options in section 3.1 by using ideas from the counting algorithm described earlier. Maurer uses this method focusing on reducing useless simulations, unpropagated changes, and for unconditional simulation [Maurer 97]. However we are not interested in this, the method of reducing unnecessary evaluations in practice mean we evaluate the gate for each input change, and the unconditional method used actually mean we get lot of register indirect branches.

The feature of the counting algorithm we would like to use is the fact that a gate does not have to know what each and every input value is, or where to find them. Only the amount of high inputs and gate type is enough to determine the output. This mean we get a simple evaluation function, and a compact netlist description. As we shall se later, over 70% of the switching gates are represented by only one standard type. This is achieved by combining the count idea with the traditional evaluation method used in EMIL. This also removes the need for proper initialization of count values and outputs.

To enable use of hierarchy and reduce the amount of information stored in the simulated netlist, we distinguish three kinds of data and separate them in different areas, *management*, *structure* and *data*. The management area stores information useful to find and access individual gates and nets, but not needed for simulation, for example instance names. The structure area stores data that is read only during simulation, e.g. how gates and nets are connected, but not values that are changed during simulation. Due to the nature of the count way of storing inputs this is a one-way structure, gates know the location of their fan-out, but not of their fan-in. Finally, the data area store all dynamic data changed during simulation.

3.3 Main problems

It is hardly a surprise that this solution creates some problems. There are mainly two of them.

1. As mentioned in section 2.2.2, Algorithm choices, only simple gates with independent inputs suit the counting algorithm. Other gates have to be identified and handled differently.
2. Separating gates in structure and data area mean we need two different pointers to each gate, or must have equal sized gates so the same pointer can access both areas. But gates have different output net sizes, from just one or a few fan-outs up to the sizes of the clock or reset network.

3.3.1 Problem 1, incompatible gates

We can create a special or hierarchical version of the gate, either compiled in same manner as EMIL does, or described with simpler count compatible gates and using our standard count evaluation. But this involves extra work storing and passing inputs to and from the special or hierarchical gate, and more choices when deciding what evaluation function to use.

If such gates instead are decomposed to several simpler gates supporting the counting algorithm, and directly replaced in the netlist we might still benefit, despite more gates. Since we get several simpler gates, instead of one complex, we expect only some parts of the complex gate to be evaluated when an input change, thus reducing gate activity.

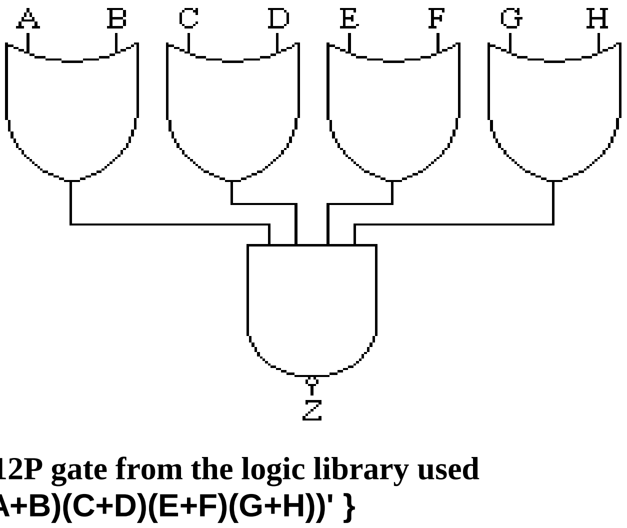


Figure 5 The AO12P gate from the logic library used
{ Z = ((A+B)(C+D)(E+F)(G+H))' }

The AO12P gate for example, viewed with simpler gates in Figure 5, has eight inputs and a large boolean function. With a special or hierarchical version all inputs must be found and the entire function evaluated as soon as one input change. But if we decompose this gate to five count compatible gates (one for each OR and one for the final NAND as the figure show), we only need to evaluate one OR and possibly the NAND as a result of one input changing. So decompose the gate seems more promising.

Still, this solution result in some really bad cases, and for some synchronous gates this is still not enough. In short, synchronous gates are represented with one standard synchronous gate and a front function to get special behavior, but this is described more in section 4.3.2, Synchronous gates.

Here we will instead mention one of the gates really bad to decompose: the two-to-one multiplexer. There are four special properties of this gate making it a bad case. First, it is the third most common gate in Leon, and because it is part of both data and control path it is likely to change often. Second, it splits to as many as four gates. Third, a change on the select signal activates all of these splits. And fourth, the function is really simple, a special routine could evaluate the gate with one if-statement. Although there was not time to implement and investigate the effect of a special type this is really needed, as we will see later.

3.3.2 Problem 2, nets are not equal sized

Using two pointers for one gate is both space consuming and inconvenient, using fixed size gates is much more appealing. The problem is then to represent the fan-out pointers. One way to go would be to have one list with the first fan-out of each gate, one with the second etc, and sort the gates by number of fan-outs. The fan-outs can then be accessed in turn using the same offset into the fan-out lists as the gate offset in the structure and data area. If it points outside the list no more fan-outs exist for that gate. This would not waste any memory, but would be bad from a cache point of view; fan-outs would be scattered in memory. It also imposes a fix order, but we might want to order gates to reduce the distance in memory to its fan-outs (enabling smaller fan-out pointers), or order so frequently used gates are clustered in memory (to increase locality).

Each gate can also store a pointer to a fan-out list. But according to statistics most gates have only one fan-out, and storing this directly would be more efficient. This lead us to the solution of storing a fix number of fan-outs in each gate, and use a special gate type for gates with many fan-outs, storing a pointer to a fan-out list.

The separation of structure and data in fixed size gates, as well as using the count evaluation method, also mean a restriction of only one output per gate. To represent gates with multiple outputs we have identical choices as in Problem 1, and we solve it by creating one gate per original output.

4 Implementation

4.1 Overview

Figure 6 gives an overview of how MICA works. The ovals describe actions taken by MICA, and the rectangular boxes depict data used as input to and output from those actions. The arrows indicate which data is used as input and what we get as result. The darker actions (ovals) indicate parts that were reused from the EMIL implementation.

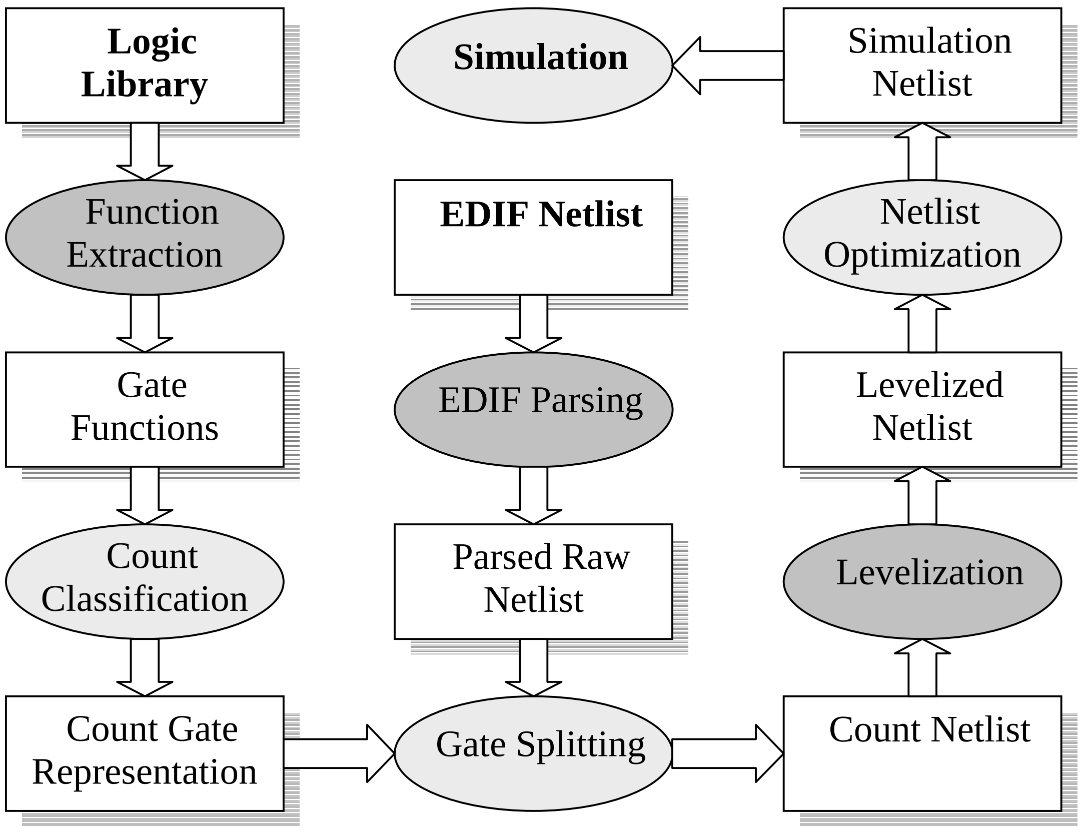


Figure 6 Implementation overview

The following sections will describe most of these actions and some of the data formats used in more detail. The input to the simulator is an EDIF netlist and a Logic Library describing the gates used by the netlist. The gate functions are extracted from the Logic Library and classified into count types. This classification is added to the parsed netlist, possibly splitting some gates, yielding a count compatible netlist. Queue and priority is then added for each gate by levelization. Finally the netlist is optimized to the MICA internal simulation format, and ready to simulate.

4.2 Function Extraction

The Logic Library contains more information than needed for our purpose. In EMIL a PERL script was used to extract the boolean function of each gate and create its C-code evaluation function. This script was stripped and slightly modified to instead create a text file describing only the important properties of each gate. The simple grammar describing each gate is given in Appendix B, Grammar.

Synchronous gates cannot yet be extracted correctly, and have to be manually described in the text file. Other gates can easily be hand tuned if desired, for example to decrease the number of gates created when splitting, but this was not done, since the automatic splitting produce best possible result in most cases (exception is large multiplexers).

4.3 Count Classification

4.3.1 Asynchronous gates

The counting algorithm uses the fact that most gates have either high or low output iff (if and only if) exactly none or all of the inputs are high. AND is for example high iff exactly all inputs are high, NOR iff exactly none of the inputs are high. We need to determine how each gate type can be described in this way.

Assume the inputs are represented by a count value telling how many inputs are high. When an input rises (low to high) we increment this count, and when an input changes from high to low we decrement. Depending on gate we let the exact situation when output is known be represented with the count value zero, and call the output in this situation OaZ (Output at Zero), i.e. for a three input AND we let the count value reach zero iff all inputs are high, and set OaZ to high. This mean the count value of our AND must start at –3 when no input is high. A two-input NOR will also have OaZ high, but start with count equal to 0 for no input high (and reach count 2 when all inputs are high). We call the gates that can be described like this *AtZero* gates (they switch at zero count).

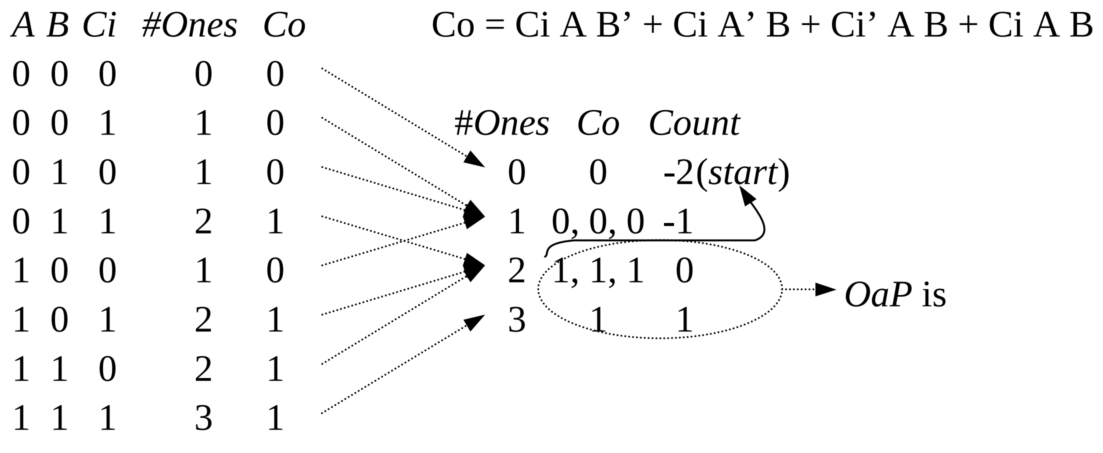
Similarly we can derive *AtEven* and *AtPos* gates (output is OaE when the count value is even, or OaP when count is positive). Those two types correspond, respectively, to the sum and carry function of a full adder gate. In all, we can distinguish three properties describing a gate, the *count type*, the *output at* value, and the count *start*.

Classification of gates to one of the AtZero, AtPos or AtEven types is of course done automatically. The boolean function of each gate output is read from the text-file that was extracted from the logic library, and evaluated for each possible input combination (since basic gates are small this is not a problem, but it could become for gates with many inputs).

The result of each input combination is stored in an array in the slot determined by the number of high inputs. If different output value occupies the same slot we know that the gate inputs cannot be represented by a count. Next, the output array is investigated. If output high (or low) exist in only one of the slots (with all other slots the opposite value) we have an AtZero gate. If only one switch from high to low (or vice verse) is found when we search the output array, an AtPos gate is found, and if it switches between every value we have an AtEven gate.

If a gate cannot be represented by a count its boolean function is split to parts that can. This is done by a depth first traversal of the binary expression tree built for evaluation of the gate. Each node in the tree has the function of OR, AND, or NOT, with each leave as one input value. The tree is then split at each point where the children of OR and AND nodes is not of same type as its parent node. Some optimization is done to transform (A'+B') sub-trees to (AB)' and A'B' sub-trees to (A+B)', as this creates one instead of three splits. Now each gate is described by one or more output functions, each with a *count type*, *output at* value and *start* value.

Let us investigate a carry function as an example, viewed in Figure 7. The function is evaluated for each input combination. Three different combinations with only one high input exist, and three combinations with two high inputs, but with one high input the output (*Co*) is always low, and with two always high, so no conflict occurs when joining the rows with the same *#Ones* count. After joining these rows we have output high in two cases and low in two cases, but only one switch point exist. Thus we have an *AtPos* gate. Now we bias the *#Ones* to start at zero at the switch point to get the *start* count value (–2), and the *output at* value (high). If we instead imagine a three-input AND gate (Co = Ci A B) the output would of course be high only when *#Ones* equal 3, giving us an *AtZero* gate, and the *Count* would start at –3.



A	*B*	*Ci*	*#Ones*	*Co*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	2	1
1	0	0	1	0
1	0	1	2	1
1	1	0	2	1
1	1	1	3	1

Co = Ci A B' + Ci A' B + Ci' A B + Ci A B

#Ones	*Co*	*Count*
0	0	-2(*start*)
1	0, 0, 0	-1
2	1, 1, 1	0
3	1	1

Figure 7 Classification example

4.3.2 Synchronous gates

Synchronous gates are special in several ways. First they should only evaluate on rising or falling clock edge, second they should evaluate all at once (if connected to matching clock), third some have synchronous or asynchronous reset, preset or both reset and preset, and at last they often have an extra inverted output. Clearly not all of this is compatible with the counting algorithm scheme used for asynchronous gates. Most important, the counting algorithm does not know which input is which, and thus cannot distinguish between data, reset or preset ports.

With only a data input (a simple D-FF) we have nothing but an AtZero gate (with OaZ low) updated only at either rising or falling clock edge. The extra inverted output could be created by simply adding a sibling flip-flop using OaZ high. However, that would also double the schedules and evaluations of synchronous gates, and still not handle reset and/or preset inputs.

Instead, we let the synchronous gate be a double-gate, two consecutive gates where the first (master) hold the main gate type, the data input and the non-inverted output, and the second (slave) the inverted output and preset or reset input. The slave gate is given a special type to instruct the simulator to either evaluate the master gate directly (asynchronous reset/preset) or only make sure the master is scheduled (synchronous reset/preset), while the master gate takes care of the evaluation and fan-out update of both. (Slave gate match the asynchronous part in EMIL.)

Now all synchronous gates are reformulated to use this ‘generic’ double-gate. For example, a JK flip-flop is built with the JK function as an asynchronous part in front:

JK front function:	D = J'K'QN' + J K' + J K QN
generic doublegate flip-flop:	QN = (Q = D)' :on CP

When it comes to the timing issues, MICA can handle those gates in same fashion as EMIL with a special ordered synchronous gate queue, or by inserting buffers between consecutive synchronous gates.

4.4 Gate Splitting

Here, the count classification is joined with the netlist (split of boolean function is already done in the classification step). For each gate in the netlist the classification of that gate type is examined. If the classification consists of several output functions, new gates are created with inputs and output according to the boolean function of that output. The new gates then replace the old gate in the netlist.

In this step we also add buffer gates as necessary to handle large fan-outs, and between synchronous gates if we want the faster synchronous evaluation switch.

4.5 Levelization

This is a technique to group gates independent of each other together, see section 2.2.3, Gate ordering. MICA reuses the levelization from EMIL. Inputs and synchronous gates are given a start level. Then gates whose fan-in gates all have a level are given the level next higher than the level of the highest fan-in gate, until all gates have a level assigned.

4.6 Netlist Optimization

This step creates the data structures actually used during simulation, removing all information not needed for simulation. The levelized netlist is kept with pointers from each gate to corresponding gate in the simulation netlist. Thus, we can still easily access the netlist by gate and net names in order to get observability.

The first step in this process is to determine the position of each gate in the simulation netlist, and the amount of memory needed. Currently the gates are simply ordered as they appear in the netlist, in the future however, other orders might be considered, see section 7, Future Optimizations. Once the position of each gate is decided the structure area is created, for each gate adding count type, output at, level and position of fan-out gates. Last, the data area is created, initiating and scheduling every gate. The initialization is done for each gate by setting the old output to low (also for complementary outputs), and storing the start value for the gate type as gate count (since all old outputs are set low, there are no high inputs, and the start value applies). As a final step an initialization half-cycle is run (using standard simulation functions), making all outputs consistent (making complementary outputs complementary, and outputs from logic_1[1] high). Strictly, this is nothing but the first simulation cycle, but it is needed to get in synch with EMIL.

4.7 Simulation Netlist

The netlist used during simulation consist of two parts, one read only structure area and one data area containing all values that change during simulation. A third management part is used to be able to access the

[1] Logic_1 and logic_0 are special gates used to store the constant values 1 and 0 in the netlist. They do not have any inputs, nor do they change during simulation. They have their own FixAt count type.

simulation netlist by gate or net names, but this will not be described. A gate is represented by an offset used to index into both areas as viewed in Figure 8 below.

This separation of the netlist into structure, describing fixed properties such as how gates are connected, and data holding dynamic things such as output values allow us to exploit hierarchy. Several independent data areas can share the same structure area. For example, a netlist describing a processor might contain several integer units. The implementation of this can then be described with one structure area, but each instance get its own data. This way memory is saved, but the drawback is that structure and data is separated in memory, reducing the data locality.

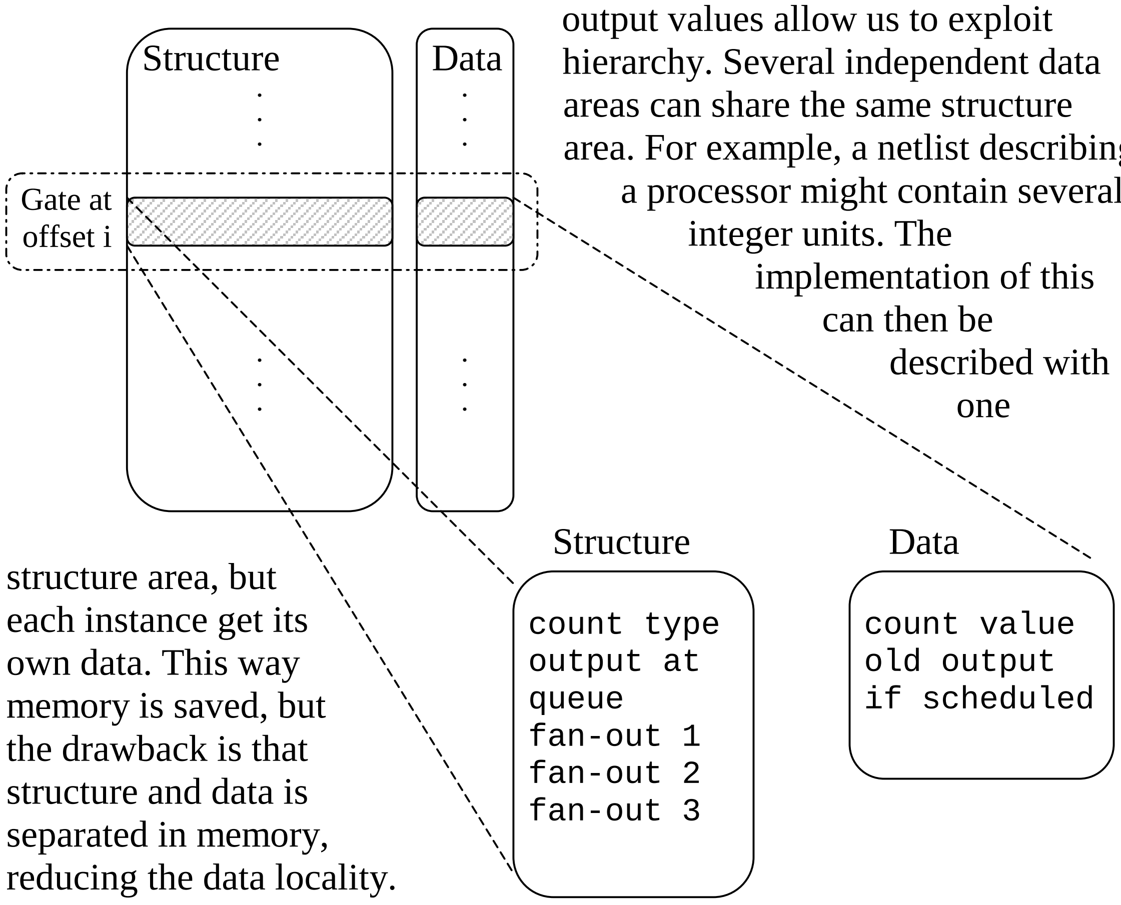


Figure 8 Simulation netlist

Figure 8 show the properties stored in respective area for each normal gate and the net it drives[1]. For each gate in the structure area the *count type* tells which evaluation method to use (AtZero etc.), *output at* contain the output when the condition determined by count type is true, *queue* hold the queue this gate should be scheduled in and last we have slots to store the offset to fan-outs. The data area contain how many inputs to the gate are high in the *count value* (but with zero bias depending on original gate type), the old output, and if the gate is scheduled. In the figure we see the advantage of

[1] To avoid confusion some clarification is in order. In MICA a gate is restricted to have only one output. The net driven by this output is stored together with the gate as fan-outs and old output. Thus the storage space allocated to a gate can equally correct be referred to as a net. Although this document tries to refer to it as a gate, the term net is more natural in some cases.

the counting algorithm, each gate can be stored without knowledge of its fan-in, and without knowing which input a given fan-out is connected to, only knowing the offset to each fan-out gate is enough. This can be compared to EMIL, who had to store both fan-in and fan-out pointers. Another difference implied by the count way of storing gates is that when each gate is evaluated it updates each of its fan-out gate counts while it have to access them for scheduling anyway. EMIL did the other way around, each gate had to both fetch the input values from fan-in gates before evaluation and access the fan-out gates for scheduling.

The current implementation uses eight bytes structure per gate and one byte data, total nine bytes per gate. The goal was to reach four bytes structure, but there was not enough time to investigate this. Six bytes can be easily achieved, but is not tested. Using two, four, or eight bytes has the advantage that the gate offset can be easily scaled in hardware address calculation, and it divides one cache line evenly.

4.8 Simulation

4.8.1 Gate evaluation

The simulator uses the counting method as described earlier to calculate each gate output. This mean we will have only a few gate types of which only one, AtZero, stand for the majority of the evaluations. To decide which evaluation function to use for each gate we need a main simulator switch. This might seem really bad, with a potential branch miss-prediction for each gate, but since 73% of the evaluations are from the AtZero gate type the processor will be able to do a good branch prediction. For the asynchronous gates and queues the evaluation loop to reach a stable state looks as follow:

```
loop forever
  pop gate from queue
  case AtZero
    evaluate AtZero gate
  case AtEven
    evaluate AtEven gate
  case AtPos
    evaluate AtEven gate
  case UserDefined
    call supplied function
  case QueueSwitch
    if scheduled asynchrounous gates left
      switch queue
    else
      break
end loop
```

For readability only the main types are included in the switch. The corresponding switch for the synchronous gates exist in two versions. One examines all synchronous gates in a predefined order just like EMIL, and evaluates the changed ones. The other, which is faster, assumes no consecutive synchronous gates exist and has the same structure as the asynchronous version above. The only difference is the queue-switch (we should evaluate only the synchronous queue), and the choices in the switch.

Evaluation of the dominating AtZero type (this can be implemented mostly unconditional) is described by this pseudo code:

```
if count value equals zero then
  new output = output at
else
  new output = not output at
end if
change = new output – old output
old output = new output
if change not equals zero then
  for all fan-outs
    fan-out count += change
    schedule fan-out
  end for
end if
```

First we determine the new output depending on the *count value* and *output at* value (see Figure 8 above). Then we determine if the output changed. If it rose the fan-outs should be incremented, and if it fell we must decrement. Last we store the new output and apply the change (if any) to each fan-out. Only the first condition differs in the other count types, AtPos and AtEven.

4.8.2 Main simulation loop

Perhaps recognized from EMIL, we finally give the main simulation loop:

```
for each half-cycle
  clock = not clock
  interface Leon ports
  evaluate asynchrounous gates
  if clock is high then
    evaluate synchrounous rising edge gates
  else
    (no falling edge gates)
  end if
  evaluate asynchrounous gates
end for
```

First, assume we are in a stable state, meaning all values have propagated to a register input or circuit port. Nothing change, everything is waiting for circuit in-ports to change. That tells it is a good time to look at the out ports, and update the in-ports.

In the case of Leon we update the data bus according to Leons wishes expressed on all out-ports (address bus etc.), or we can reset Leon by changing the reset port. As the last port interaction we let the clock tick a half cycle and update Leons clock port. After this some asynchronous gates depending on the updated ports need to update (they possibly affect inputs to some synchronous gates), so we must evaluate them. Now all synchronous gates have correct inputs and depending on whether the clock rose or fell those are updated (Leon has only rising edge flip-flops).

The synchronous gates will in turn change the data on the fan-out inputs, and careful here, so we call the evaluation of asynchronous gates a second time. Again we have reached a stable state, all circuit in-port changes has been processed, and we are waiting for new changes (most likely the clock).

Now, what was that about careful? As said before the synchronous gates have to be evaluated in correct order. The synchronous gates update all at once, so output changes from one does not have time to affect the inputs of a consecutive synchronous gate, immediately following the first. But if we update the second after the first precisely that will happen, that the inputs might have changed. This does not occur when an asynchronous gate is placed between the synchronous gates, since the evaluation of this one occurs after all synchronous gates.

A potential problem with the synchronous scheme used here and in EMIL is that at the same once as synchronous gates update, also latches (handled with the asynchronous gates) that are just enabled update, potentially causing same order problem for latches sooner or later affecting a synchronous (or latch) input. Luckily Leon latches is enable low, and synchronous gates rising edge, meaning latches always are disabled when synchronous gates change, and we can ignore this potential problem for now.

4.8.3 Queue system

Gate queues are handled in the simplest manner. Each queue receives a memory area fitting all gates in that queue, implemented as a stack with the bottom at the beginning of the memory area. A special queue-switching gate is placed at stack bottom (much like queue trailer in [Maurer 97]), automatically switching to next queue when the current is empty. A scheduled gate count is used to keep track of scheduled gates. This has the benefit of not having to check all queues for gates every cycle, and it handles gate loops (gates scheduled in a queue already evaluated) by simply wrap around to first queue if there are gates left to evaluate when last queue is finished. The drawback is extra increment and decrement instructions for *each* evaluated gate (the cheaper way would be to identify all loop gates and create a special exception type for those). MICA has 61 queues, evaluated twice every half cycle, meaning checking 12 million queues for gates if all queues have to be checked, with the gate count only 4.5 million is needed.

4.9 Test setup

The goal of this work was to be able to simulate large netlists reasonably fast. To know how the simulator performs with a large netlist we have to load such a netlist in the simulator and actually test. Only reasoning as of what to expect, as done with EMIL above, might give inaccurate result.

Here we face some problems, the first is that we have no large freely available netlist, next is that loading such large netlist would take too long time (reading the netlist fast was not the goal of neither this work nor EMIL, thus the slow (double linked list) EMIL code was reused). Third, it would require much extra work to synthesize a new netlist, adapt special components, and write test programs etc.

All of those problems, as well as letting us compare to MICA against EMIL, were solved by instantiating the Leon netlist up to 256 times, and simulate instance by instance cycle by cycle. Since MICA was implemented with hierarchical netlists in mind this showed to be easy. (Hierarchical support is not entirely completed or tested since Leon has no shared hierarchical instances, and the old netlist loading code flattens the netlist.)

The solution resulted in a large netlist consisting of up to 256 Leon data instances sharing the same structure. Note that in the case of more than a couple of Leon instances it is probably, in comparison to a real million-gate netlist, not realistic that the degree of shared structure is this high. When examining the result we must keep this in mind. Each Leon was connected to its own memory and prom, the latter loaded with one out of three slightly different matrix addition programs. Thus each instance of Leon was run independent of the others. For the purpose of simulating several Leon instances the main simulator loop was slightly modified:

```
for each halfcycle
  clock = not clock
  for each instance
    interface Leon ports
    evaluate asynchrounous gates
    if clock is high then
      evaluate synchrounous rising edge gates
    else
      (no falling edge gates)
    end if
    evaluate asynchrounous gates
  end for
end for
```

As shown above we simulate as if it was one large netlist possibly interchanging data between different hierarchical instances each cycle. Simulate the first instance all cycles, then switching to the next instance and simulate all cycles again etc. (i.e. exchange the two outer loops) would have been possible in our case since no data exchange took place, but would not have been of practical interest since it would give unrealistic good cache behavior (it would be equal to multiplying the one instance execution time).

To conclude, we simulated several instances of a gate-level netlist implementing the SPARC v8 processor Leon, each Leon independently running a matrix addition program.

4.10 Verification

Of course we have to be sure the simulator is correct, i.e. that the loaded netlist has exactly the same state (viewed from gate level) as the corresponding physical implementation in each stable state. Otherwise the simulator would be useless. This comparison was done against the previous simulator EMIL. However, due to the old implementation of the netlist in EMIL, which was reused in this work, finding the logic value of a net is slow. So this check was not done in each stable state, but only at sparse points and last simulated cycle, based on the assumption that most errors would remain and multiply in the next cycle (which do tend to be true from debug experiences). Also, an exhaustive check would only be useful if we are sure the reference simulator is correct.

The result of the program running on the simulated netlist Leon was also checked to be correct (though this does *not* guarantee the correctness of the simulator).

5 Result

MICA was tested with different number of Leon instances and 50000 cycles with results listed in Table 3 and Figure 9. The table lists in turn number of Leon instances simulated, total simulation time for all instances, total memory, total memory times gate activity, execution time for each Leon instance, memory used per Leon instance, and finally how many bytes each gate uses on average.

Leon Instances	*Time (mm:ss)*	*Memory (KB)*	*Critical Mem (KB)*	*Time / Inst. (s)*	*Mem / Inst. (KB)*	*Mem / Gate (Bytes)*
1	00:10	242	28.4	10.0	242.1	11.1
2	00:20	267	31.4	10.0	133.7	6.1
4	00:42	318	37.3	10.5	79.5	3.6
8	01:37	419	49.2	12.1	52.4	2.4
16	03:23	621	72.9	12.7	38.8	1.8
32	06:54	1026	120.4	12.9	32.0	1.5
64	13:48	1834	215.3	12.9	28.7	1.3
128	27:36	3452	405.2	12.9	27.0	1.2
256	55:10	6686	785.0	12.9	26.1	1.2

Table 3 Benchmarks

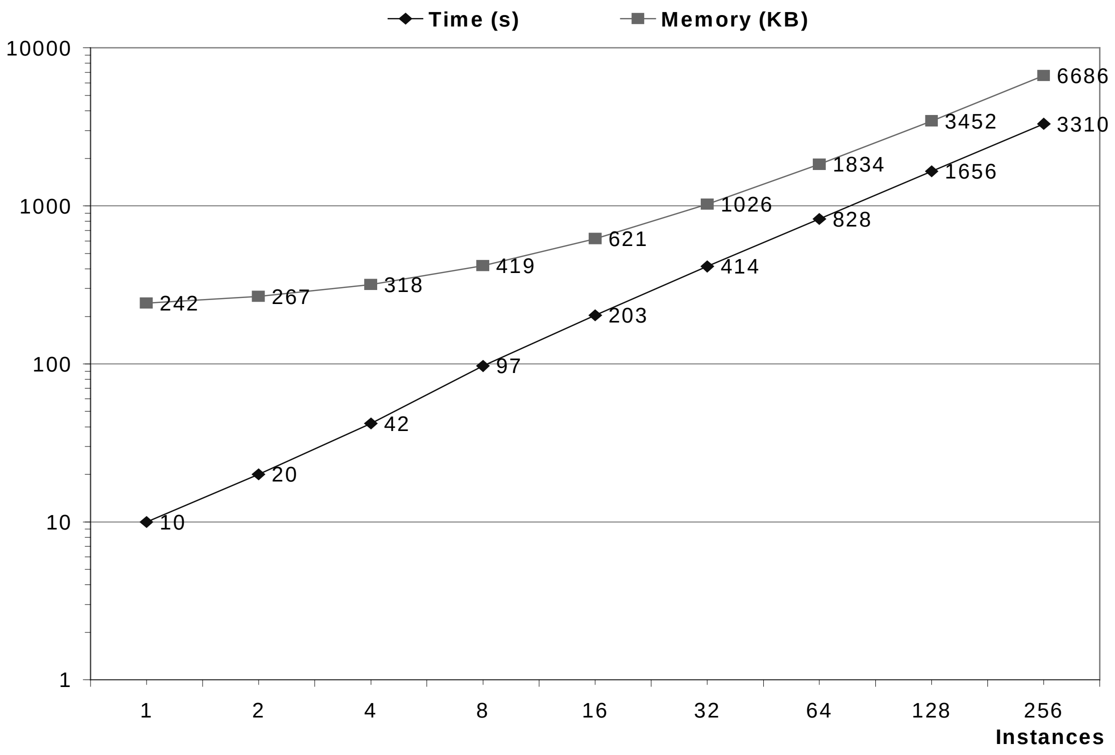


Figure 9 Execution time

The benchmarks was run on an Athlon Thunderbird 1466MHz (64+64KB exclusive L1 cache, 256KB L2 cache, and 512MB RAM), compiled with gcc 3.2.2 –O3 –march=athlon. Old EMIL was also rerun on an identical configuration, 50000 cycles took 7.25s (from Table 4 on page 35).

For the purpose of comparison, and since a buffered solution can be implemented in EMIL as well, MICA was run with ordered synchronous gates (the order problem was explained in section 2.2.3, Gate ordering). Using the buffered solution gave an 11% performance increase (8.9s). 26 buffer gates had to be added after gate split, causing a total of only 30 extra evaluations (in EMIL we must expect much more extra buffers (~800) and evaluations).

The graph in Figure 9 shows the execution time and memory requirements from Table 3. We use a logarithmic scale on both axes to get a unified distance between the plots (to avoid all plots ending up in lower left corner). The memory requirements behave as we can expect from the implementation. With few instances structure memory dominates, whereas data dominates with many instances. Looking at the execution time it seems really good, increasing only linearly with the number of Leon instances. However, looking closer, things are a little worse.

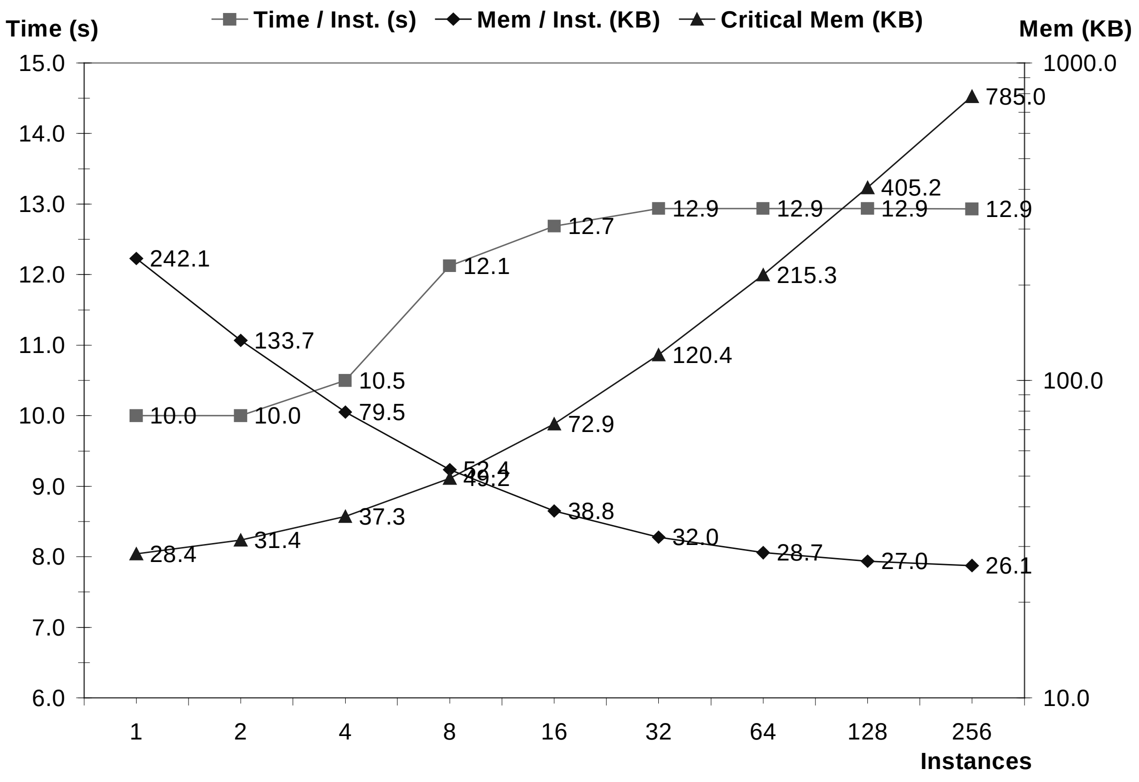


Figure 10 Per instance graph

In Figure 10 the graph over memory and execution time per instance shows us we actually have a 30% performance decrease going from approximately 4 to 16 instances. What we see can be interpreted as the move from utilizing mostly L1 cache to utilizing the L2 cache. With up to four instances most frequently used gates fit in the 64KB L1 data cache, but with 16 to 32 instances this is not enough. This interpretation would expect a similar performance decrease when reaching above 128 instances, when not even the L2 cache is enough. Amazingly this does not happen, and the explanation is probably that with several Leon instances, the structure area constantly occupies the L1 cache, while the data areas are thrashed back and forth between main memory and the L2 cache.

This is an effect of the fact that the memory demand with many Leon instances is strongly idealized, sharing one structure for all instances is growing unrealistic the more instances we have. Thus, a real million-gate design most likely needs more memory for structure area, increasing memory requirement and decreasing performance by more thrashing in caches and memory.

The measure of most frequently used memory (critical memory) can be discussed. It is not likely that it is the same subset of the gates that switch each cycle. But it is a simple way to get a rough hint. From the graph we can se that the performance degradation occurs in the interval from 30 to 90KB, just around the L1 data cache size.

Using the statistics to do some calculations with a more reasonable structure sharing of three to one, we find that 3194KB memory would be required already at 32 instances. That mean a critical size of 374K and more cannot be expected to give good cache behavior. Thus, we cannot expect the speed of 117M/12.9=9M gates per second with more than ~275000 gates (~635000 count gates).

The observant reader might already have noticed that MICA actually is 38% slower than EMIL simulating one Leon 50000 cycles. Is this all a big failure then? Well, not really. The main purpose was to simulate large designs fast, and from the discussion of what to expect from EMIL with many gates, MICA would be faster at this task. (Of course it would be very interesting to actually test EMIL's performance on a large design, or on several Leon instances, but due to the lack of large freely available designs and the implementation of EMIL this would be too cumbersome, and the discussion have to suffice.)

What we can do is to look at the statistics from EMIL and MICA in Table 4 and try to explain why MICA is slower, and how to overcome this.

	EMIL			MICA		
	Total	*Each cycle*	*Each second*	*Total*	*Each cycle*	*Each second*
Execution time	7.25s	145ns	-	10.00s	200ns	-
Simulated cycles	50000	-	6897	50000	-	5000
Gate evaluations	64M	1278	8.8M	117M	2331	11.7M
Synchronous	4.0M	79	548K	3.4M	69	343K
Queue switches	-	-	-	4.5M	-	-
Schedules[1]	22M	447	3.1M	131M	2614	13.1M
Dummy schedules	-	-	-	13M	266	1.3M
Memory used[2]	348KB			242KB		
Gate activity	14.8%			11.7%		
Critical memory[3]	52KB			28KB		
Total gates	8637			19856		
Total nets	10858			21896		

[1] *Calculated differently in MICA*

[2] *Not including 11328 bytes for Leon internal registers and cache*

[3] *Memory times gate activity as a simple approximation of frequently used memory*

Table 4 Comparison

The first we note is the huge difference in schedules and evaluations. For schedules this is partly because they are calculated differently. EMIL counts calls to a schedule routine that schedules all fan-out for a gate (one call per changed out signal). MICA counts each gate actually put in evaluation queue. Dummy schedules occur in MICA when the gate to schedule is already scheduled (equal to the number of gates with multiple changed inputs). The other reason, valid for both schedules and evaluations, is that MICA has more gates to evaluate due to split gates. Something interesting is that MICA evaluates 11.7 million gates per second wile EMIL only evaluate 8.8 million, thus MICA is 33% faster per gate but still 38% slower in total. The explanation is also here to be found in the total number of gates. MICA requires count compatible gates with only one output per gate, meaning a lot of gates have to be split to simpler versions.

Despite this large number of gates MICA is much more memory efficient. MICA requires 30% less memory for one instance of Leon. Thus, the counting method of storing and evaluating gates are very memory efficient. The main reason for this is that all fan-in pointers can be removed, saving about 50% memory per gate.

To see where all extra gates come from we have to look in Appendix A, Statistics. Since it is the new nets that cause evaluations not done before[1], we look at which nets that are added rather than gates. Because gates and nets primarily have a one-to-one relationship added nets closely match added gates anyway. (Only pads, synchronous gates and some adders have several output nets.) Originally there are 10858 nets in Leon. 10584 are added because of split count incompatible gates. 454 are added to buffer gates with huge nets (since we need AtZero gates to hold huge nets). The statistics include only the gate types out of the 84 used that need special treatment. Toggle statistics from EMIL is also included together with a worst-case estimation of how many extra evaluations the split gate would cause for a single input change. (Schedule statistics show that only 10% of the gates had more than one input change, but we must remember this is collected in MICA, after some gates with many inputs are split.) Adding the total sum of estimated extra evaluations with the total evaluations in EMIL give us a total sum of $64M + 67M = 131M$ evaluations, slightly more than those of MICA. It is thus reasonable (and disappointing) to say that the worst case is generally what we get. Early estimations was to optimistic, and the assumption that a change in one input to a complex gate would not lead to the evaluation of all parts in the split version, leading to reduced gate activity, had only little effect.

But looking at the statistics we also see that the main problem is FDS2L and the small multiplexers (discussed in section 3.3.1, Problem 1). Together they stand for over half of the extra evaluations. Creating special types for those is expected to pay off. Some late experiments indicated 11M less evaluations if the FDS2L gate uses a special type.

---

[1] It is the nets that are given the results of gate evaluations. The calculation of net values must have been done in some way for all nets that existed in EMIL simulation. New gates storing the result to old nets just move this calculation to the new gate. Thus it is mainly the new nets that create work in MICA that was not done in EMIL.

6 Conclusion

Simulation is an important part of digital circuit development and verification. But hardware simulators are expensive and software simulation is slow on today's large designs.

We have implemented MICA to simulate large designs. One major part of this work has been to reduce the memory required for such design, in order to use the faster cache memory in a higher degree. The Count Algorithm used has shown to be very space efficient, by not storing fan-in information. We use 30% less memory than the reference simulator EMIL, and according to new statistics from MICA this can be further reduced by as much as 40%, see Reduce structure size further in section 7.

This reduction is despite the major drawback of the algorithm, that incompatible gates have to be split or specially handled. The MICA simulation netlist contain twice as many nets as the original netlist. This is also the reason that the reference simulator is in total 28% faster than MICA, at least for small netlists. Seen from evaluation speed MICA is 33% faster, evaluating 12M gates per second compared 9M per second for EMIL. We note that the overall speed of MICA depends on the Logic Library. When using a library containing only simple gates MICA would be faster also in total.

To emulate a netlist with a high number of gates we simulated up to 256 instances of our testing netlist. This corresponds to 2.2M gates (5.1M gates in MICA) and was simulated at 9M gates per second, which is the same speed as EMIL despite a much larger netlist. However, due to unrealistic hierarchy sharing, we can't expect this speed with more than 250000-300000 gates (0.6M in MICA). A larger cache would raise this limit.

From simulator perspective this is reasonable fast, on average 163 processor cycles are used per evaluated gate. From a user perspective however, the speed attained is very slow. A real circuit can be clocked in MHz, while we in simulation of 300000 gates can measure this in Hz only. With this slowdown simulation of one second circuit time would take 12 days. If we need to simulate the circuit for more than some hundred thousands cycles this is not feasible. It seems software gate-level simulation of huge netlists have a long way to go, and much work remain.

To conclude, MICA shows interesting features when it come to memory requirement and gate evaluation speed. Gates can be stored without fan-in information, and evaluated fast with a standard function. The major drawback is that not all gates are directly compatible, requiring special handling. In section 7, Future Optimizations below we briefly describe some ideas to reduce the penalty of this and other ideas.

7 Future Optimizations

There are many ways to modify MICA in order to increase speed or reduce memory needs. Some of these was intended to be part of this work, but was canceled due to lack of time. Those improvements are instead presented in this section.

7.1 Remove FixAt gates

The Leon netlist has 662 logic_1 gates and 320 logic_0 gates. In MICA those are kept as is. One optimization would be to remove those gates, since they are not needed. In the case of logic_0 gates they can just be removed, and logic_1 should be possible to remove after first incrementing each fan-out gates count start value. This will not affect simulation speed, but reduce the structure and data size by almost eight and one kilobyte respectively. From observability point of view this mean those gates and nets will not exist in the simulator, which might be undesired or confusing.

7.2 Remove unnecessary gates

Some gates in Leon, most notably flip-flops and adders, have two outputs, but in some cases one output is unused. When those gates are split to one count gate for each output the count gate with the unused output can be removed. A total of 428 such unused outputs exist in Leon, but many of those are from flip-flops, and since they are double-gates this do not apply to them. Thus no noticeable improvement is expected for Leon.

Second, all inverters being the only fan-out of a gate can be removed by simply inverting the output at value for the fan-in gate, but we lose observability for the net connecting the inverter and its fan-in gate, since the net is removed. Most such inverters are removed in the synthesis [EMIL 2002].

7.3 Reduce structure size further

The original plan was to use a smart gate ordering to minimize the distance to fan-out gates, and then store the fan-out offsets in each gate relative to its own position. This can enable use of less storage (smaller pointers) per fan-out, and is still to be investigated.

One interesting thing when it comes to size reduction is the fan-out statistics after gates are split. It shows that 77% of all gates have only one fan-out. Since those gates uses only half the space allocated for a gate in structure area almost 40% of the structure area is unused. Moreover, for the most common gate AtZero, this fact is true for 89% (65% of total evaluated gates). A highly optimized one fan-out AtZero type might be worthwhile.

We could for example shrink the standard gate storage size to include only one fan-out, and store larger fan-out special.

What takes space and time is to store and load pointers to inputs and outputs. So we could partition the netlist in small asynchronous networks, for example networks between synchronous gates, and use smaller pointers within each network. This would be best done to reduce input and outputs. Grouping x gates with y inputs to one with xy inputs do not gain very much.

7.4 Special MUX21 and FDS2L types

This is something really needed. Those are very common gates in Leon, and probably any netlist. Despite their very simple function they are split to 4-5 gates. Estimations show that 26% of all evaluations could be removed if each of them could be represented with a single gate. The drawback of adding extra gate types is more choices in the main switch.

7.5 Circular queue system

A simple version of this was implemented in EMIL, but it was not successful due to the way the levelization is done. It tried to exploit that gates most often schedule their fan-out in next level to reduce the number of queues. Here is a vague idea of preparing for this already when levelizing the netlist. If we are able to restructure the levels so no gate have their fan-out further away than say 8 levels, the queue just evaluated could be reused as the queue 8 away from the current. As is, levelization places all synchronous gates in level zero, causing gates with low level to have many fan-outs in all levels. Instead allowing several synchronous levels could increase the level of many gates, decreasing the gap to fan-outs.

7.6 Take care of clock timing

A potential problem regarding evaluation order of synchronous gates and latches were observed in section 4.8.2, Main simulation loop. This should be solved somehow, probably involving some classification of how gate nets depend on different clock events. Two phase clocking is shortly mentioned in [Jennings 91].

7.7 Assemble main evaluation loop

By assembling the main simulation switch loop and the most common types we could probably utilize the CPU registers more efficient, and remove some instructions. Performance would increase at cost of portability.

7.8 Partitions and compiled super-gates

In previous works gates have been partitioned in groups to reduce scheduling overhead, for example [Blaauw 93]. If we manage to partition the netlist into many small asynchronous nets, compilation could be done selectively for partitions with high activity.

Creating a dumb oblivious compiled version of a structure area seems fairly easy. We simply replace the count type and output at value with code for corresponding condition, each fan-outs with an add instruction, and order all code fractions by queue. The data area would not need to change. Interesting is that for such compiled version all inverters can be removed, simply by issuing a sub instruction instead of add when a fan-out points to an inverter. All we need is a flag identifying inverters since the count type do not tell.

We can also think of partitioning all gates whose output later on only affect one specific input to a multiplexer into one cluster. Such cluster would only need evaluation when that specific multiplexer input is selected.

7.9 Order gates by evaluation frequency

Currently gates have no specific order in the simulation netlist, and the locality of event-driven simulation is poor. MICA uses only one byte per gate in the data area, and 8 bytes per gate in the structure area, while a cache line of an Athlon CPU is 64 bytes (32 bytes for a Pentium III CPU) [AMD]. This means seldom evaluated gates will waste much of each cache line. By placing gates in order of evaluation frequency the locality could be increased to utilize the cache more effective.

7.10 Hardware support

Calculating the new output of a gate is a rather simple operation, and so is the update and schedule operation of a single fan-out. Nevertheless those actions need quite a few assembler instructions each. Not that it will ever happen, but just for the interesting thought we could imagine if we had hardware support to do these operations in one clock cycle or instruction each.

8 Explanations

ASIC	Application Specific Integrated Circuit.
ASIP	Application Specific Instruction set Processor.
co-design	Simultaneous design of hardware and software to speed up time to market.
critical size	Dumb estimation of most frequently used memory by the product of gate activity and total memory.
EDIF	Electronic Design Interchange Format.
fan-in	The gates connected to a gates inputs.
fan-out	The gates connected to a gates output.
gate activity	The average percentage of gates changed during one cycle.
gcc	GNU Compiler Collection.
half-cycle	The time during witch the clock is high or low, that is, the time from rising to falling clock edge, or from falling to rising.
IP-package	Intelligent Property package, circuits can be delivered as nothing but a soft description in some language or file format.
LCC	Levelized Compiled Code.
level	Maximum number of gates to furthest preceding circuit port or synchronous gate.
levelization	Method to determine each gates level.
logic library	A collection of gate types and a functional description for each type.
net	The wire connecting a gate output with the inputs of its fan-out gates.
netlist	Design description in form of gate networks.
PCB	Printed Circuit Board.
PERL	Practical Extraction and Report Language, a text manipulating script language.
queue	Storage space for one or a set of levels.
SPARC v.8	Scalable Processor Architecture version 8.

stable-state	Time-point when all signals have propagated to a latch or register.
system-on-chip	Everything integrated in one chip, no more PCBs crammed with chips.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuit.

9 Acknowledgements

I would like to thank Bengt Werner at Virtutech, and Erik Larsson at LiU, for their ideas and support. I also want to thank Tomas Holmberg for EMIL. Can’t do with it, and could certainly not have done without it. Thanks.

10 References

[COSMOS 87] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, Thomas Sheffler
COSMOS: A Compiled Simulator for MOS Circuits
24th ACM/IEEE Design Automation Conference, pp. 9-16, 1987

[EMIL 2002] Tomas Holmberg
EMIL: An Event-Driven Multi-Queue Interpretive Levelized Gate-Level Simulator
Thesis work, Microelectronics and Information Technology, Royal Institute of Technology, Sweden, 2002

[LECSIM 90] Zhicheng Wang, Peter M. Maurer
LECSIM: A Levelized Event Driven Compiled Logic Simulator
27th ACM/IEEE Design Automation Conf., pp. 491-496, 1990

[SLS 88] Zeev Barzilai, Daniel K. Beece, Leendert M. Husmain, Vijay S. Iyengar, Gabriel M. Silberman
SLS – A Fast Switch-Level Simulator
IEEE Transactions on Computer-Aided Design vol. 7, no. 8, pp.838-849, August 1988

[SSIM 87] Lang-Terng Wang, Nathan E. Hoover, Edwin H. Porter, John J. Zasio
SSIM: A Software Levelized Compiled-Code Simulator
24th ACM/IEEE Design Automation Conference, pp. 2-8, 1987

[Ashar 95] Pranav Ashar, Sharad Malik
Fast Functional Simulation Using Branching Programs
Proceedings of the 1995 IEEE/ACM International Conference on Computer-aided Design, pp. 408-412, December 1995

[Bertacco 99] Valeria Bertacco, Maurizio Damiani, Stefano Quer
Cycle-based Symbolic Simulation of Gate-level Synchronous Circuits
36th ACM/IEEE Design Automation Conf., pp. 391-396, 1999

[Blaauw 93] David T. Blaauw, Larry G. Jones
Reducing the Scheduling Cost in Event-Driven Simulation Through Component Clustering
4th European Conference on Design Automation, pp. 18-22, 1993

[DeVane 97] Charles J. DeVane
Efficient Circuit Partitioning to Extend Cycle Simulation Beyond Synchronous Circuits
Proceedings of the 1997 IEEE/ACM International Conference on Computer-aided Design, pp. 154-161, November 1997

[French 95] Robert S. French, Monica S. Lam, Jeremy R. Levitt, Kunle Olukotun
A General Method for Compiling Event-Driven Simulations
32nd ACM/IEEE Design Automation Conference, 1995

[Gateways 94] Peter M. Maurer, Yun Sik Lee
Gateways: A Technique for Adding Event-Driven Behavior to Compiled Simulators
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 3, pp. 338-352, March 1994

[Jennings 91] Glenn Jennings
A Case against Event-Driven Simulation for Digital System Design
24th annual Symposium on Simulation, pp. 170-176, April 1991

[Jiang 2003] Yunjian Jiang, Slobodan Matic, Robert K. Brayton
Generalized Cofactoring for Logic Function Evaluation
40th Conference on Design Automation, pp.155-158, June 2003

[Jones 94] Larry G. Jones, David T. Blaauw
A Cache-Based Method for Accelerating Switch-Level Simulation
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 2, pp. 211-218, February 1994

[Lewis 91] David M. Lewis
A Hierarchical Compiled Code Event-Driven Logic Simulator
IEEE Transactions on Computer-Aided Design, vol. 10, no. 6, pp. 726-737, June 1991

[Maurer 94] Peter M. Maurer
The Inversion Algorithm for Digital Simulation
Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design, pp. 258-261, 1994

[Maurer 96] Peter M. Maurer
Is Compiled Simulation Really Faster than Interpreted Simulation?
9th International Conference on VLSI Design, pp. 303-306, 1996

[Maurer 97] Peter M. Maurer
The Inversion Algorithm for Digital Simulation
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 16, no. 7, pp. 762-769, July 1997

[Maurer 99] Peter M. Maurer
Efficient Simulation for Hierarchical and Partitioned Circuits
12th International Conference on VLSI Design, pp. 236-241, 1999

[Maurer 2000] Peter M. Maurer
Event Driven Simulation Without Loops or Conditionals
Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design, pp. 23-26, 2000

[Olukotun 98] Kunle Olukotun, Mark Heinrich, David Ofelt
Digital System Simulation: Methodologies and Examples
35th Conference on Design Automation, pp. 658-663, June 1998

[Shadow 93] Peter M. Maurer
The Shadow Algorithm: A Scheduling Technique for both Compiled and Interpreted Simulation
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, no. 9, pp. 1411-1413, September 1993

[Yee Au 91] Wing Yee Au, Daniel Weise, Scott Seligman
Automatic Generation of Compiled Simulations through Program Specialization
28th ACM/IEEE Design Automation Conference, pp. 205-210, 1991

[AMD] Advanced Micro Devices Inc., www.amd.com, 2004

[EDIF] www.edif.org, 2004

[Gaisler] Gaisler Research, www.gaisler.com, 2004

[SPARC] SPARC International Inc.
The SPARC Architecture Manual version 8, 1992
www.sparc.org, 2004

11 Appendix A, Statistics

11.1 Split statistics

Gate type	*In Leon*	*Splits*	*Extra nets**	*EMIL evaluations*	*EMIL Activity*	*WCET***	*Extra evaluations*
MUX21L	730	4	2,190	6,992,736	19%	4	20,978,208
FDS2L1	676	5	2,950	3,808,294	11%	5	15,233,176
MUX31L	34	7	204	1,176,269	69%	5	4,705,076
AO12P	237	5	948	4,062,416	34%	2	4,062,416
AO7	536	2	536	3,698,926	14%	2	3,698,926
MUX21H	78	4	234	965,651	25%	4	2,896,953
AO2	383	3	766	2,426,294	13%	2	2,426,294
AO1P	284	2	284	2,032,912	14%	2	2,032,912
AO3	163	2	163	1,524,714	19%	2	1,524,714
AO4	266	3	532	1,520,091	11%	2	1,520,091
AO11	152	4	456	1,519,467	20%	2	1,519,467
FJK1S	26	8	195	151,252	12%	7	907,512
EO1	76	3	152	855,586	23%	2	855,586
FA1A	61	2	0	845,849	28%	2	845,849
AO6	109	2	109	775,939	14%	2	775,939
MUX81P	8	12	88	67,956	17%	12	747,516
EON1	122	3	244	705,495	12%	2	705,495
IVDA	52	2	0	401,923	15%	2	401,923
FD1S	98	5	392	67,976	1%	5	271,904
FDS2LP	3	5	12	58,395	39%	5	233,580
AO7P	7	2	7	229,651	66%	2	229,651
B2I	19	2	0	176,862	19%	2	176,862
HA1	40	2	0	145,418	7%	2	145,418
AO3P	4	2	4	72,300	36%	2	72,300
FDS2	102	2	102	63,853	1%	2	63,853
FD4S	1	5	4	5,539	11%	5	22,156
B3IP	4	2	0	5,879	3%	2	5,879
IVDAP	5	2	0	1,990	1%	2	1,990
FJK1	1	6	6	342	1%	6	1,710
MUX41	1	7	6	1	0%	7	6
FD1	575	1	0	624,193	2%	1	0
LD2	128	1	0	13,031,101	204%	1	0
FD2	27	1	0	20,848	2%	1	0
FD4	14	1	0	5,992	1%	1	0
FD1P	1	1	0	67	0%	1	0
Total	5,023		10,584	48,042,177	19%		67,063,362

** FDS2L, FJK1S and FJK1 add nets for some earlier unused inverted outputs, thus adding slightly more than expected (splits-1)*in_leon*

*** WCET (Worst Case Evaluations Trigged by singel input change)*

11.2 Schedule and evaluation statistics

Type	*Schedules*	*Percent*	*Explanation*
Real	117,401,953	89.83%	Not scheduled before
Dummy	13,287,387	10.17%	Scheduled before
Total	130,689,340		

Type	*Evaluations*	*Percent*	*Explanation*
AtZero	88,674,199	73.27%	Standard type
Eval1Up	12,800,128	10.58%	Change on latch enable input
HugeNet	9,000,661	7.44%	Standard type with large fanout
QueueSw	4,473,446	3.70%	Queue switches
SynFF	3,431,113	2.83%	Change on synchronous flipflop data input
AtEven	871,013	0.72%	Full adder sum function
AtPos	845,849	0.70%	Full adder carry function
Compiled	672,523	0.56%	User-defined (Pads, registers and cache)
Latch	231,229	0.19%	Change on latch data input
FlipFlop	26,762	0.02%	Change on asynchronous flipflop data input
FixAt	982	0.00%	Logic 0/1 gates
Eval1UpIf	42	0.00%	Change on flipflop clear or preset input
Total	121,027,947		

11.3 Fan-out statistics

Fanout statistics for all gates

Fanout	*Orig nets*	*Percent*	*After split*	*Percent*	*After buf*	*Percent*
0	306	2.82%	306	1.43%	306	1.40%
1	6,483	59.71%	16,480	76.86%	16,934	77.34%
2	1,948	17.94%	2,254	10.51%	2,254	10.29%
3	833	7.67%	940	4.38%	940	4.29%
4	381	3.51%	448	2.09%	448	2.05%
>5	907	8.35%	1,014	4.73%	1,014	4.63%
Added	0		10,584		454	
Total	10,858		21,442		21,896	

Fanout statistics for AtZero type

Fanout	*Amount*	*Percent*	*Evaluations*	*Percent*	*Of total evaluations*
1	14,662	88.43%	78,965,438	89.05%	65.25%
Other	1,919	11.57%	9,708,761	10.95%	8.02%
Total	16,581		88,674,199		73.27%

12 Appendix B, Grammar

```
gate:      NAME { stmtlist }

stmtlist:  stmtlist stmt
|          stmt

stmt:      set ';’

assign:    OUTPUT '=' or

or:        or '+' and
|          and

and:       and not
|          not

not:       expr '\''
|          expr

expr:      '(' assign ')'
|          '(' or ')'
|          INPUT
|          LOGIC

clock:     assign ON INPUT
|          assign WHEN INPUT
|          assign

set:       clock RESET INPUT
|          clock PRESET INPUT
|          clock
```

LINKÖPINGS UNIVERSITET

Avdelning, Institution
Division, Department

Institutionen för datavetenskap
581 83 LINKÖPING

Datum
Date
2004-05-12

Språk Language	**Rapporttyp** Report category
Svenska/Swedish	Licentiatavhandling
X Engelska/English	X Examensarbete
	C-uppsats
	D-uppsats
	Övrig rapport
	____

ISBN

ISRN LITH-IDA/DS-EX--04/046--SE

Serietitel och serienummer **ISSN**
Title of series, numbering

URL för elektronisk version
http://www.ep.liu.se/exjobb/ida/2004/dt-d/046/

Titel Simulering av miljoner grindar med Count Algoritmen
Title

The Counting Algorithm for simulation of million-gate designs

Författare Klas Arvidsson
Author

Sammanfattning
Abstract
A key part in the development and verification of digital systems is simulation. But hardware simulators are expensive, and software simulation is not fast enough for designs with a large number of gates. As today's digital designs constantly grow in size (number of gates), and that trend shows no signs to end, faster simulators handling millions of gates are needed. We investigate how to create a software gate-level simulator able to simulate a high number of gates fast. This involves a trade-off between memory requirement and speed. A compact netlist representation can utilize cache memories more efficient but requires more work to interpret, while high memory requirements can limit the performance to the speed of main memory. We have selected the Counting Algorithm to implement the experimental simulator MICA. The main reasons for this choice is the compact way in which gates can be stored, but still be evaluated in a simple and standard way. The report describes the issues and solutions encountered and evaluate the resulting simulator. MICA simulates a SPARC architecture processor called Leon. Larger netlists are achieved by simulating several instances of this processor. Simulation of 128 instances is done at a speed of 9 million gates per second using only 3.5MB memory. In MICA this design correspond to 2.5 million gates.

Nyckelord
Keyword
simulation, count-algorithm, gate-level, interpretive, event-driven, hierarchical, multi-queue, computer-aided design, logic evaluation, zero-delay