SSIM: A SOFTWARE LEVELIZED COMPILED-CODE SIMULATOR

Laung-Terng Wang, Nathan E. Hoover, Edwin H. Porter, and John J. Zasio

Research & Development Department
AIDA Corporation
5155 Old Ironsides Drive
Santa Clara, CA 95054

ABSTRACT

This paper presents a new logic simulation technique that uses software levelized compiled-code (LCC) for synchronous designs. Three approaches are proposed: C source code, target machine code and interpreted code. The evaluation speed for the software LCC simulator (SSIM) is about 140,000 (gate) evaluations per second using C source code or target machine code, or 50,000 evaluations per second using interpreted code. It is about 40 to 100 times slower than the AIDA hardware LCC simulator, but is about one order of magnitude faster than a traditional software event simulator. For a 32-bit multiplier with gate activity more than 100%, experiments indicate that SSIM runs about 250 to 1,000 times faster than the AIDA event simulator that evaluates about 4,500 gates per second.

Index Terms — Levelized compiled-code (LCC) simulation, Logic simulation, Synchronous design.

1. INTRODUCTION

The traditional selective-trace, event-driven simulation technique [Breuer 76] has been widely used to simulate digital circuits. The major advantages of using this approach are its abilities of (1) handling both synchronous and asynchronous designs, and (2) performing timing as well as functional analysis during simulation. The problem with this approach is that the software simulation speed is very slow, typically, at about 1,000 evaluations per second running in a one-million-instructions-per-second (1-MIPS) machine [Smith 86]. As designs get larger, it becomes more difficult to rely on software event-driven simulation to obtain reasonable performance.

Hardware accelerators [Blank 84] are commonly used to solve this problem. With hardware acceleration, the performance improvement is, usually, over 100 times. Designers, however, must pay a sizable extra cost, usually, in the range between $100K to $1M, in order to benefit from this speed. Moreover, for complex system designs, gate activity is typically very high, often over 100 percent. In this case, even with hardware acceleration, event simulation is still very time-consuming.

Today, as the demands for maintainability, testability, portability (the ability to implement the design in more than one technology) and improved performance (through pipelining) increases, synchronous designs become the preferred method for very large-scale integration (VLSI) applications [Chiang 86]. When combined with an acceleration coprocessor and levelized synchronous design techniques, hardware compiled-code simulation [Chiang 86] [Ishiura 85] [Pfister 82] can easily reach millions of gate evaluations per second, performance that event-driven simulation can hardly obtain.

Levelized compiled-code (LCC) simulation requires logic levelization and code generation for a synchronous design before logic simulation can be performed. Logic levelization orders gates (primitives) in the design which are all one-level deep, two-level deep, etc. from primary inputs and data outputs of storage elements. The ordered sequence of the gates is then translated into machine-executable code. Every gate is evaluated once during each clock cycle, independent of the input pattern dynamics. Performance of LCC simulation, thus, does not depend on the gate activity in the design, and the need for event queue management is entirely eliminated.

This paper will address three cost-effective software logic simulation techniques using LCC: C source code, target machine code and interpreted code. The program executing the compiled-code at the software level is called *SSIM, a software levelized compiled-code simulator*. Experiments will indicate that the simulation speed can reach 140,000 (equivalent 2-input) gate evaluations per second using target machine code compiled from C code, or 50,000 gate evaluations per second using interpreted code.

Given a synchronous design, a C program can be generated, containing only a set of assignment statements, one for each gate in the design. The advantage of using this approach is its high portability to any target machine. The problems with this approach is that C code must be first compiled before it can be used for simulation.

Target machine code solves the above compilation problem by directly generating machine-executable code for the target machine (e.g., Motorola 68020, DEC 11/780, or IBM 370). The problem with this approach is that different code generators must be maintained. Whenever there is a change in the code generation process, this change must be propagated to all code generators.

To ease maintainability, interpreted code can be used as an alternative to C source code and target machine code. During simulation, SSIM will interpret (or emulate) each instruction used in the hardware LCC accelerator. Since the same code is used for simulation, SSIM using the interpreted code approach will be fully independent of all target machine codes. The penalty, however, one has to pay is that the simulation speed falls to between 25,000 to 50,000 evaluations per second.

2. LCC LOGIC SIMULATION ARCHITECTURE

Fig. 1 shows an LCC logic simulation architecture for synchronous designs. The synchronous design is first compiled by a netlist compiler into a netlist. Levelized compiled-code (LCC) can then be generated from the netlist either by a *software compiled-code generator (SIMGEN)* or by a *hardware compiled-code generator (COGEN)*.

When SIMGEN is used, C source code or target machine code can be produced. If C source code is needed, SIMGEN will generate the C program and then invoke the C compiler to produce machine-executable code. If target machine code is needed, SIMGEN will directly generate machine-executable code for that target machine. No C program will be generated. This target machine code is then used in combination with the input test pattern file for simulation. SSIM is the software LCC simulator that produces the output results.

When COGEN is used, simple RISC (reduced-instruction set computer) code is produced. This code can then be used either by a hardware accelerator or be interpreted in software. If hardware acceleration is to be used, the hardware LCC simulator, CSIM, will coordinate with the hardware accelerator and produce output results. If software interpretation is desired, the *interpretive SSIM* (or *SSIM using interpreted code*) will be called to do the task.

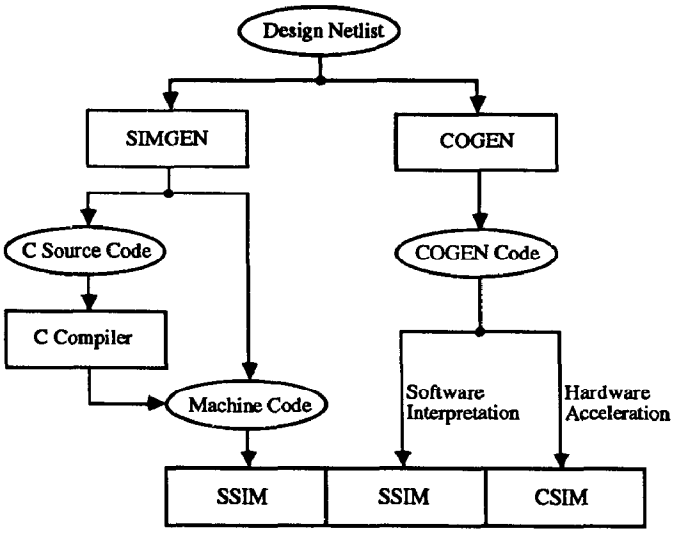

Figure 1. An LCC logic simulation architecture.

The following algorithm describes how the compiled code is used in logic simulation. It is very similar to that given in [Denneau 82]. For every clock cycle, the entire program is executed once. This LCC simulation approach, obviously, is very different from logic simulation that uses selective-trace, event-driven approach (event simulation) [Breuer 76]. For designs with lower gate activity (event activity), e.g., 10% or less, not much benefit (in terms of simulation speed) may be gained compared to that using event simulation. For designs with high activity, however, LCC simulation becomes highly desirable.

Experimental results will indicate that among the three LCC approaches proposed, (1) in terms of simulation speed, the interpretive SSIM runs the slowest, (2) in terms of portability and maintainability, SSIM using target machine code is the least portable, and (3) in terms of code generation, SIMGEN takes the longest to compile the C source code.

Algorithm 1: (LCC logic simulation)

Repeat n clock cycles
{
(1) Take an input pattern that corresponds to the clock cycle,
(2) Execute the entire LCC Simulation once,
(3) Produce output results, and
(4) Compare against expected output results if necessary.
}

3. LCC CODE GENERATION

Fig. 2 is a flow diagram of an LCC code generation process for synchronous designs. Given a synchronous design, LCC is produced by performing logic optimization, logic levelization, and code generation. Depending on the designer's request, three types of codes can be generated: C source code, target machine code, or hardware COGEN code. SIMGEN is used to generate C source code or target machine code, whereas COGEN is used to generate hardware COGEN code.

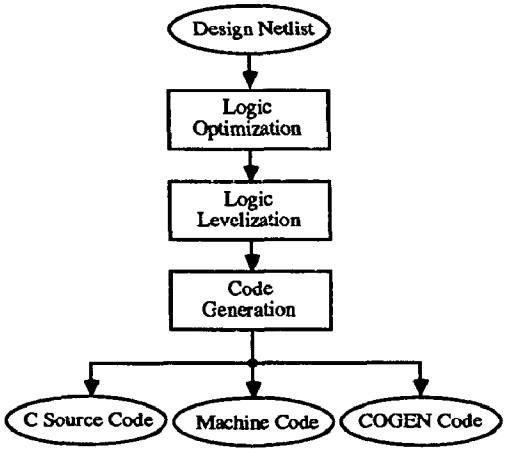

Figure 2. Levelized compiled-code generation algorithm.

3.1. Logic Optimization

SIMGEN and COGEN provide designers with two options of logic optimization: NOOPT (no optimization) and FULLOPT (full optimization). When no optimization mode is selected, the code generated by SIMGEN/COGEN represents the actual circuit topology with one statement for

each gate. When full optimization mode is chosen, n-input logic gates with m inputs tied high or low are changed into n-m input gates. If n-m = 1, the gate is changed into an inverter or a buffer. Finally, buffers and inverters are optimized out of the network. This mode of operation can substantially increase logic simulation speed.

3.2. Logic Levelization

Logic levelization [Breuer 76] is a process of emulating the data flow from primary inputs and latch outputs to primary outputs and latch inputs. If a feedback loop exists in the design, SIMGEN/COGEN will identify those gates (or objects) that cannot be levelized.

Consider the 2-input XOR gate shown in Fig. 3. The circuit contains 2 input ports (primary inputs), 1 output port (primary output) and 5 objects (gates). It has a total of 7 nets (nodes). When no optimization is used, the objects in the circuit will be levelized as numbered in the figure. The algorithm used in SIMGEN/COGEN for logic levelization is given as follows:

Algorithm 2: (Logic levelization)

(1) Mark all nets connecting to input ports and latch outputs as "available";

(2) Put the fanout objects connecting to input ports and latch outputs into a queue;

(3) While (the queue is not empty)
{
Take an object from the queue; if all of its input pins are marked "available"
{
(A) Mark all output nets of the object as "available",
(B) Generate C source code, target machine code, or hardware COGEN code for the object, and
(C) Put into queue those fanout objects that connect to the output nets.
}
}

(4) If the number of "available" nets is not equal to the total number of nets, then report the error. The circuit contains feedback loops on those objects generating nets that are not marked "available".

3.3. Code Generation

Levelized compiled-code (LCC), in either instruction or assignment statement format, is generated during logic levelization. For general-purpose applications, assignment statements can be generated in C source code. For dedicated applications, instructions can be generated in target machine code. Interpreted code is a special case of target machine (COGEN) code that is currently used in the AIDA hardware accelerator. This approach permits LCC simulation to run on any target machine other than the hardware accelerator.

3.3.1. C Source Code

For a given synchronous design, a C program is produced consisting of a set of assignment statements that emulates the behavior of the design. C source code is very easy to debug and port to any target machine as long as the target machine can compile C programs. The disadvantage with this approach is that overall LCC simulation time is degraded due to the need to compile the C program first. If fault simulation is required, tremendous effort to generate compiled-code for different faults is required. This may not be a viable approach for fault simulation.

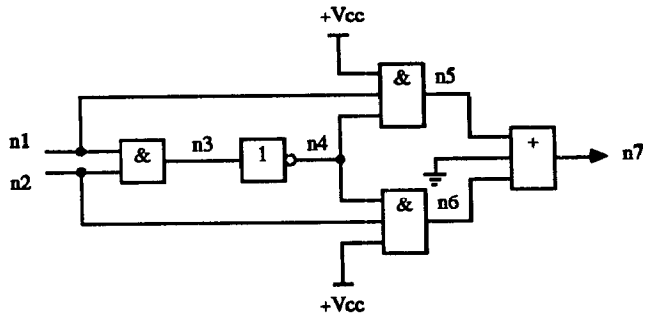

Figure 3. 2-input XOR gate for compiled-code generation.

Consider the 2-input XOR gate shown in Fig. 3, again. The C source codes corresponding to NOOPT and FULLOPT options are given in Tables 1 and 2, respectively. When the NOOPT option is used, five assignment statements, one for each object, are generated, as shown in Table 1. The n5, n6 and n7 statements contain references to V0 and V1, which denote logic state values 0 and 1, respectively. The circuit topology is not changed during this mode of operation. When the FULLOPT option is used (see Table 2), the n4 statement is removed. Instead, the n5 and n6 statements are evaluated by using the complement value of net n3 (comp[n3]). The tied high and low inputs to nets n5, n6 and n7 are also removed.

It should be noted that this example is only for illustration. When tristate gates and other types of objects that produce the high-impedance state (VZ) are considered, the code generation process will be slightly different.

To handle the high-impedance state, two schemes can be used . One scheme is to map states {V0, VZ, VX, V1} to {V0, VX, VX, V1}, respectively whenever it is necessary, where VX is an unknown state. This mapping is very expensive since an indirect indexing is required for each net in the assignment statement.

The other scheme is to use a tag bit, which is called VZ_MASK in our examples. Whenever any object creates a VZ state, state VX is stored in the output net state table, while a VZ_MASK tag is set to indicate that it is actually a VZ state. Special conversion on input and output port states is also required. Though this scheme looks cumbersome, it is much more efficient than the first scheme. This is because in a design, in general, only a small portion of nets will exhibit VZ states. Thus, the code generation process is based on the second scheme.

For interested readers, a synchronous circuit consisting of a RAM functional model (written in C), a tri-state gate, and a bidirectional I/O pin is given in the Appendix. C source code and COGEN code are listed.

Table 1. C source code for Fig. 3 using NOOPT mode

```
typedef unsigned char byte;
#define V0      0x00
#define VZ      0x01
#define VX      0x02
#define V1      0x03
byte comp[] = {V1, VX, VX, V0};

sim_design()
{
   byte n[7];

   n[3] = n[1] & n[2];
   n[4] = comp[n[3]];
   n[5] = V1 & n[1] & n[4];
   n[6] = n[4] & n[2] & V1;
   n[7] = n[5] | n[6] | V0;
}
```

Table 2. C source code for Fig. 3 using FULLOPT mode

```
typedef unsigned char byte;
#define V0      0x00
#define VZ      0x01
#define VX      0x02
#define V1      0x03
byte comp[] = {V1, VX, VX, V0};

sim_design()
{
   byte n[7];

   n[3] = n[1] & n[2];
   n[5] = n[1] & comp[n[3]];
   n[6] = comp[n[3]] & n[2];
   n[7] = n[5] | n[6];
}
```

3.3.2. Target Machine Code

Instead of generating C source code as the first step to simulation, it is possible to generate code directly from the design that is executable by a target machine. The code generated in this manner is called *target machine code*. It consists of a set of machine-executable instructions.

Target machine code is very efficient to run during logic or fault simulation. The high efficiency is partly due to the technique used for (machine) code optimization. Since register access is usually faster than memory access, target machine code can yield very high efficiency if code optimization can take advantage of all data registers available in the target machine. It might be more efficient if logic levelization can be done so that code optimization can fully utilize the long data registers.

Table 3 shows the COGEN instructions for the circuit given in Fig. 3. The function of each instruction will be described in the next subsection. These instructions were generated by COGEN using FULLOPT mode. It can be seen that for the n7 statement only two instructions were generated. This is because the state of net n6 resides in the accumulator.

Table 3. COGEN code for Fig. 3 using FULLOPT mode

```
(01)   LA     n1     | Load n1
(02)   AN     n2     | And with n2
(03)   ST     n3     | Generate n3
(04)   LA     n1     | Load n1
(05)   AN,C   n3     | And with comp[n3]
(06)   ST     n5     | Generate n5
(07)   LA     n2     | Load n2
(08)   AN,C   n3     | And with comp[n3]
(09)   ST     n6     | Generate n6
(10)   OR     n5     | Or with n5
(11)   ST     n7     | Generate n7
(12)   HA     0      | Halt simulation
```

3.3.3. Interpreted Code

Although the simulation speed using C source code and target machine code is fast, (about 140,000 gate evaluations per second for a 3-MIPS machine), many serious problems have been encountered.

First, when the design is large, e.g., more than 1,600 primitives, it takes a long time to compile the C source code. Although target machine code can be directly generated from the design, the task becomes difficult as different machine code translations for each type of target processor must be done.

Second, program maintainability and quality assurance (QA) are serious concerns. Whenever there is a change in the code generation process, this change must be assured to propagate through C source code and all target machine codes.

With these concerns, interpreted code is proposed as an alternative to C source code and target machine code. Since only COGEN is required for simulation, SSIM using the interpreted code approach will be fully independent of all target machines. This substantially eases portability, compatibility, maintainability and quality assurance problems. The penalty, however, is that the simulation speed falls to 50,000 evaluations per second.

Table 4 lists 16 basic COGEN opcodes currently used in the AIDA design system [Aida 86]. The first 11 opcodes are associated with logic operations. Of the remaining 5 opcodes, the CO (compare) opcode is hardware-oriented. The LS (load-store) opcode, which is associated with D flip-flip (DFF) and latch operations, controls whether the present state should be stored in the state memory or not. The HA

(halt) opcode stops simulation or starts to execute the next functional model.

Table 4. The 16 COGEN instruction opcodes

(01)	LA	Load accumulator
(02)	AN	AND operation
(03)	NA	NAND operation
(04)	OR	OR operation
(05)	NO	NOR operation
(06)	XO	Exclusive-OR operation
(07)	XN	Exclusive-NOR operation
(08)	TR	Tri-state operation
(09)	TX	Transmission gate operation
(10)	DO	Dot operation
(11)	WD	Weak driver operation
(12)	CO	Compare operation
(13)	NI	No-op
(14)	ST	Store accumulator
(15)	LS	Load store control register
(16)	HA	Halt operation.

Each opcode takes two operands: an address field and a modifier field [Aida 86]. Emulation of COGEN code is done by simply interpreting COGEN instructions one at a time during simulation. In each interpretation, the net state associated with the address field is first computed. Under the control of the modifier field, the required operation is then performed according to the opcode.

Table 5.
Logic tables for interpreting COGEN code

A	B	LA	AN	OR	XO	TR	TX	DO	WD
0	0	0	0	0	0	Z	Z	0	0
0	1	0	0	1	1	Z	Z	X	1
0	X	0	0	X	X	Z	Z	X	X
0	Z	0	0	X	X	Z	Z	0	0
1	0	1	0	1	1	0	0	X	0
1	1	1	1	1	0	1	1	1	1
1	X	1	X	1	X	X	X	X	X
1	Z	1	X	1	X	X	Z	1	1
X	0	X	0	X	X	X	X	X	0
X	1	X	X	1	X	X	X	X	1
X	X	X	X	X	X	X	X	X	X
X	Z	X	X	X	X	X	X	X	X
Z	0	Z	0	X	X	X	X	0	0
Z	1	Z	X	1	X	X	X	1	1
Z	X	Z	X	X	X	X	X	X	X
Z	Z	Z	X	X	X	X	X	Z	Z

To execute the "LA,C M" instruction, (load the complement of net M to the accumulator), the state of the net is first computed. Then, the complement of that state is transferred to the accumulator. Interpretation of logic operations such as AN, OR, and TR are done by table lookup. Table 5 lists some of the tables used to perform logic operations, where A is the state in instruction operand, and B is the value in the accumulator.

The interpreted code approach entirely eliminates the use of an LCC hardware accelerator. It is very effective for small designs. The problem with this approach is that the simulation speed is much slower (usually, two orders of magnitude) compared to the hardware accelerator. Thus, for larger designs, the running time may be prohibitively long. In these cases, a hardware accelerator is recommended.

4. PERFORMANCE

Table 6 shows some statistics and experimental results about six benchmarked circuits running on an Apollo DN570-T 32-bit workstation (with instruction cache). The machine has 16 megabytes of physical memory and uses a Motorola 68020 CPU running at 20 MHz. Peak performance is about 3 to 4 MIPS. "alu4" and "alu32" are 4-bit and 32-bit combinational ALUs, respectively. "mpy32" is a 32-bit multiplier implemented with a 2-dimensional, iterative full-adder array. "achip1" is a real chip design containing a 72-stage random number generator, a 32-bit ALU, a 64-stage signature analyzer, and some other logic. "achip2" is another real chip design including about 1,700 D flip-flops, 6,800 transmission gates and 3,100 dot gates. These dot gates are not counted as equivalent gates. "alu102k" is a large combinational circuit containing 102,100 equivalent 2-input gates.

The simulation time for the first two circuits is mostly dominated by the time needed for setting input patterns. Not much time difference was observed between LCC and event simulations. For the last four circuits, time to set input patterns had been reduced to a minimum. Results indicated that the interpretive SSIM is about 3 to 4 times slower than SSIM using (compiled) C source code, and is about 50 to 140 times slower than CSIM, the AIDA hardware LCC simulator. If C code compile time is considered, then SSIM using C source code loses attractiveness, since the C compiler takes a long time to compile. In the case of using target machine code, however, SSIM remains attractive since time to compile a design is very close to the COGEN time.

It is interesting to note that for one particular circuit "mpy32", SSIM runs about 250 to 1,000 times faster than ESIM, the AIDA event simulator. The reason for this significant improvement is because "mpy32" has very high gate activity (see Table 7). The result is not surprising since in event simulation, an object may have to be evaluated more than once during each clock cycle. This strongly demonstrates that LCC simulation is highly desirable for such designs.

Table 7 lists simulation performance calculated from the results in Table 6. It indicates that the interpretive SSIM and SSIM using C source code reach the execution speeds of 50,000 and 140,000 gate evaluations per second, respectively.

5. SUMMARY AND CONCLUSIONS

A logic simulation technique using levelized compiled-code (LCC) for synchronous designs is presented. This technique allows designers to run logic simulation in machines (such as CAE workstations or computers) without a

hardware accelerator. It requires an LCC generator to levelize the design and generate machine-executable code.

The first approach produces C source code that represents the design. This code is quite portable. The problem is that the C compiler must be used before the code can be used for simulation. Target machine code eliminates the time-consuming compilation process, however, it introduces maintainability and portability problems.

By using an interpretive approach, the software LCC logic simulator, SSIM, reaches 50,000 evaluations per second. Although the speed is much slower than 10 MIPS (or 5M gates per second) running on the AIDA hardware accelerator, SSIM allows designers to run LCC on any target machine without portability, maintainability, and incompatability problems. In addition, it expands the simulation capabilities to user-defined functional models, where a functional model is a C program that describes the behavior of a macro, such as a RAM or ALU.

Experiments indicate that SSIM runs much faster than a traditional event simulator that runs at about 1,000 evaluations per second. In some cases, it outperforms those event-driven logic simulation programs that use hardware accelerators to improve performance.

ACKNOWLEDGMENTS

The authors wish to thank Professor Edward J. McCluskey, Darrell R. Parham, Dr. Aamer Mahmood, Edwin B. Forbes, and G. Patrick Scandalis for their valuable comments and suggestions. The help from Hua-Ju C. Wang and John Hevelin in preparing this manuscript is also deeply appreciated.

REFERENCES

[Aida 86] Aida Corp., *Aida Design System*, Vols. II and IV, Aida Corp., Santa Clara, California, 1986.

[Blank 84] Blank, T., "A Survey of Hardware Accelerators Used in Computer-aided Design," *IEEE Design & Test of Computers*, Vol. 1, No. 3, pp. 21-39, August 1984.

[Breuer 76] Breuer, M.A., and A.D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Inc., Woodland Hills, CA, 1976.

[Chiang 86] Chiang, M., and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, pp. 87-91, March 1, 1986.

[Denneau 82] Denneau, M.M., "The Yorktown Simulation Engine," Proc. of the *ACM/IEEE 19th Design Automation Conf.*, pp. 55-59, June 1982.

[Ishiura 85] Ishiura, N., H. Yasuura, T. Kawata, and S. Yajima, "High-Speed Logic Simulation on a Vector Processor," Digest of Papers, *IEEE 1985 Int'l Conf. on Computer-Aided Design (ICCAD-85)*, pp. 119-121, Santa Clara, CA, Nov. 18-21, 1985.

[Pfister 82] Pfister, G.F., "The Yorktown Simulation Engine: Introduction," Proc. of the *ACM/IEEE 19th Design Automation Conf.*, pp. 51-54, June 1982.

[Smith 86] Smith, R.J., II, "Fundamentals of Parallel Simulation," Proc. of the *ACM/IEEE 23th Design Automation Conf.*, pp. 2-12, June 1986.

APPENDIX

This appendix describes a code generation process for circuits containing functional models, tri-state gates and bidirectional I/O pins, where a functional model is a C program that describes the behavior of a macro, such as an ALU, a RAM or a multiplier.

Fig. 4 shows a 16-by-1 RAM cell (functional model) with read/write control logic. Net n6 acts as an ENABLE pin. When n6 is in the active-high (V1) state, the data coming out from the RAM (operated in read mode) will appear at the output port n15. When n6 is in the active-low (V0) state, data at n7 will appear at n15, and be stored in the RAM if the RAM is in write mode (or the n5 state is V0).

Table 6.
Experimental simulation results (time in seconds)

circuit	primitives	equiv. gates	instructions[1]	patterns	total evals.[2]	SIMGEN time[3]	COGEN time	SSIM time[4]	SSIM time[5]	CSIM time	ESIM time
alu4	52	82	201	16,384	609,931	13	1	85	47	70	185
alu32	502	984	1,717	448	123,258	52	2	20	20	10	36
achip1	1,643	3,184	4,645	1,000	2,223,619	213	5	84	23	1	504
mpy32	12,681	14,634	41,829	1,000	700,118,451	1,482	30	590	152	10	161,762
achip2	19,208	16,344	52,956	1,006	*	*[6]	51	923	*	7	*
alu102k	54,227	102,100	136,450	2,688	*	*[6]	111	5748	*	50	*

[1] For CSIM and Interpretive SSIM
[2] From Event Simulation
[3] Including C code compile time
[4] Interpretive SSIM
[5] SSIM using compiled C source code
[6] C compiler time too long, not C code generation
* Test case too large to run

Table 8 lists the C source code generated for the circuit with the corresponding COGEN code given in Table 9. To handle such a circuit, a pseudo-buffer (n10) and a dot gate (n14) have been created. It can be seen that C source code generated by SIMGEN is not as efficient as hardware COGEN code. This is because in SIMGEN, the high-impedance state is given special treatment, i.e., a VZ_MASK tag bit is used to handle tri-state gates.

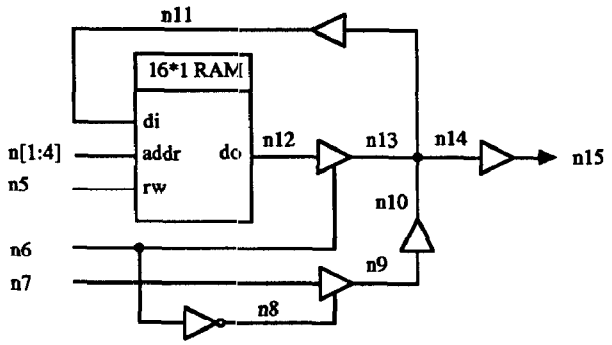

Figure 4. Circuit containing a RAM functional model.

Table 9. COGEN code for Figure 4 using NOOPT mode

```
(01)  LA,C  n[6]
(02)  ST    n[8]
(03)  LA    n[7]
(04)  TR    n[8]
(05)  ST    n[9]
(06)  ST    n[10]
(07)  ST    n[11]
(08)  HA    0       | Execute functional model
(09)  LA    n[12]
(10)  TR    n[6]
(11)  ST    n[13]
(12)  DO    n[10]
(13)  ST    n[14]
(14)  ST    n[15]
(15)  HA    0       | Halt simulation
```

Table 8. C source code for Fig. 5 using NOOPT mode

```
typedef unsigned char byte;
#define V0      0x00
#define VZ      0x01
#define VX      0x02
#define V1      0x03
byte comp[] = {V1, VX, VX, V0};
byte dot_table[4][4] = {{V0, V0, VX, VX},
                        {V0, VZ, VX, V1},
                        {VX, VX, VX, VX},
                        {VX, V1, VX, V1}};
byte tr_table[4][4]  = {{VZ, VX, VX, V0},
                        {VZ, VX, VX, VX},
                        {VZ, VX, VX, VX},
                        {VZ, VX, VX, V1}};
#define VZ_MASK 0x01
#define z(k,state)  ((mask[(k)] & VZ_MASK) ? VZ : state)
#define set_z(k)    {mask[(k)] |= VZ_MASK; n[(k)] = VX;}
#define reset_z(k) mask[(k)] &= ~ VZ_MASK;

sim_design()
{
   byte n[15];

   n[8] = comp[n[6]];
   n[9] = tr_table[z(7,n[7])][n[8]];
   if (n[9] == VZ) set_z(9) else reset_z(9)
   n[10] = z(9,n[9]);
   if (n[10] == VZ) set_z(10) else reset_z(10)
   n[11] = z(10,n[10]);
   if (n[11] == VZ) set_z(11) else reset_z(11)
   (void) sim_funct(0);
   n[13] = tr_table[z(12,n[12])][z(6,n[6])];
   if (n[13] == VZ) set_z(13) else reset_z(13)
   n[14] = dot_table[z(13,n[13])][z(10,n[10])];
   if (n[14] == VZ) set_z(14) else reset_z(14)
   n[15] = z(14,n[14]);
   if (n[15] == VZ) set_z(15) else reset_z(15)
}
```

Table 7.
Simulation performance calculated from the results in Table 6

circuit	SSIM equiv. gates/sec[1]	SSIM equiv. gates/sec[2]	CSIM equiv. gates/sec	ESIM evals./sec	gate activity[3]
alu4	16,000	29,000	19,000	3,300	72%
alu32	22,000	22,000	44,000	3,400	54%
achip1	38,000	138,000	3,184,000	4,400	135%
mpy32	25,000	96,000	1,463,000	4,300	5,521%
achip2	17,800	*	2,349,000	*	*
alu102k	47,750	*	5,489,000	*	*

[1] Interpretive SSIM
[2] SSIM using Compiled C source code
[3] For Event Simulation
* Test case too large to run