SIMULATING DIGITAL CIRCUITS WITH ONE BIT PER WIRE

Andrew W. Appel

CS-TR-093-87

May 1987

Appeared in IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, &(9) : 987-993, September 1988.

Simulating Digital Circuits with One Bit Per Wire

Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Conventional digital circuit simulators represent circuits using linked data structures, using one or more pointers per connection. To simulate a circuit of N nodes requires space proportional to $N\log N$ bits.

Many circuits have a hierarchical or repetitive nature, so their specifications can be significantly smaller than the circuits themselves. This paper shows that such circuits can be simulated in space equal to one bit of memory per wire of the circuit, plus space proportional to the (smaller) size of the specification; that is, the space required is only $O(N)$ bits.

The algorithm has been implemented; measurements of its efficiency are given.

1. A simple model of synchronous circuits

A combinational digital logic circuit is one which implements one or more boolean functions on a given set of inputs using *and, or,* and *not* gates. These gates are connected using wires which tie exactly one output of one gate to one or more inputs of other gates. There are no cycles, in which the input to a gate functionally depends on the output of that gate.

The circuit can be modelled as an acyclic graph with directed edges whose nodes are labelled by *and, or, not, input,* and *output.* The *input* nodes have only out-edges; they provide the boolean values on which the graph implements boolean functions. The *output* nodes have only in-edges; they represent the functions computed.

Clocked synchronous sequential circuits have, in addition to the above-described components, latches that hold values. These latches copy their inputs to their outputs only on receipt of a clock signal. All of the latches in a synchronous circuit must be clocked at the same instant. It is legal to have a cycle in the graph as long as it goes through a latch. These circuits can be modelled as combinational circuits in which the inputs to the latches are considered outputs of the function-graph, and outputs of the latches are considered inputs to the function-graph.

This work was supported in part by NSF Grant DCR-8603453 and in part by a Digital Equipment Corporation Faculty Incentive Grant.

Simulation of a combinational circuit is the computation of the output values from the input values. This computation can be easily done as follows: the graph is first topologically sorted — that is, totally ordered in such a way that each node follows all the nodes on which it functionally depends. Then the boolean value at each node is computed in this order, using the previously calculated values of its inputs.

This algorithm is relatively efficient. For a circuit of N nodes, it takes time proportional to N and memory space proportional to $N\log N$ bits. We will use the uniform cost model of time-complexity, and the log cost model of space-complexity. This is appropriate for conventional computers because (in general) a full-word instruction is just as fast as a single-bit instruction, but it is always possible to make use of the individual bits of a memory word.

To simulate a synchronous sequential circuit, this algorithm can be executed for each simulated clock cycle. In each cycle, the outputs of the latches are propagated (by the simulation algorithm) to the inputs; then the inputs of the latches are all copied to their outputs.

Some circuits have very regular or hierarchical specifications, in which similar subcircuits are repeated many times. The specifications of these circuits can be much smaller than the circuits themselves. We show in this paper how to simulate such circuits in much less memory than the conventional algorithm uses: one bit per wire, rather than one pointer ($\log N$ bits) per wire.

2. Hierarchically specified circuits

Many large digital circuits have a hierarchal or repetitive nature. They can be specified using repeated instances of ''building blocks'' — smaller subcircuits. Many specification and simulation languages allow the hierarchical specification of such modules[1-4]. For example, a 64-bit shift register can be built from four 16-bit shift registers connected together. By specifying the design of the 16-bit shift register just once rather than four times, the length of the circuit's description can be shortened considerably.

In fact, the 16-bit shift register can itself be specified as four 4-bit shift registers connected together; by taking this process to the limit, a circuit of size N can often be specified in space $\log N$ (or perhaps $\log^2 N$ in the log-cost model). In practice, circuits are not always so regular or hierarchical; but even so, specifications can be much smaller than the circuits they describe.

This section describes a simple language for specifying hierarchical circuits. The language is not meant to be novel; it just helps clarify the explanation. Here, the curly braces are meta-syntax, so {x} means ''zero or more repetitions of x.''

```
spec  →  {mod} mod

mod   →  MODULE ID {decl} BEGIN {conn} END

decl  →  IN : {ID}
decl  →  OUT : {ID}
decl  →  LOCAL : {ID}
decl  →  ID : {ID}

conn  →  { pin <- } pin <- exp
conn  →  exp -> pin { -> pin }

exp   →  pin
exp   →  exp + exp
exp   →  exp * exp
exp   →  ~ exp
exp   →  ( exp )

pin   →  ID
pin   →  ID . ID
```

Each module specification gives the name of the module, followed by declarations of the input pins, output pins, internal pins, and submodules. Between the BEGIN and END are the specifications of gates and connections. The gates are represented using the infix operators + *(or)*, * *(and)*, and ~ *(not)*.

Figure 1 gives the specification of a half-adder. This module uses no submodules, and has input pins *x,y* and output pins *sum*, *carry*.

```
module HalfAdder
    in: x y
    out: sum carry
    begin (x+y)*~(x*y) -> sum
      x*y -> carry
    end
```

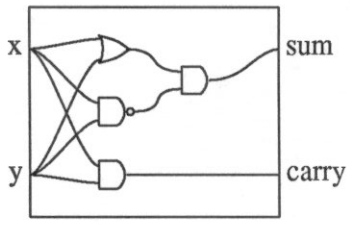

Figure 1.

Using three half adders, we can construct a two-bit full adder (Figure 2). The specification of FullAdder describes the three instatiations of the module HalfAdder, and the wires connecting them to each other and to the input and output pins. There is also one *or*-gate.

We will introduce a primitive latch component (Figure 3). The internals of this 4-bit register cannot be specified in the language, but the intent is that on every clock cycle (after all of the gates and connections have been fully propagated) the values of the d_i will be copied to the q_i of all *reg4* components. A 4-bit shift register can be made using one instance of *reg4* and some

```
module FullAdder
    in: x0 x1 y0 y1
    out: z0 z1 z2
    HalfAdder: low mid high
    begin x0 -> low.x
      y0 -> low.y
      low.sum -> z0
      x1 -> high.x
      y1 -> high.y
      high.sum -> mid.x
      low.carry -> mid.y
      mid.sum -> z1
      mid.carry + high.carry -> z2
    end
```

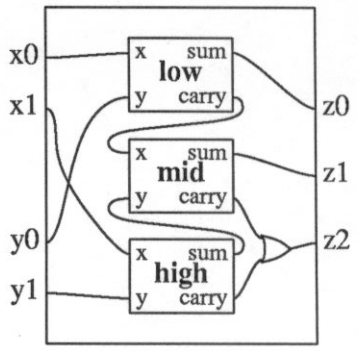

Figure 2.

```
module reg4
    in:  d0 d1 d2 d3
    out: q0 q1 q2 q3
    begin
    end
```

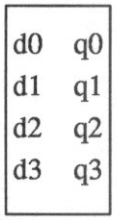

Figure 3.

additional gates. A 16-bit shift register can be made of four 4-bit shift registers; a 64-bit shift register can be made of four 16-bit shift registers, etc.

3. Economizing on memory

A circuit with millions of connections can be specified using a hierarchical specification (with nested modules as described in the previous section) just dozens of lines long. A straightforward simulation algorithm would be to expand the specification by substituting module bodies for their uses until a graph composed of just gates and wires was reached. This would take approximately one pointer per wire. Since each pointer must be large enough to encode the range of integers between 1 and the number of gates, an N-connection circuit takes $\Omega(N \log N)$ bits to encode in this way.

Previous systems have avoided a similar blowup in symbol-table size by storing the symbol-table as a hierarchical structure isomorphic to the circuit description, with repeated reference to a single description for each replicated module[4]. The space to store active circuit elements in such systems has still been proportional to the size of the circuit, however, not to the size of the description.

Suppose only one bit per connection is used to record the current signal value at that node of the circuit, and the original specification of the circuit is used as a ''road map'' to correctly propagate the signal values. The original specification is much smaller than the number of gates, so this method would take $O(N)$ bits of storage.

In the new algorithm, each module of the hierarchical specification will be implemented as a ''subroutine.'' The subroutine will be passed (as a parameter) a pointer to a block of memory in which are stored the current signal values of an instantiation of the module. The subroutine will propagate the module's input signals to its outputs, and then return.

If another instantiation of the same module is used, then the same subroutine will be called. A pointer to a different block of memory will be passed as an argument, because the signal values of different instantiations will differ even though the connections are similar.

The information about the connections between different gates in the module will be encoded once (in the subroutine) rather than many times (in each instantation). The blocks of data on which the subroutine operates will contain only signal values (1's and 0's, in a digital circuit), not pointers.

If one module contains some instantiations of smaller modules in the hierarchical specification, then the subroutine for the enclosing module will simply call the subroutines associated with the smaller modules. Thus, nested modules in the specification correspond to nested subroutine calls in the simulation.

4. Layout of data

To implement a module as a subroutine, the block of memory used to store the signal values at the connections of the module must be arranged in the same way for each instantiation. As an example, consider the half-adder example given previously. It has 2 input ''pins'' (x and y) and 2 output ''pins'' (*sum* and *carry*). We can allocate a 4-bit block of memory to hold the values at these pins (figure 4).

0	x	1
1	y	0
2	sum	1
3	carry	0

Figure 4

Then the subroutine implementing the half-adder can be passed a pointer to this block, and it will propagate the inputs to the outputs:

```
HalfAdder(p)
 {p[2] = (p[0]|p[1])&~(p[0]&p[1]);      (x+y)*~(x*y) → sum
  p[3] = (p[0]&p[1]);                   x*y → carry
 }
```

(A notation like that of the C language is used here just to illustrate the algorithm.) The HalfAdder subroutine can be passed a pointer to any 4-bit block of memory, and it will assume that these bits are to be interpreted as (x, y, *sum*, *carry*).

When one module is embedded within another, then a data block of appropriate size must be embedded within the enclosing module's data block. For example, the full adder described previously would have the data block shown in figure 5.

in	0	x0	1
	1	x1	1
	2	y0	0
	3	y1	1
out	4	z0	0
	5	z1	0
	6	z2	1
low	7	x	1
	8	y	0
	9	sum	1
	10	carry	0
mid	11	x	1
	12	y	0
	13	sum	1
	14	carry	0
high	15	x	1
	16	y	0
	17	sum	1
	18	carry	0

Figure 5

Each instantiation of the HalfAdder has a data block with the same internal arrangement, and each of these data blocks is a sub-block of the FullAdder's data block. The FullAdder procedure can be implemented as follows:

```
FullAdder(p)
 {p[7]   := p[0];     x0 → low.x
  p[8]   := p[2];     y0 → low.y
  HalfAdder(p+7);     propagate internals of low
  p[4]   := p[9];     low.sum → z0
  p[15]  := p[1];     x1 → high.x
  p[16]  := p[3];     y1 → high.y
  HalfAdder(p+15);    propagate internals of high
  p[11]  := p[17];    high.sum → mid.x
  p[12]  := p[10];    low.carry → mid.y
  HalfAdder(p+11);    propagate internals of mid
  p[5]   := p[13];    mid.sum → z1
  p[6]   := p[14]|p[18];     mid.carry + high.carry → z2
 }
```

5. Cycles and pseudo-cycles

In general, a circuit can be simulated in one pass (as shown in the implementations of HalfAdder and FullAdder) only if there are no cycles in the circuit. If this is the case, then the gates of the circuit can be topologically sorted; this gives an order of evaluation in which each gate is evaluated before any other gates which use the resulting value.

Some circuits cannot be topologically sorted because they contain cycles: the input of some gate depends functionally on the gate's output. Figure 6 shows two examples of such circuits.

Figure 6

The circuit on the left doesn't have any stable output, and the circuit on the right has two different stable states. Such circuits, though not without applications, are very difficult to simulate without using timing information (gate and wire delays), and will not be considered further here.

On the other hand, in a hierarchical specification of a circuit, there can be "pseudo-cycles." A pseudo-cycle occurs when the output of a module is fed (directly or indirectly) back to an input of that module, but in a way that wouldn't create a cycle if the module boundaries were erased. Figure 7 shows an example of a pseudo-cycle.

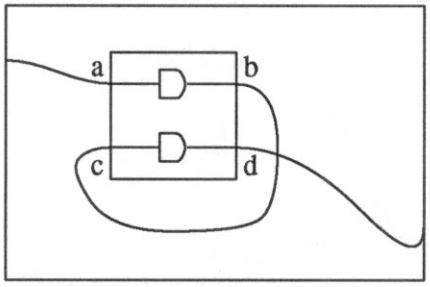

Figure 7.

Output *b* of the submodule is propagated to input *c*. If the internal connections of the submodule were unknown, then this would looks like an apparent cycle. On the other hand, if the module boundaries are erased, it is clear that at the gate level there are no cycles.

Since pseudo-cycles can be simulated easily by the conventional (*N*log*N* space) algorithm, the new, space-efficient algorithm must be made to accept them as well.

A simple approach to evaluation order for the new algorithm would be to consider the instantiations of submodules as single nodes (just like gates) and perform a topological sort on the gates and submodules. Unfortunately, the presence of pseudo-cycles will render this impossible. Instead, we will temporarily abandon the idea of topological sorting; the evaluations of the gates and submodules will be ordered arbitrarily.

Now it is the case that in one evalution pass through the gates and submodules, some nodes will be evaluated before their inputs. The values produced for these nodes will be meaningless. However, it must be the case that some gates have all of their inputs ready — these are the gates connected only to input pins of the entire circuit. Furthermore, the (correct) evaluation of these gates will provide inputs to other gates in the circuit.

If another evaluation pass is made through the circuit, then additional gates will be meaningfully evaluated. Any gate which was meaningfully evaluated on a previous pass must yield the same output, since the inputs to the circuit as a whole have not changed. If there are no cycles in the circuit, then each pass must meaningfully evaluate some gates which previously did not have all their inputs evaluated (unless, of course, all gates have been meaningfully evaluated).

Thus, if a circuit has N gates, then N passes will suffice to propagate meaningful values to its outputs. In many cases, significantly fewer passes will be required.

The simulation algorithm for one module runs as follows:

1. Start with meaningful values at input pins, and arbitrary values elsewhere.
2. Evaluate all gates in an arbitrary but fixed order.
3. Evaluate all submodules in an arbitrary fixed order.
4. If any (gate or module) output values changed in step 2 or step 3, repeat from step 2.

The evaluation of one of the submodules may itself loop until its outputs have settled.

6. Variations on the algorithm

Even though a topological sort is impossible in the presence of loops, an approximation can still be made. For example, the set of strongly connected components can be topologically sorted. (Strongly connected components are those subgraphs in which there is a path from each node to every other; a cycle is an example of a strongly connected component.)

Within a strongly connected component (which, remember, is only pseudo-strongly connected) a feedback vertex set can be removed, and the remaining nodes can be topologically sorted. (A feedback vertex set is a set of nodes (i.e. gates and submodules) whose removal leaves no cycles in the component.)

Since the strongly connected components are the ones which impose a need to evaluate in more than one pass, it makes sense to confine the iteration to the strongly connected components. To evaluate a component, remove (and evaluate) one of the feedback vertices, and then evaluate the remaining subcomponents in topological-sort order. Since the original feedback vertex didn't have entirely meaningful inputs, this must be iterated until the signal values settle.

By confining the repeated evaluation to the pseudo-cycles (where it is needed), the simulation should require fewer gate-evaluations before it converges.

Any module with no pseudo-cycles can be simulated in one pass. In general, it is possible to compute an upper bound b for the number of iterations i any component will require. It may be faster to evaluate b iterations without checking for quiescence, than to iterate for $i+1$ passes

with the quiescence check — even though the actual i may be less than the bound b.

All of the analysis required to reorder the evaluations within a module take space proportional to the size of the module's description. Thus, the algorithm as a whole still uses approximately one bit per wire when the specification is smaller than the number of wires in the circuit.

7. An implementation

Because the layouts of all the instances of a module are the same, and the sequence of gate evaluations can be predetermined, it is possible to compile the gate evaluations into efficient machine code for a von Neumann computer. The individual gate evaluations can be done very quickly.

The use of compiled code to evaluate modules in hierarchical circuits is not new; this is known as "functional modelling"[5]. In the new algorithm, however, the functional model is automatically calculated from the circuit description, which cleanly unifies the functional modelling language with the circuit description language.

This section describes an implementation of the simple algorithm described in section 5 (that is, without any topological sort). The input language is is exactly that described in section 2. An ad-hoc lexical analyzer and a recursive-descent parser are used. A separate symbol tables is kept for each module.

The circuits are compiled into machine code for the VAX. The VAX's bit-field instructions are used to extract and insert binary values from memory. These instructions are expensive; a one-byte-per-wire implementation would use more memory but would run much faster.

Several registers are reserved for special uses:

CV — One register is used to keep track of whether any pin has **C**hanged **V**alue in an iteration of a module.

DP — One register is used to point to the data area (containing the signal values of an instance) of a procedure.

SP — The stack pointer points to the top of the runtime stack, which is used to keep return addresses of procedure calls, and to save copies of register CV.

Here is a summary of how each of the operators is translated:

pin — Individual pins (represented as **ID** or as **ID.ID**) are translated as extract-field instructions. The byte-offset from the data pointer (DP), and the bit-position with the byte, are known at compile-time. The fetched value is put into a register.

\+ — The two subexpressions are evaluated into registers, and then a bitwise-or instruction is used.

* — Similar to +, except that bitwise-and is done.

~ — The single subexpression is evaluated, then complemented.

→ — The two versions of arrow are really the same operator with reversed syntax. In either case, the old value of the destination pin is fetched and exclusive-or'ed with the new

value. This yields a boolean value specifying whether the pin has changed value; this boolean is inclusive-or'ed with register CV. Thus, when the end of a module is reached, register CV contains a 1 if any pin has changed value.

Each module's subroutine must also call the subroutines for the instances of included modules. This is quite straightforward; the data-pointer register (DP) is adjusted, a jump-to-subroutine is done, and then the data-pointer is restored. The subroutine returns a flag indicating whether any internal pins changed value; this is then or'ed into register CV.

In this implementation of the algorithm, an entire boolean expression (and not just one gate) is evaluated before being stored back into the data area. This means that the intermediate "simulated wires" will not require storage space. Thus, a space usage of significantly less than one bit per wire can be achieved.

For technical reasons, the data for each module must begin on a byte boundary. Unused bits are inserted into the data spaces to ensure this.

A four-bit clocked latch module is introduced as a primitive. This has four inputs (d_0, d_1, d_2, d_3) and four outputs (q_0, q_1, q_2, q_3). In each clock cycle, the q_i are treated as constants, and the circuit is evaluated (and modules are iterated) until all pins settle. Then the d_i are copied to the q_i of all latches in the system by a recursive traversal of the symbol table.

8. Performance of the implementation

The two characteristics of performance for any implementation of the algorithm described in this paper are:

1. How fast is the simulation of one gate?
2. How many times must each gate be iterated in a clock cycle?

The first question can be approximated accurately by simply looking at the instruction sequence generated for a gate. In any case, it is a parameter of the implementation and is mostly independent of the circuit being simulated.

The second question is independent of the implementation; it is a function of the circuit being simulated. On the whole, it is a more interesting question than the first, because its answer has implications for any iterative simulation.

Two small but nontrivial circuits were simulated using the implementation described in the previous section. The first circuit implements a chess clock using several sychronous divide-by-*n* counters and some "random logic" state circuitry. The second circuit implements a Turing machine (with finite tape size) using a RAM and a shift-register. In each case, the specifications were built just using the 4-bit latch primitive and *and, or,* and *not* gates.

The measurements were done on a VAX-8600; the results are given below.

	Chess Clock	Turing Machine
Specification (lines)	111	178
Gates	986	17570
Wires	1989	35973
Data space (bytes)	70	968
Spec. space (bytes)	8k	19k
Translation Time (msec)	100	300
Simulation Time (msec)	11	49
Time/Wire (μsec)	1.48	1.34
Iterations	4.07	1.01

The timings above (except the Translation Time) are for each clock cycle of the simulation. "Iterations" measures the total number of wires evaluated divided by the total number of gates in the system, so it is the average number of times each wire is evaluated. This is the parameter of interest; if it is small, then the new algorithm will be competitive and useful; if it very large, then the new algorithm will be much slower than conventional algorithms. For these two example circuits, the number of iterations is small.

"Time/Wire" is just the total runtime divided by the number of wires evaluated; this number should be independent of the circuit simulated, since it just measures the speed of the VAX instructions used to simulate each gate.

"Data space" is the total memory used to store the state of all the wires in the circuit; it is less than one bit per wire because not all wires need their state saved. "Spec. Space" shows the total memory overhead attributable to the specification, including the symbol tables of the modules, and the machine code that implements them. This can be expected to be roughly proportional to the number of symbols in the input specification. There is also a constant space overhead of about 45 kilobytes for the compiler/simulator program itself. The program is implemented in the C language using 1066 lines of code.

For these small examples, the specification size is larger than the data space; but for large, strongly hierarchical circuits, the data space can be expected to dominate the specification size.

9. Worst case execution time

The simulations of the previous section show that the number of iterations required in typical problems is small. However, it is possible to construct artificial examples which require a number of iterations equal to the number of wires in the circuit. This is done by using deeply nested pseudo-cycles.

Define the module A_0 to have two inputs and two outputs, and simply copy the inputs to the outputs. For any $i>0$, we can make the module A_{i+1} using two instantiations of the module A_i, each with a pseudo-cycle (figure 8).

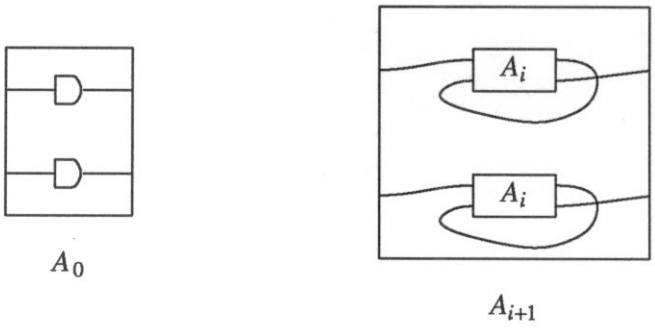

Figure 8

Now it turns out that if all the internal pins have the value 0, and a 1 value is put at the input of a module A_n, then $\Theta(2^k)$ gate evaluations are required (using the algorithm described in this paper) before quiescence is reached. This can be proved using a simple induction argument.

This circuit, however, has $\Theta(2^k)$ wires. Thus, the number of iterations is equal to the number of wires. Thus, evaluating a combinational circuit of n wires using the hierarchical algorithm can take n^2 time; the conventional algorithm takes $O(n)$ time.

Without a topological sort, the conventional algorithm can take n^2 time; the deeply nested nature of the modules A_k prevents the appropriate re-arrangement from taking place. Deeply nested pseudo-cycles are probably rare in real circuits, so that the time will be close to linear in the number of wires (the number of iterations will be small). The empirical results of the preceding section are reassuring.

10. Conclusion

Synchronous digital logic circuits can be simulated in space proportional to one bit per wire, as long as the specification has a hierarchical nature. The simulation algorithm is simple to implement, and runs relatively quickly. Although the algorithm has a quadratic worst-case running time, empirical results show that the running time for typical circuits is close to linear.

It is easy to extend this algorithm to work on multi-bit datapaths instead of single-bit values. It would be harder, though probably possible, to make the simulation event-driven, so that large quiescent parts of the circuit are not re-evaluated on every clock cycle.

References

References

1. N. Giambiasi, A. Miara, and D. Muriach, "SILOG: A practical tool for large digital network simulation," *16th Design Automation Conference*, pp. 263-271, ACM, 1979.
2. Dwight Hill and Willem vanCleemput, "SABLE: A tool for generating structured, multi-level simulations," *16th Design Automation Conference*, pp. 272-279, ACM, 1979.
3. Dan Holt and Steve Sapiro, "BOLT - A block oriented design specification language," *18th Design Automation Conference*, pp. 276-279, ACM, 1981.
4. Mahesh H. Doshi, Roderick B. Sullivan, and Donald M. Schuler, "Themis logic simulator: a mix mode, multi-level, hierarchical, interactive digital circuit simulator," *21st Design Automation Conference*, pp. 24-31, ACM, 1984.
5. Phil Wilcox, "Digital logic simulation at the gate and functional level," *16th Design Automation Conference*, pp. 242-248, ACM, 1979.