Use of a Simulator for Teaching Logic Circuits

R. W. Doran

Abstract

A simulator for introducing the topic of *logic circuits* is described along with the simulator's implications for course content. The simulator is designed for educational purposes rather than as a tool for realistic circuit development. Because it is removed from design detail, the simulator permits a hierarchical approach to circuit description that makes it possible to proceed much further into the subject than is normal in an introductory course. It becomes much more reasonable to structure a course such that its content is suited to the majority of students' interests (i.e. those who will never design practical circuits) and to draw on their existing understanding of programming and serial algorithms.

- -

The LOGIC Simulator

The LOGIC simulator is a program written for the Apple Macintosh that allows logic circuits to be designed and tested. The program is intended to be used for introducing the topic to beginners. The emphasis is thus on simplicity, "ease of use", and introduction of concepts rather than on realism and similarity to existing tools. Combinational circuits are the main concern although basic sequential circuit elements may be demonstrated.

The LOGIC simulator is used in the second-year course in Computer Organization at Auckland. A set of notes introducing the subject of *Logic Circuits* has been developed based on the use of the simulator (see table of contents in the appendix). The simulator itself is provided on a standard Macintosh disk along with a set of example files that support exercises in the notes.

On opening the LOGIC simulator the student is presented with a screen as follows.

1
2
3
4
5
6
blank screen
8
9
10
11
12
13
14
15

The main space on the screen is a work area on which a circuit can be constructed. The circuit is built up from components that are obtained from a menu on the left hand side of the screen. Components may be placed anywhere in the work area. Each component is represented as a rectangle with input stubs on the left and output stubs on the right. Inputs to the entire circuit are on the left of the active area and outputs are to the right. Components may be wired together by clicking on an output and an input in succession. Flow of data through the circuit is generally from left to right.

It is best to try the simulator out to "get the feel of it". By selecting the appropriate components one could quickly come up with an *exclusive or* circuit as follows.

1
2
3
4
5
6
xor initial
8

The only thing that is unreal about the above circuit is the use of *fanout* elements when it is necessary to conect one source to multiple destinations, as is the case with the inputs on the left. Note that the symbols used by the simulator for *and, or* and *not* gates are not the standard. A decision was made to use rectangles partially because, to be honest, it was easier to implement that way. However, a more important reason is, as we shall see, that more-complex circuits are built from composite subcircuits for which the rectangular notation is quite reasonable and there is little sense in regarding elementary circuits as significantly different. The *and* and *or* gates are labeled with their logic symbols and *not* is shown by a bubble as is usual in circuit diagrams.

After a circuit is constructed it may be simulated. Before simulation the circuit should be reset by selecting the *reset* option in the test menu. (Truth to tell, *reset* actually constructs a representation of the circuit to be simulated, making copies of replicated subcircuits). Inputs to the circuit can by set to true (black) and false (grey) by clicking on the input boxes. The simulation occurs when *go* is selected from the test menu or, more usually, *control-g* is typed. Truth and falsity may be seen to flow down the wires to eventually set the output indicators. If desired, the simulation may be performed at a slower speed to see what components are doing. The resulting screen after one stage of simulation could be as follows:

1
2
3
4
5
6
exclusive or simulation
8

Any circuit may be used to construct other circuits. It is thus possible to provide students with an initial set of subcircuits to use as components. In the following screen a master/slave D flip/flop has been provided and a shift register then constructed. Note that, although wiring is initially mindless point-to-point', provision is made for manual modication of a circuit to give a neat layout. Note also that comments may be placed in the circuit to make it easier to understand.

1
2
3
4
5
6
shift register
8
9
10
11
12
13
14

Any component may be re-selected and modified by "double clicking" in the component menu or diagram. In the example above we could select the clocked D flip/flop from which the master slave is constructed. In this case, because the circuit involves feedback, it is necessary to have wires in which data flows from right to left. LOGIC makes no attempt to draw these but merely represents them as labelled stubs (labelled *C* and *D* in this example). The D flip/flop has a special "reset" line. This is necessary because the lockstep simulation of LOGIC can cause circuits to be unstable unless they are properly initialised.

1
2
3
4
5
6
7
blank screen
8
9
10
11
12
13

The LOGIC simulator is exceedingly simple; it is quickly mastered by students so that they can, with little delay, proceed to use the simulator to learn about logic circuits. The significance of a simulator is not its intrinsic merit but the added freedoms it gives over standard hardware and software trainers:

- students are freed from dealing with the nasty properties of real circuits such as loose wires etc

- the simulator can be used to teach large classes with little supervision

- much larger circuits can be simulated than normally possible

- hiearchical design techniques analaogous to those used in programming can be taken advantage of

- the visible flow of change through the circuits allows analogy with flow of control in programs

Perusal of the table of contents of the notes accompanying the simulator (see appendix) will show how far it is possible to take beginners. However, the last two advantages above are worthy of more explanation because they do make a significant difference to the way the subject is taught. We will expand on these topics below.

Hierarchical Design

With conventional hardware design there are only two levels of structure, namely the *components* provided and the *circuits* constructed from them. The components are usually very limited in type.

Despite these restrictions the circuit designer does make use of hierarchical abstraction. Many levels of circuit may be invented in order to make the device being designed easy to comprehend. However, when it comes to implementing the circuit, or testing it with a logic trainer, all such structure has to be removed.

With the LOGIC software simulator, abstraction and structure may be used relatively freely. The instructor is not constrained when providing components for use by students. In the following example exercise where the student is asked to construct a circuit for adding signed magnitude numbers, the instructor has provided a simple 6-bit binary adder as a component:

1
2
3
4
5
6
set up for sadder
8
9
10
11
12
13
14

The adder component is called *2add6*. Other components, *fadd* a full adder, and *xor* a two-input exclusive-or, that were needed to construct the simple adder are also available to the student.

The student designer can immediately use *2add6* and *xor* but is not constrained to just these. New components can be created as needed but do not have to be fully defined. In this example there is need for a subcircuit *comp6* that complements its input and *sgnov* that forms the sign of the result and the overflow indicator. The subcircuit *comp6* is needed in two places in the solution and can be defined temporarily as follows:

1
2
3
4
5
6
gutless complementor
8
9
10
11
12
13
14

Here the inputs are tied to tacks temporarily. This component, and others, can then be used to construct the top level of the solution (note that an unconnected is treated as 0):

1
2
3
4

- -

5

6

exclusive or simulation

8

9

10

11

12

13

14

Before the correctness of this can be tested, the components have to be fully defined. The complementor *comp6* could be expanded to:

1
2
3
4
5
6
exclusive or simulation
8
9
10
11
12
13
14

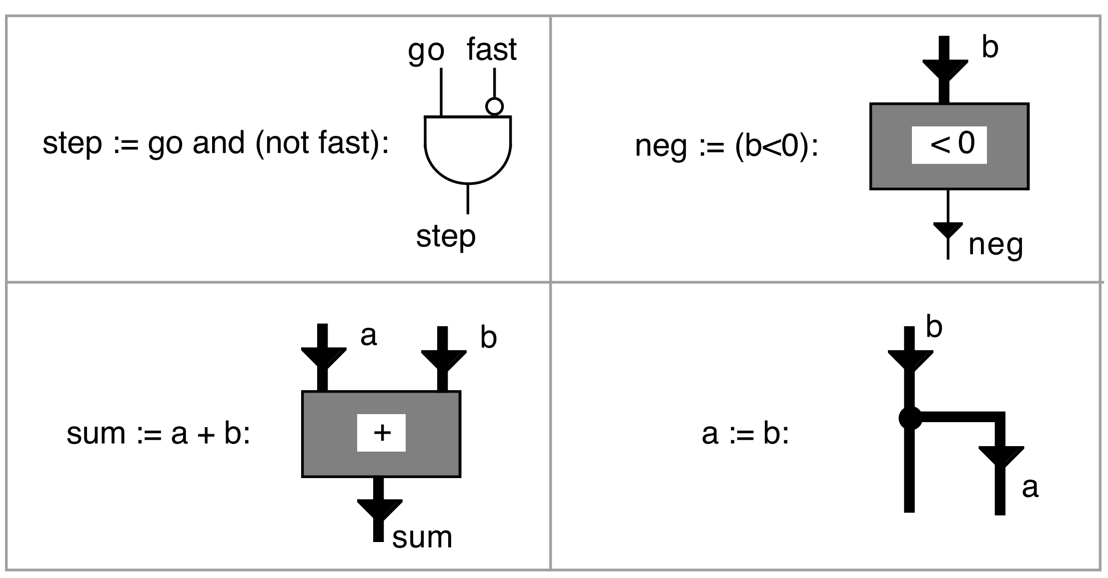
In this case it has been found useful to construct a further component *gate6* comprising a set of six and-gates with common control.

As this example should have illustrated, the process of design is changed to use the freedom that the simulator provides. This approach is somewhat unrealistic because in reality the designer is still constrained by available components. However, the simulator allows the beginner to ignore such problems and concentrate on understanding the circuit being designed, using abstraction and structuring just as in programming. Indeed, the limited size of the simulator screen forces circuits to be structured if they are to fit at all (rather like the guideline of requiring subroutines to fit on one screen).

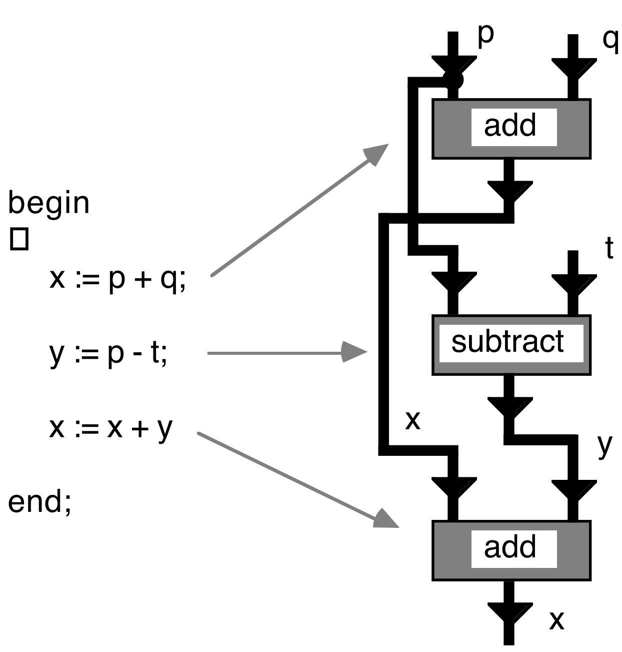
Circuits as Realization of Algorithms

Not only does a simulator like LOGIC allow the use of abstraction and structuring as used in all problem solving, but it also permits the analogies between algorithms and circuits to be explored to a far greater depth. It is possible to show how data flow in a circuit can be used to mimic the flow of control in programs. One can convince students that if a design problem can be solved with an algorithm then a circuit can be produced to do the same job, the transition from algorithm to circuit being almost automatic.

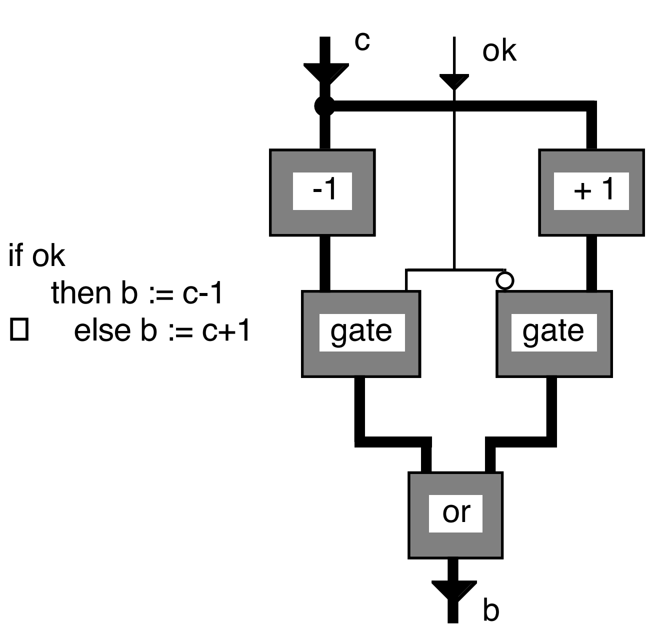
The notes prepared for use with LOGIC show one approach to introducing the program-circuit analogy. Firstly, simple circuits are given for representing atomic programming statements. Variables are mapped to labelled lines, operators to circuit elements, and assignments to connections between wires:



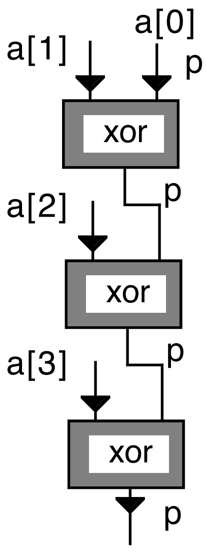
Next, appropriate connection of wires is used to get the effect of a compound sequence of statements:

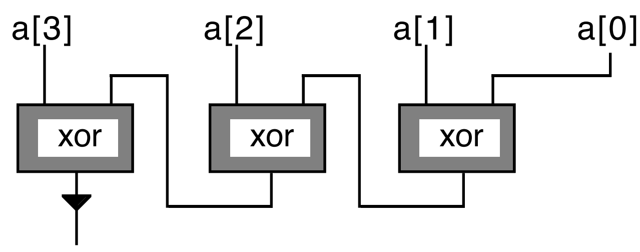


Then come conditional statements. These take a little more thought. The general rule is straightforward but reasonably complex and there are many simple cases to watch out for:

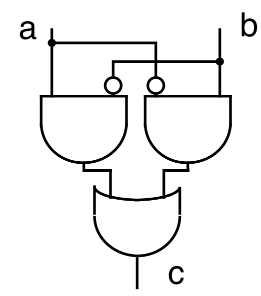


Things get interesting at the next stage where repetitive structures are considered. In a *combinational* circuit there is no possibility of a section of a circuit being used for more than one purpose. Consequently, all sections of programs that are used, by repetition or by procedure call, have to be expanded like macros before they can be converted to circuits. With algorithms that repeat for each element of an array such expansion gives rise to *ripple*-type circuits such as a ripple adder or the following parity circuit (where use is made of an *xor* component with definition as in the procedure):






```
procedure xor(a,b; var c);
begin
      c := (a and (not b))
           or ((not a) and b)
□□□    end;
```



Where the LOGIC simulator really shows its use is that it allows us to explore the relationship between recursive algorithms and the practical fast tree-circuits. For example, in the following LOGIC screen a width-8 incrementor has been expressed in terms of two identical incrementors of width-4 and a subcircuit that selects one of two 5-bit input words based on a control input. The width-4 incrementors can be designed in terms of width-2 incrementors, which are expressed in terms of single bit incrementors which are finally defined in terms of basic logic gates.

1
2
3
4
5
6
width-8 incrementor
9
10
11
12
13
14

1
2
3
4
5
6
width-1 incrementor
8
9
10
11
12
13
14

Summary

The above brief outline should have given some feeling for what the LOGIC simulator does and the effect that it can have on the content of a course in computer organization. As well as effecting its own subject matter, a simulator such as LOGIC acts as a reinforcement of programming principles, practices,and techniques for the advancing student. The ideas of structured programming are shown to have immediate application elsewhere and the important concept of recursion is shown to underly the hardware of computers as well as software (of particular interest is that the recursive binary search algorithm is analogous to the tree decoder).

The simulator has now been used for a couple of years and has shown itself to be a useful tool for stimulating student interest.

Acknowlegements

The simulator was programmed by students at Auckland University Computer Science Department, Bruce Palmer as part of his MSc thesis in the first instance, then extended by Bruce Benson and Max Nilson.

Th analogy between recursive algorithms and tree-circuits was described initially in "Recursive Algorithms in Combinational Circuit Design" by R W Doran and LK Thomas, Massey University Computer Unit, Report No. 13, December 1974.

Ordering Information

Information on ordering the LOGIC simulator and notes may be obtained from:

The Department of Computer Science,

Auckland University,

Private Bag, Auckland, New Zealand.

Table of Contents of Introduction to Logic Circuits

8.
Fast Practical circuits.

Recursive definition. Parity circuits. Tree decoders and dual-tree decoders. Shifters. Multipliers. Adders.

9. **Sequential Circuit Elements.**

RS Flip/flops. State transition diagrams. Clocked flip/flops. Timing diagrams. Master/slave flip/flops. JK and T flip/flops. The D flip/flop.

10. **Simple Sequential Circuits**.

Asynchronous circuits. Synchronous circuits. Reduction to combinational design. Simple serial circuits.

11. **Parallel Circuit Components**.

Register arrays. Shift registers. Multi-function registers. Control programs. Serial adder. Multiple-function combinational circuits.

12.
Miscellaneous Logic Circuits.

Sorting circuits. Cellular circuits and division. Circuits with 1 *not*. Combinational circuits with feedback.