Digsim: The User Guide

by

1. Publication Information

Date published: 25 March 2004

2. Introduction

The *digsim* application simulates digital circuits. (The application is implemented in the Java programming language and has been used to introduce electrical engineering students to object-oriented analysis, design and implementation. The source code and design documentation are available.) However, this document describes the user interface only; no knowledge of *digsim*'s inner workings is required or assumed.

2.1. Ensure that digsim is installed

Start a *digsim* session with the command: digsim

If the *digsim* application has been installed, you will see the message (on *stderr*): *Welcome to digsim...We await your commands.*

Terminate the session by entering the `end` command. (*digsim* will respond with *All done*.)

2.2. First simulation

The first circuit to simulate is a 2-input NAND gate that evaluates the Boolean expression:

```
Z = not(AB)
```

Invoke *digsim* and enter the command:

```
Nand A B Z
```

Next enter the command: display * The following is displayed:

```
Node "A": 0
```

```
Node "B": 0
Node "Z": 1
```

To terminate the simulation session, type in the command `end`.

3. Clocked D-Latch circuit

We now examine a more complex circuit (a clocked D-latch based on nand gates) and its simulation. The following annotated listing shows how the circuit is described (using a spice-like description language) and then simulated.

3.1. Clocked D-Latch

```
//Circuit description
Nand D=1 Dbar (delay=5)
Nand D Clk=0 Sbar //default 10ns delay
Nand Dbar Clk Rbar (delay=15)
Nand Sbar Qbar Q=1 (delay=20)
Nand Rbar Q Qbar (delay=25)

display *
trace *
set Clk 1 at 10
set D 0 at 100
set Clk 0 at 200
step 100
//The next line is illegal
set Clk 1 at 100
set Clk 1 in 100
set D to 1 in 90
step 50
```

The "//" string can be used anywhere as a comment; the rest of the line is ignored.

The default delay can be overriden with the delay parameter.

The initial value of a node can be specified using the equals (=) sign.

Blank lines can be used for readability.

The `display` command prints the current value of the named nodes. The wildcard character (*) can be used to specify all nodes.

The `trace` command prints any change to a Node during a simulation.

The `set` command schedules a value for a Node at a specific (future) time.

The `step` command simulates the circuit for at most n Node changes.

This `set` command is illegal because it attempts to set the value for a Node in the past. (While we all sometimes wish we could change the past, *digsim* is not a “Back to the Future” machine.)

3.2. Output

Most commands produce no output. (Syntax error messages are printed to *stderr*.) Two commands always produce some output: `display` *nodes*

Immediately prints to *stdout* the values of the named nodes.

`step` *num*

Prints results of any trace commands. Always prints a summary.

Note that the `trace` *node(s)* command does not produce output immediately; however, it specifies which Nodes should print any value changes when the simulator is invoked with the `step` command.

3.2.1. Clocked D-Latch example output

The first command in the previous example that produces output is the `display *` command on line 8. The output is:

```
Node "Clk": 0
Node "D": 1
Node "Dbar": 0
Node "Q": 1
Node "Qbar": 0
Node "Rbar": 1
Node "Sbar": 1
```

The next command that produces output is the `step 100` command at line 13. The `step` *n* command simulates at most *n* Node changes. The Node changes may be either previously explicitly specified changes (with the `set` command) or derived changes deduced by the simulator. In the present case, the circuit settles down to a stable and consistent configuration after 10 Node changes. (We use the term “Event” to mean a change in the value of a Node.) The last line of the output gives the total number of Events that were processed and the number of scheduled Events that have not yet been simulated. The output is:

```
"Clk" set to 1 at 10
"Sbar" set to 0 at 20
"D" set to 0 at 100
"Dbar" set to 1 at 105
"Sbar" set to 1 at 110
"Rbar" set to 0 at 120
"Qbar" set to 1 at 145
"Q" set to 0 at 165
```

```
"Clk" set to 0 at 200
"Rbar" set to 1 at 215
Events processed: 10; Events pending: 0
```

The final block of output is triggered by the `step 50` command at line 18. The output is:

```
"D" set to 1 at 305
"Dbar" set to 0 at 310
"Clk" set to 1 at 315
"Sbar" set to 0 at 325
"Q" set to 1 at 345
"Qbar" set to 0 at 370
Events processed: 6; Events pending: 0
```

4. Block macros

Complex digital circuits can be defined in terms of other blocks using the `defBlock` construct.

Suppose we need a 2-input OR gate and wish to construct it from NAND gates. Boolean algebra (DeMorgan's laws) tell us that we can turn a 2-input NAND gate into an OR gate by inverting its inputs. This idea is expressed with the following macro definition:

```
defBlock or2 a b out
Nand a aBar
Nand b bBar
Nand aBar bBar out
enddef
```

Note that you do not type in macros like this directly to the simulator. Rather, you place them in a file and then load the file into the simulator.

Suppose that the macro definition above were in a file called `or2.block`. A sample simulator session is given below. Note the use of the `load` *file* command used to inform the simulator of the macro. (In the session sample below, this font indicates user input and another `font` is used for `computer output`.

```
load or2.block
or2 A B Z
display A B Z
Node "A": 0
Node "B": 0
Node "Z": 0
or2 AA=1 BB CC
display AA BB CC
Node "AA": 1
Node "BB": 0
Node "CC": 1
set AA 0 in 100
```

```
trace *
step 20
"AA" set to 0 at 100
"or2.2.aBar" set to 1 at 110
"CC" set to 0 at 120
Events processed: 3; Events pending: 0
```

A block can be defined in terms of primitive circuit elements (such as Nand gates) supported directly by the simulation engine or in terms of previously defined macro blocks. For example, the file `or.blocks` contains definitions for both 2-input and 3-input OR gates as follows:

```
defBlock or2 a b out
Nand a aBar
Nand b bBar
Nand aBar bBar out
enddef

//Define a 3-input OR gate in terms of the or2 macro
defBlock or3 a b c out
or2 a b tmp
or2 tmp c out
enddef
```

A sample session using this macro definition file is shown below:

```
load or.blocks
or3 P Q R z
display P Q R z
Node "P": 0
Node "Q": 0
Node "R": 0
Node "z": 0
set P to 1 at 100
trace *
step 20
"P" set to 1 at 100
"or2.1.aBar" set to 0 at 110
"or3.1.tmp" set to 1 at 120
"or2.2.aBar" set to 0 at 130
"z" set to 1 at 140
Events processed: 5; Events pending: 0
```

5. That's all folks...(for now)...

This document is *not* a complete user guide. Alas, the user has to infer the meaning of various commands (and options) from the examples given here. In addition, not all features are described. For additional information, your only option to UTSL.
[1]

Nonetheless, I hope that these simple examples may be useful to students in COE618 and ELE428. COE618

The user-interface presented here is based on the simulation engine analyzed in the course. No changes have been made to the `digsim` package; this user-interface is implemented entirely in the separate `digsim.ui` package. The source code is available.

ELE428

Labs 7 and 8 have examined the architecture and some of the implementation of a digital circuit simulator written in C. Although *digsim* is written in Java rather than C, the overall architecture of the software is basically the same as the one you have studied. Note that the user interface in *digsim* goes well beyond the requirements in Lab 8. Note also that while you are welcome to look at the source code of *digsim*, it is unlikely that you could just cut and paste various code snippets to complete Lab 8. (In particular, although both the Java and C versions of the simulation engine have a very similar architecture, the software that implements the user interface in Java has a very different structure than the simpler one suggested for the C implementation in Lab 8.)

6. Footnotes

1) The *Star Wars* inspired acronym UTSL means “Use the Source Luke”. (*May the source be with you...*) However, you don't need to use the actual source, the API (Application Programming Interface) documentation is sufficient. (I cannot suggest that you RTFM—Read the F...ing Manual—since this document *is* is the frigging manual.)