ISTD 50.002 Computation Structures

Zhang Yue
Oka Kurniawan
Wei Lu

Original creator: Lozano-Perez, Tomas, using materials originally developed by Christopher J. Terman and Stephen A Ward. Course materials for ISTD 103, Computation Structures. MIT-SUTD Collaboration, 2012.

Modified by: Oka Kurniawan, 2013.

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

ISTD 50.002 Computation Structures
JSim

Contents:

1. Introduction to JSim

JSim is a computer-aided design (CAD) tool that provides:

- A simple editor for entering a circuit description (called a "netlist")
- A choice of programs to simulate your circuit
- A waveform browser to view the results of a simulation

JSim uses mathematical models of circuit elements to make predictions of how a circuit will behave both statically (DC analysis) and dynamically (transient analysis). The model for each circuit element is parameterized, with each parameter providing information about the design or physical properties of the device. You'll specify the design parameters in your netlist, e.g., for a mosfet you would specify its length and width. The parameters specifying physical properties have been derived from measurements taken at the integrated circuit fabrication facility. We'll provide those parameters as part of the mosfet model.

2. Running JSim

Running standalone

First, you'll need to install a Java system. The Sun Java Runtime Environment, Standard Edition (J2SE), for Linux, Solairs and Windows can be downloaded from http://java.sun.com. J2SE for Mac OS X is available from http://www.apple.com/java. On Linux, you'll want to change your PATH environment variable so that the "java" command is on your search path. On Windows and OS X, double-clicking any of the 50.002 jar files will run the program (assuming you've installed the Sun Java environment).

Once you've installed a Java environment and downloaded the 50.002 files, you can run JSim using the following command:

```
java –jar jsim.jar –Xms8m –Xmx32m –reporterrors file…
```

You may have to specify complete pathnames for "java" and "jsim.jar" depending on your current search path and working directory. Each of the arguments is explained below.

- **-jar jsim.jar** adds the java archive `jsim.jar` to the list of files Java examines when trying to find classes. `jsim.jar` contains the classes used by jsim for displaying/editing netlists, running simulations and browsing the results.

  If you get an error of the form "`Exception in thread "main" java.lang.NoClassDefFoundError: jsim/JSim`", the Java runtime didn't find the jsim.jar file -- try giving its full pathname, e.g., `C:\50.002\jsim.jar` or whatever is appropriate for your installation.

- **-Xms8m -Xmx32m** sets the minimum heap size to 8MB and the maximum heap size to 32MB. Starting JSim with a generous heap allocation avoids a lot of garbage collection overhead the first time your circuit is processed. If you run out of memory, try specifying the **-no-local-names** option when running JSim. This will greatly reduce the size of the node name hashtable JSim constructs when processing the netlist. The downside of using this option is that nodes can only be referred to by using their name in the (sub)circuit where they were first defined.

- **-reporterrors** asks JSim to provide a backtrace whenever it encounters an internal error. In the unlikely event an error occurs, it would be very helpful if you can email this backtrace to 6004-labs@lists.csail.mit.edu.

- **file...** are optional arguments specifying one or more JSim netlist files.

Editing netlists

The netlist editor built into JSim is based on the JTextArea class in Swing. Many people find the editing facilities provided by this class to be underwhelming and prefer to use an external editor. Jsim.el (available from the Courseware webpage) defines a new major mode for EMACS useful for editing JSim netlists. You can invoke the mode automatically when reading in a ".jsim" file by adding the following to your .emacs file:

```
;;; jsim support, assumes jsim.el lives in your home directory
(autoload 'jsim-mode "~/jsim" nil t)
(setq auto-mode-alist (cons '("\.jsim$" . jsim-mode) auto-mode-alist))
(add-hook 'jsim-mode-hook 'turn-on-font-lock)
```

JSim toolbar

There are various handy buttons on the JSim toolbar:

Exit. Asks if you want to save any modified file buffers and then exits JSim.

New file. Create a new edit buffer called “untitled”. Any attempts to save this buffer will prompt the user for a filename.

Open file. Prompts the user for a filename and then opens that file in its own edit buffer. If the file has already been read into a buffer, the buffer will be reloaded from the file (after asking permission if the buffer has been modified).

Close file. Closes the current edit buffer after asking permission if the buffer has been modified.

Reload file. Reload the current buffer from its source file after asking permission if the buffer has been modified. This button is useful if you are using an external editor to modify the netlist and simply want to reload a new version for simulation.

Save file. If any changes have been made, write the current buffer back to its source file (prompting for a file name if this is an untitled buffer created with the “new file” command). If the save was successful, the old version of the file is saved with a “.bak” extension.

Save file, specifying new file name. Like “Save file” but prompts for a new file name to use.

Save all files. Like “save file” but applied to all edit buffers.

3. Netlist format

Input to JSim is processed line-by-line. Fields within a line are separated by whitespace (blanks or tabs), a comma, an "=" (equal sign), or a left or right parenthesis; extra spaces are ignored. Blank lines are ignored during processing.

The circuit to be simulated is described to JSim by a set of element statements, which define the circuit topology and element values, and a set of control cards, which define the model parameters and the simulation controls. The order of the statements is arbitrary (except, of course, that continuation lines must immediately follow the line being continued).

Continuation lines

A line may be continued by entering a "+" (plus) as the first character of the following line. JSim will continue reading continuation lines starting with the second character. There's no limit to the number of continuation lines allowed. Example:

```
.MODEL NENH NMOS LEVEL=3 PHI=0.700000 TOX=9.4000E-09
+ XJ=0.200000U TPG=1 VTO=0.6746 DELTA=1.1480E+00 LD=3.4510E-08
+ KP=1.8217E-04 UO=495.9 THETA=1.7960E-01 RSH=3.2470E+01
```

Comments

Lines whose first character is "*" (asterisk) are treated as comments and ignored during processing. Note that comments can have continuation lines (see above). Examples:

```
* Both of the following lines are treated as a comment
*.MODEL NENH NMOS LEVEL=3 PHI=0.700000 TOX=9.4000E-09
+ KP=1.8217E-04 UO=495.9 THETA=1.7960E-01 RSH=3.2470E+01
```

You can also add comments at the end of a line by preceding the comment with the characters "//" (C++- or Java-style comments). All characters starting with "//" to the end of the line are ignored. Any portion of a line or lines can be turned into a comment by enclosing the text in "/*" and "*/" (C-style comments). Examples:

```
R1 A B 3k   // 3K-ohm resistor between nodes A and B

/* use when testing at high temps!  But not now…
.temp 125
*/
```

Names

Each device and node in the circuit has a unique name. A name can be either a simple integer (0 is reserved as the name of the ground node) or an alphanumeric string. The string is sequence of characters consisting of letters, digits, "_", ":", "$", "[", "]", and ".". Strings cannot begin with a digit and names begining with "$" are reserved for naming built-in devices. "." (period) is reserved for use as a separator in hierarchical names. Examples:

```
R1
stdcell:and
This_is_a_very_long_name
cpu.alu.adder.bit31.carry_in
```

When naming a collection of nodes that serve as a bus it is convenient to use iterators to save on having to enter many names sequentially. Iterators have the form

prefix[*start*:*end*]*suffix*

where *start* and *end* are integers specifying the first and last indicies of the sequence respectively. JSim expands the iterator into |*start-end*| + *1* names substituting a different value for the bracketed expression in each iteration. Examples:

```
data[7:0]   =>  data7 data6 data5 data4 data3 data2 data1 data0
xxx[1:4]yyy =>  xxx1yyy xxx2yyy xxx3yyy xxx4yyy
```

One can also specify an increment between successive indicies using the form:

prefix[*start*:*end*:*inc*]*suffix*

Example:

```
data[7:0:2]  =>  data7 data5 data3 data1
```

More than one iterator can be used in a name; they are expanded from left to right:

```
a[3:0][1:0]  =>  a3[1:0] a2[1:0] a1[1:0] a0[1:0]
             =>  a31 a30 a21 a20 a11 a10 a01 a00
```

Note that using an iterator is exactly equivalent to specifying the expanded sequence:

```
* The following two element statements are equivalent
X1 out in[7:0] $xor
X1 out in7 in6 in5 in4 in3 in2 in1 in0 $xor
```

Finally, there's a shortcut for specifying multiple copies of the same node:

```
foo#4     =>   foo foo foo foo
x[1:0]#2  =>   x[1:0] x[1:0]      =>    x1 x0 x1 x0
```

Numbers

A number may be an integer (`12, -44`), a floating point number (`3.14159`), an integer or floating point number followed by an integer exponent (`1E-14, 2.65E3`), or an integer or a floating point number followed by one of the following scale factors:

Scale Factor	Pronounced	Multiplier
T, t	tera	1E12
G, g	giga	1E9
MEG, meg	mega	1E6
K, k	kilo	1E3
M, m	milli	1E-3
U, u	micro	1E-6
MIL, mil		25.4E-6
N,n	nano	1E-9
P, p	pico	1E-12
F, f	femto	1E-15

Letters immediately following a number that are not scale factors are ignored and letters immediately following a scale factor are ignored. Integers can be entered in binary, octal or hexadecimal notation by using the appropriate prefix:

0b1011101110100	6004 in binary (“0b” prefix)
013564	6004 in octal (“0” prefix)
0x1774	6004 in hex (“0x” prefix)

Examples:

```
 * The following all represent the same numeric value
1000  1000.0  1000Hz  1E3  1.0E3  1kHz  1K 0x3E8 01750 0b1111101000
```

4. Device statements

Each device in the circuit is specified by a device statement that specifies the device name, the circuit nodes to which the device is connected, and the values of the parameters that determine the electrical characteristics of the element. The first letter of the device name specifies the element type, the remainder of the name can be any legal name (see above). For example, a resistor name must begin with the letter "R" or "r". Hence "R1", "rse", and "R3ac_2xy" are valid resistor names. Device names must be unique at the level of circuit in which they appear.

In the following description, data field enclosed in braces ("{" and "}") are optional. All indicated punctuation (parenthesis, equal signs, etc.) are required.

With respect to branch voltages and currents, JSim uniformly uses the associated reference convention: current flows in the direction of voltage drop. The circuit cannot contain a loop of voltage sources and/or inductors and cannot contain a cutset of current sources and/or capacitors. Each node in the circuit must have a dc path to ground.

```
Rid n+ n- value              // resistor, units in ohms, value > 0
Cid n+ n- value              // capacitor, units in farads, value >= 0
Lid n+ n- value              // inductor, units in henries, value >= 0
```

Linear devices. "n+" and "n-" are the two terminal nodes and "value" is the specified resistance, capacitance, or inductance. Examples:

```
R1 1 2 1k                // 1k ohm resistor between nodes 1 and 2
Rbias base 0 33ohms      // 33 ohm resistor between nodes base and ground
CMILLER gate drain 1fF   // 1 femtofarad capacitor
Lshunt a b 10u           // 10 microhenry inductor
```

```
Mid nd ng ns nb model L=number W=number {params}     // mosfet, units in meters
Mid nd ng ns nb model SL=number SW=number {params}   // mosfet, scaled units
```

Mosfets.

The other new twist introduced in the example netlist is the use of symbolic dimensions for the mosfets ("SW=" and "SL=") instead of physical dimensions ("W=" and "L="). Symbolic dimensions specify multiples of a parameter called SCALE, which is also defined in nominal.JSim:

```
.option SCALE=0.6u
```

```
Xid nodes.. subckt {params}      // instance of user-defined subcircuit
```

Subcircuit instance.

```
Vid n+ n- {{DC=}dcvalue} {tran}  // voltage source, units in volts
Iid n+ n- {{DC=}dcvalue} {tran}  // current source, units in amperes
```

Independent sources. *n+* and *n–* are the positive and negative nodes respectively. Note that voltage sources need not be grounded. Positive current is assumed to flow from the positive node, through the source, to the negative node. A current source of positive value will force current to flow out of the *n+* node, through the source, and into the *n–* node. Voltage sources, in addition to be used for circuit excitation, are the "ammeters" for JSim, that is, zero-value voltage sources may be inserted into the circuit for the purpose of measuring current. They

will, of course, have no effect on circuit operation since they represent short-circuits.

The *dcvalue* is used during DC and OP analyses and the initialization phase of TRAN analysis. If no dc specification is provided, a value of 0 is assumed.

Any source can be assigned a time-dependent value for transient analysis. If a source is assigned a time-dependent value, the t = 0 value is used in place of any specified dc value. There are six source functions: pulse, exponential, sinusoidal, piece-wise linear, amplitude modulation and single-frequency frequency modulation. Each source function takes several parameters that determine the shape of the waveform; the form of the specification can be any of the following:

```
fname param1 param2 ... paramN
fname param1, param2, ... paramN
fname(param1 param2 ... paramN)
fname(param1, param2, ... paramN)
```

i.e., the parenthesis and commas are optional. The following paragraphs provide a detailed description for each of the source functions.

more here…

```
Eid n+ n- ctl+ ctl- gain  // voltage-controlled voltage source (VCVS)
Fid n+ n- ctl+ ctl- gain  // current-controlled current source (CCCS)
Gid n+ n- ctl+ ctl- gain  // voltage-controlled current source (VCCS)
Hid n+ n- ctl+ ctl- gain  // current-controlled voltage source (CCVS)
```

Dependent sources.

more here…

`Wid nodes… nrz(vlow,vhigh,tperiod,tdelay,trise,tfall) data…`

The "W" voltage source that generates digital waveforms for many nodes (e.g., a bus) at once. If N nodes are specified, think of them as an N-bit value where the node names are listed most-significant bit first. The "W" source will set those nodes to a sequence of data values using the data specified at the end of the "W" statement. At each step of the sequence, the N low-order bits of each data value will be used to generate the appropriate voltage for each of the N nodes. The voltage and timing of the signals is given by the nrz parameters:

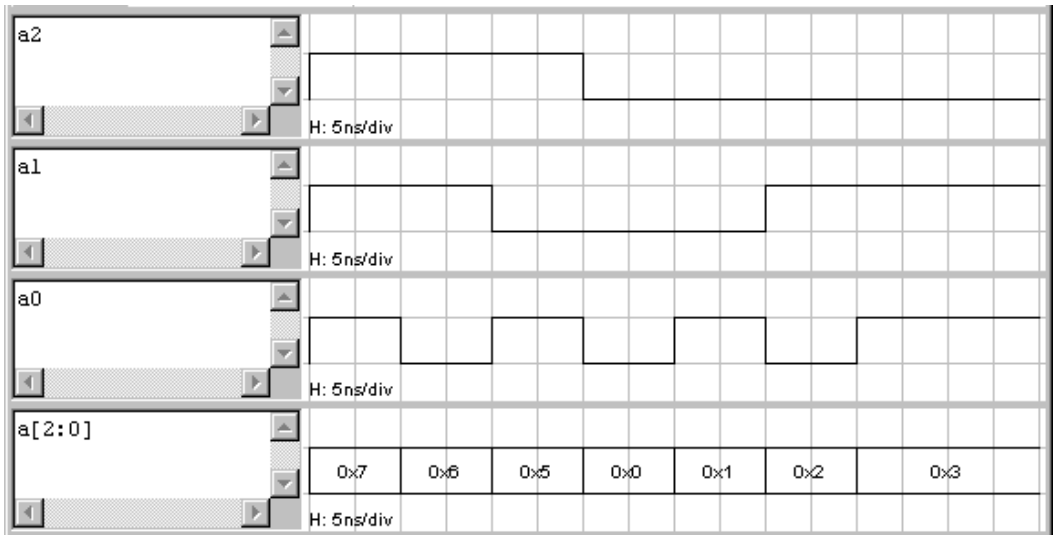
vlow	voltage used for a logic low value (usually 0)
vhigh	voltage used for a logic high value (usually 3.3V)
tperiod	interval (in seconds) at which values will be changed
tdelay	initial delay (in seconds) before periodic value changes start
trise	rise time (in seconds) for low-to-high transitions
tfall	fall time (in seconds) for high-to-low transitions

Note that the times are specified in seconds, so don't forget to specify the "n" scale factor when entering times! Example:

```
.include "/mit/6.004/jsim/nominal.jsim"
Wtest a[2:0] nrz(0v,3.3v,10ns,0ns,.1ns,.1ns) 7 6 0x5
+ 0 1 0b010 3
.tran 80ns
.plot a2
.plot a1
```

```
.plot a0
.plot a[2:0]
```

Note that you can specify data values in decimal, hex ("0x" prefix), octal ("0" prefix) or binary ("0b" prefix). The last data value is repeated if necessary. This example produces the following plot when run using the gate level simulator



JSim knows how to plot a set of nodes as a single multi-bit data value, as shown in the bottom channel of the plot above. If you zoomed in on one of the transition times, you would see that the values actually turn to "X" for 0.1ns while making the transition between valid logic levels.

5. User-defined subcircuits

```
.subckt name terminals…
* internal circuit elements are listed here
.ends
```

The ".subckt" statement introduces a new level of netlist. All lines following the ".subckt" up to the matching ".ends" statement will be treated as a self-contained subcircuit. This includes model definitions, nested subcircuit definitions, electrical nodes and circuit elements. The only parts of the subcircuit visible to the outside world are its terminal nodes which are listed following the name of the subcircuit in the ".subckt" statement:

Once the definitions are complete, you can create an instance of a subcircuit using the "X" circuit element:

```
Xid nodes… name
```

where *name* is the name of the circuit definition to be used, *id* is a unique name for this instance of the subcircuit and *nodes*... are the names of electrical nodes that will be hooked up to the terminals of the subcircuit instance. There should be the same number of nodes listed in the "X" statement as there were terminals in the ".subckt" statement that defined *name*. For example, here's a short netlist that instantiates 3 NAND gates (called "g0", "g1" and "g2"):

```
.global vdd
Vdd vdd 0 3.3V

* 2-input NAND: inputs are A and B, output is Z
.subckt nand2 a b z
MPD1 z a 1 0 NENH sw=8 sl=1
MPD2 1 b 0 0 NENH sw=8 sl=1
MPU1 z a vdd vdd PENH sw=8 sl=1
MPU2 z b vdd vdd PENH sw=8 sl=1
.ends

Xg0 d0 ctl z0 nand2
Xg1 d1 ctl z1 nand2
Xg2 d2 ctl z2 nand2
```

The node "ctl" connects to all three gates; all the other terminals are connected to different nodes. Note that any nodes that are *private* to the subcircuit definition (i.e., nodes used in the subcircuit that don't appear on the terminal list) will be unique for each instantiation of the subcircuit. For example, there is a private node named "1" used inside the nand2 definition. When JSim processes the three "X" statements above, it will make three independent nodes called "xg0.1", "xg1.1" and "xg2.1", one for each of the three instances of nand2. There is no sharing of internal elements or nodes between multiple instances of the same subcircuit.

It is sometimes convenient to define nodes that are shared by the entire circuit, including subcircuits; for example, power supply nodes. The ground node "0" is such a node; all references to "0" anywhere in the netlist refer to the same electrical node. The example netlist defines another shared node called "vdd" which is used whenever a connection to the power supply is required.

JSim makes it easy to specify multiple gates with a single "X" statement. You can create multiple instances of a device by supplying some multiple of the number of nodes it expects, e.g., if a device has 3 terminals, supplying 9 nodes will create 3 instances of the device. To understand how nodes are matched up with terminals specified in the .subckt definition, imagine a device with P terminals. The sequence of nodes supplied as part of the "X" statement that instantiates the device are divided into P equal-size contiguous subsequences. The first node of each subsequence is used to wire up the first device, the second node of each subsequence is used for the second device, and so on until all the nodes have been used. For example, since xor2 has 3 terminals:

```
Xtest a[2:0] b[2:0] z[2:0] xor2
```

is equivalent to

```
Xtest#0 a2 b2 z2 xor2
Xtest#1 a1 b1 z1 xor2
Xtest#2 a0 b0 z0 xor2
```

There is also a handy way of duplicating a signal: specifying "foo#3" is equivalent to specifying "foo foo foo". For example, xor'ing a 4-bit bus with a control signal could be written as

```
Xbusctl in[3:0] ctl#4 out[3:0] xor2
```

which is equivalent to

```
Xbusctl#0 in3 ctl out3 xor2
Xbusctl#1 in2 ctl out2 xor2
Xbusctl#2 in1 ctl out1 xor2
Xbusctl#3 in0 ctl out0 xor2
```

Using iterators and the “constant0” device from the standard cell library, here’s a better way of connecting cmp[31:1] to ground:

```
Xgnd cmp[31:1] constant0
```

Since the “constant0” has one terminal and we supply 31 nodes, 31 copies of the device will be made.

6. Built-in subcircuits

$memory
$xxx

We'll be using a new component this week: a multi-port memory. JSim has a built-in memory device that can be used to model memories with a specified width and number of locations, and with one or more ports. Each port has 3 control signals and the specified number of address and data wires. You can instantiate a memory device in your circuit with a statement of the form

```
Xid ports… $memory width=w nlocations=nloc options…
```

The width and nlocations properties must be supplied: *w* specifies the width of each memory location in bits and must be between 1 and 32. *nloc* specifies the number of memory locations and must be between 1 and 2^{20}. All the ports of a memory access the same internal storage, but each port operates independently. Each *port* specification is a list of nodes:

oe clk wen $a_{naddr-1}$ … a_0 d_{w-1} … d_0

where

oe is the output enable input for a read port. When 1, data is driven onto the data pins; when 0, the output pins are not driven by this memory port. If this port is only a write port, connect this terminal to the ground node "0". If the port is only a read port and should always be enabled, connect this terminal to the power supply node "vdd".

clk is the clock input for write ports. When wen=1, data from the data terminals is written into the memory on the rising edge of clk. If this port is only a read port, connect this terminal to the ground node "0".

wen is the write enable input for write ports. See the description of "clk" for details about the write operation. If this port is only a read port, connect this terminal to the ground node "0".

$a_{naddr-1}$ … a_0 are the address inputs, listed most significant bit first. The values of these terminals are used to compute the address of the memory location to be read or written. The number of address terminals is determined from the number of locations in the memory: naddr = ceiling($\log_2$(*nloc*)). When the number of locations in a memory isn't exactly a power of 2, reads that refer to non-existent locations return "X" and writes to non-existent locations have no effect.

d_{w-1} … d_0 are the data inputs/tristate outputs, listed most significant bit first.

By specifying one of the following options it is possible to specify the initial contents of a memory (if not specified, the memory is initialized to all X's):

file="*filename*"
: The memory is initialized, location-by-location, from bytes in the file. Data is assumed to be in a binary little-endian format, using ceiling(w/8) bytes of file data per memory location. Bits 0 through 7 of the first file byte are used to initialize bits 0 through 7 of memory location 0, bits 0 through 7 of the second file byte are used to initialize bits 8

through 15 of memory location 0, and so on. When all the bits in a memory location have been filled, any bits remaining in the current file byte are discarded and then the process continues with the next memory location. In particular, the ".bin" files produced by BSim can be used to initialize JSim memories. For example, the following statement would create a 1024-location 32-bit memory with three ports: 2 read ports and 1 one write port. The memory is initialized from the BSim output file "lab6.bin".

```
Xmem
+ vdd 0 0 ia[11:2] id[31:0]    // (read) instruction data
+ vdd 0 0 ma[11:2] mrd[31:0]   // (read) program data (LDs)
+ 0 clk wr ma[11:2] mwd[31:0]  // (write) program data (STs)
+ $memory width=32 nlocations=1024
+ file="/mit/6.004/bsim/lab6.bin"
```

`contents=(` *data…* `)`

The memory is initialized, location-by-location, from the data values given in the list. The least significant bit (bit 0) of a value is used to initialize bit 0 of a memory location, bit 1 of a value is used to initialize bit 1 of a memory location, etc. For example, to enter the short test program ADDC(R31,1,R0); ADDC(R31,2,R1); ADD(R0,R1,R2) one might specify:

```
Xmem
+ vdd 0 0 ia[11:2] id[31:0]    // (read) instruction data
+ vdd 0 0 ma[11:2] mrd[31:0]   // (read) program data (LDs)
+ 0 clk wr ma[11:2] mwd[31:0]  // (write) program data (STs)
+ $memory width=32 nlocations=1024
+ contents=(0xC01F0001 0xC03F0002 0x80400800)
```

Initialized memories are useful for modeling ROMs (e.g., for control logic) or simply for loading programs into the main memory of your Beta. One caveat: if the memory has a write port and sees a rising clock edge with its write enable not equal to 0 and with one or more of the address bits undefined (i.e., with a value of "X"), the entire contents of the memory will also become undefined. So you should **make sure that the write enable for a write port is set to 0 by your reset logic** before the first clock edge, or else your initialization will be for naught.

The following options can be used to specify the electrical and timing parameters for the memory. For this lab, these should not be specified and the default values used.

`tcd=`*seconds*

the contamination delay in seconds. Default value = 20ps.

`tpd=`*seconds*

the propagation delay in seconds. This is how long it takes for changes in the address or output enable terminals to be reflected in the values driven by the data terminals. Default value is determined from the number of locations:

Number of locations	t_{PD}	***Inferred type***
nlocations ≤ 128	2ns	Register file
128 < nlocations ≤ 1024	4ns	Static ram
nlocations > 1024	40ns	Dynamic ram

`tr=`*`seconds_per_farad`*
the output rise time in seconds per farad of output load. Default value is 1000, i.e., 1 ns/pf.

`tf=`*`seconds_per_farad`*
the output fall time in seconds per farad of output load. Default value is 500, i.e., 0.5 ns/pf.

`cin=`*`farads`*
input terminal capacitance in farads. Default value = 0.05pf.

`cout=`*`farads`*
output terminal capacitance in farads. Default value = 0pf (additional t_{PD} due to output terminal loading is already included in default t_{PD}).

The size of a memory is determined by the sum of the sizes of the various memory building blocks shown in the following table:

Component	*Size (μ^2)*	*Notes*
Storage cells	nbits * cellsize	nbits = nlocs * width cellsize = nports (for ROMs and DRAMS) cellsize = nports + 5 (for SRAMS)
Address buffers	nports * naddr * 20	nports = total number of memory ports
Address decoders	nports * (naddr+3)/4 * 4	Assuming 4-input ANDs
Tristate drivers	nreads * width * 30	nreads = number of read ports
Write-data drivers	nwrites * width * 20	nwrites = number of write ports

7. Control statements

`.checkoff`
info…

`.connect` *`node1 node2 node3...`*
The .connect statement is useful for connecting two terminals of a subcircuit or for connecting nodes directly to ground. For example, the following statement ties nodes cmp1, cmp2, ..., cmp31 directly to the ground node (node "0"):

```
.connect 0 cmp[31:1]
```

Note that the `.connect` control statement in JSim works differently than many people expect. For example,

```
.connect A[5:0] B[5:0]
```

will connect **all** twelve nodes (A5, A4, ..., A0, B5, B4, ..., B0) together – usually not what was intended. To connect two busses together, one could have entered

```
.connect A5 B5
.connect A4 B4
...
```

which is tedious to type. Or one can define a two-terminal device that uses `.connect` internally, and then use the usual iteration rules (see next section) to make many instances of the device with one "X" statement:

```
.subckt knex a b
.connect a b
.ends
X1 A[5:0] B[5:0] knex
```

`.dc`
info…

`.end`
info…

`.global`
info…

`.include`
info…

`.model`
info…

`.mverify`
info…

`.op`
info…

`.options`
info…

```
.plot
```
info…

```
.plotdef
```
info…

```
.subckt
   … circuit description…
.ends
```
info…

```
.temp
```
info…

```
.tempdir
```
info…

```
.verify
```
info…

8. Running a simulation

 Stop simulation. Clicking this control will stop a running simulation and display whatever waveform information is available.

 Device-level simulation. Use a Spice-like circuit analysis algorithm to predict the behavior of the circuit described by the current netlist. After checking the netlist for errors, JSim will create a simulation network and then perform the requested analysis (i.e., the analysis you asked for with a ".dc" or ".tran" control statement). When the simulation is complete the waveform window is brought to the front so that the user can browse any results plotted by any ".plot" control statements.

 Fast transient analysis. This simulation algorithm uses more approximate device models and solution techniques than the device-level simulator but should be much faster for large designs. For digital logic, the estimated logic delays are usually within 10% of the predictions of device-level simulation. This simulator only performs transient analysis.

 Gate-level simulation. This simulation algorithm only knows about gates and logic values (instead of devices and voltages). We'll use this feature later in the term when trying to simulate designs that contain too many mosfets to be simulated at the device level.

 Switch to waveform window. In the waveform window this button switches to the editor window. Of course, you can accomplish the same thing by clicking on the border of the window you want in front, but sometimes using this button is less work.

 Using information supplied in the checkoff file, check for specified node values at given times. If all the checks are successful, submit the circuit to the on-line assignment system.

9. Waveform browsing

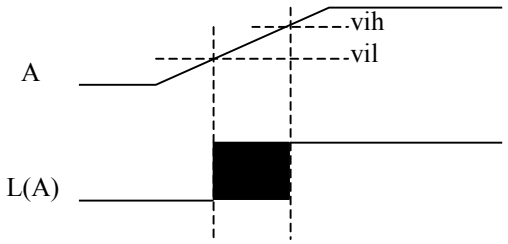
The waveform window shows various waveforms in one or more "channels." Initially one channel is displayed for each ".plot" control statement in your netlist. If more than one waveform is assigned to a channel, the plots are overlaid on top of each using a different drawing color for each waveform. If you want to add a waveform to a channel simply type in the appropriate signal name to the list appearing to the left of the waveform display (the name of each signal should be on a separate line). You can also add the name of the signal you would like displayed to the appropriate ".plot" statement in your netlist and rerun the simulation.

If you simply name a node in your circuit, its voltage is plotted. You can also ask for the current through a voltage source by entering "I(Vid)".

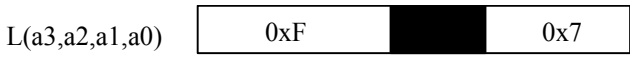
Interpreting analog signal levels as logic values can be tedious. JSim will do it for you automatically if you ask to plot "L(a)" instead of just "a". The logic-high and logic-thresholds are determined by the "vih" and "vil" options:

```
.options vih=2.6 vil=0.6
```

Initial values are specified in "/mit/6.004/jsim/nominal.jsim", but you can respecify them in your own netlist. Voltages between vil and vih are displayed as a filled-in rectangle to indicate that the logic value cannot be determined. For example:



You can also ask for the values of a set of signals to be displayed as a bus, e.g., "L(a3,a2,a1,a0)". The signals should be listed most-significant bit first. A bus waveform is displayed as a filled-rectangle if any of the component signals has an invalid logic level or as a hexadecimal value otherwise. In the following plot the four signals a3, a2, a1 and a0 are interpreted as a 4-bit integer where the high-order bit (a3) is making a 1→0 transition. The filled-in rectangle represents the period of time during which a3 transitions from V_{IH} to V_{IL}.



The default display radix for a bus is hexadecimal, but you can ask for other radicies as well:

L(…)	hexadecimal display, unsigned number
d(…)	decimal display, unsigned number
sd(…)	decimal display, signed two's-complement number
b(…)	binary display, unsigned number
o(…)	octal display, unsigned number
x(…)	hexadecimal display, unsigned number (same as "L")

It's also possible to define your own "symbolic" display using plotdef, for example:

```
.plotdef numbers zero one two three four five six seven eight nine ten
+ eleven twelve thirteen fourteen fifteen
```

When you plot a bus using numbers(…), the value of the bus is used as an index into the table of strings and the resulting string is displayed in the plot:

L(a3,a2,a1,a0)	0xF	0x7	0x0
numbers(a3,a2,a1,a0)	fifteen	seven	zero

Making Measurements

Each waveform plot has an oscilloscope-like grid in the background. The scale for each division is shown at the bottom left-hand corner of the plot. You can change the scale by using the zoom buttons at the bottom of the browser window (or you can use the "X" or "x" keyboard shortcuts).

The scrollbar at the bottom of the browser window can be used to pan along the horizontal dimension of the plot. Or you can use the "c" keyboard shortcut to recenter the plot at the current mouse position.

As you move the mouse over a particular waveform plot, a crosshair cursor follows the first waveform found above the current mouse position. The readout in the upper left-hand corner of the plot gives the current coordinates of the crosshair cursor.

To measure an interval on the plot, position the crosshair cursor at one end of the interval, click and drag the mouse to the other end of the interval. A second crosshair cursor will appear as you drag the mouse, and a second readout line gives the current coordinates of the second cursor as well as the delta between the two cursors' coordinates.

The Waveform Toolbar

The waveform window has several other buttons on its toolbar:

 Select the number of displayed channels; choices range between 1 and 16.

 Print. Prints the contents of the waveform window (in color if you have a color printer!). If you are using Athena, you have to print to a file and then send the file to the printer: select "file" in the print dialog, supply the name you'd like to use for the plot file, then click "print". You can send the file to one of the printers in the lab using "lpr", e.g., "lpr -Pcs foo.plot".

You can zoom and pan over the traces in the waveform window using the control found along the bottom edge of the waveform display:

 zoom in. Increases the magnification of the waveform display. You can zoom in around a particular point in a waveform by placing the cursor at the point on the trace where you want to zoom in and typing upper-case “X”.

 zoom out. Decreases the magnification of the waveform display. You can zoom out around a particular point in a waveform by placing the cursor at the point on the trace where you want to zoom out and typing lower-case “x”.

 surround. Sets the magnification so that the entire waveform will be visible in the waveform window.

The scrollbar at the bottom of the waveform window can be used to scroll through the waveforms. The scrollbar will be disabled if the entire waveform is visible in the window. You can recenter the waveform display about a particular point by placing the cursor at the point which you want to be at the center of the display and typing “c”.