


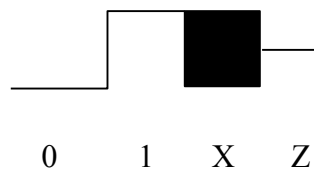
Gate-level Simulation

Since we're designing at the gate level we can use a faster simulator that only knows about gates and logic values (instead of transistors and voltages). You can run JSim's gate-level simulator by clicking  in the toolbar. Note that your design can't contain any mosfets, resistors, capacitors, etc.; this simulator only supports the gate primitives in the standard cell library.

Inputs are still specified in terms of voltages (to maintain netlist compatability with the other simulators) but the gate-level simulator converts voltages into one of three possible logic values using the VIL and VIH thresholds specified in nominal.jsim:

0	logic low (voltages less than or equal to VIL threshold)
1	logic high (voltages greater than or equal to VIH threshold)
X	unknown or undefined (voltages between the thresholds, or unknown voltages)

A fourth value "Z" is used to represent the value of nodes that aren't being driven by any gate output (e.g., the outputs of tristate drivers that aren't enabled). The following diagram shows how these values appear on the waveform display:



Connecting electrical nodes together using .connect

JSim has a control statement that lets you connect two or more nodes together so that they behave as a single electrical node:

```
.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
```

Using iterators to create multiple gates with a single “X” statement

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:

```
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
```

since xor2 has 3 terminals. 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.