

1. Initialization and Technology Files

To invoke Cadence 4.2 with the CMOS4S technology, type “startCds -t cmos4s”. Cadence first reads in the contents of the .cdsinit file in /nfs/d06/cds_4.2/local_V1.1. This file calls /nfs/d06/cds_4.2/local_V1.1/skill/CMCinit.il in order to initialize the Cadence environment for use with the CMC CMOS4S process. Just before finishing, this skill script invokes ~/.cdsinit and ./cdsinit, in that order. Thus to customize your Cadence environment at startup, one should create a .cdsinit file in either one’s home directory or the directory from which Cadence will be started. In order to ensure all members of a design team to have the same Cadence settings as the initialization is altered to fix bugs one should probably have a startup directory containing .cdsinit from which everyone invokes Cadence.

Another pair of initialization files used by Cadence are .simrc and .simlocal. While .cdsinit is invoked only once, when Cadence starts, .simrc and .simlocal are invoked whenever a simulator (such as HSPICE or Verilog) is started via Cadence. /nfs/d06/cds_4.2/local_V1.1/.simrc contains the default CMC setup for these simulators; a .simlocal file should be maintained in your startup directory to perform any desired customizations. The customized startup files used in our project are contained in /pc/vrg/d1/atm94/start_up and show how to set many of the more useful Cadence variables.

The technology file for the CMOS4S process is /nfs/d06/cds_4.2/local_V1.1/lib/cmos4s/cmos4s.tf. In order to change the display colours and patterns of some layout layers and to add some DRC rules, we created our own copy of this file and modified it. After modifying a technology file, be sure to recompile the technology file for all the affected libraries using Technology File -> Compile Technology.

If you start Cadence 4.2 twice in the same OpenWindows session, the X server usually crashes. Logout of the system and restart it if this happens. As well, Cadence appears to have some memory corruption problems -- if you ever find it is behaving very strangely for no apparent reason, try exiting Cadence and restarting.

2. Layout hints

2.1 Modality

The Cadence Virtuoso interface is modal - that is, the response to a mouse click or keypress depends on internal state which is often not visible to the user. It is well known that modal interfaces confuse users and are to be avoided. Since the Cadence designers have never read the Motif style guide (or if they did, they ignored it) you have to cope with the modality yourself. The functions performed when any of the mouse buttons is clicked are listed underneath the text entry widget in the CIW.

In general, the action performed when a command (move, stretch or delete) is invoked depends on whether an object was selected and whether the command was invoked from a bindkey or from a menu.

- If a command requires you to select an object, and the command is invoked through the menu, then you will be asked to select the object. If the command was invoked through a bindkey

then:

- If an object was selected when the command was invoked, then the command applies only to the selected object.
- If no object was selected, then Cadence applies the command to all objects selected from there on, until the mode is exited. You can exit a mode by pressing Esc.

It follows that menus and bindkeys are not equivalent.

Note that clicking on an empty area is one way to “not select” an object, but be careful: if you are editing a small area in a hierarchy, then you may have selected an instance, or an N well (which is large and transparent in the middle). A better way to deselect everything is to use the Ctrl-D bindkey.

2.2 Selection tricks

The f4 key (`geTogglePartialSelect`) toggles “partial select” mode.

In partial select mode, you can select edges or corners of shapes, rather than the whole shape. This is useful for stretching, since the stretch command acts on the entire object if it is invoked from a bindkey. In this case, the object is moved instead of stretched.

You cannot use partial select mode to delete objects. If you do, then the selected edges or corners will be deleted, but the object won't. At best, you get an annoying dialog box telling you that the shape cannot be deleted because it would become malformed.

You can select multiple objects using the shift and control keys with the mouse. Shift-click adds the selection to the current selection, and Ctrl-click deselects the current object.

Finally, the AV, NV, AS, and NS toggles in the LSW are useful, as follows:

- AS,NS: all/none selectable: makes all or none of the layers selectable.
- AV,NV: all/none visible: makes all or none of the layers visible.

Clicking the right mouse button on a layer in the LSW toggles its selectability. Clicking the middle mouse button toggles the layer's visibility.

Making a layer non-selectable is useful when two layers overlap exactly, and you want to move or stretch only one of them. Just be sure to turn the selectability on again lest you wonder why you can't move or stretch something later.

The Inst and Pin buttons at the top of the LSW control the selectability of instances and pins.

Making a layer invisible simplifies the display for certain purposes, such as filling in N wells or doping masks. You must use the ctrl-R bindkey to force a redraw after changing the visibility of a layer; Cadence will not update the display automatically after a visibility change.

Some layers are not shown in the LSW by default. To see these layers, choose Edit->Set Valid Layers and make them valid. They will now show up in the LSW, and you can make them visible or invisible as desired.

2.3 Miscellaneous

The q bindkey brings up the properties of an object. Among the more useful properties to change

is the mask layer (useful if you've just drawn a complex shape OK, but in the wrong layer). The q bindkey is also useful for changing the names or types of pins, or for changing the cell to which an instance refers.

If you move a bunch of objects, then Cadence normally highlights the objects so that you can line them up with another part of the layout. However, if the bunch is large, then Cadence will draw a box around the objects, and you get to move a big box instead of an outline of all the objects. To line something like this up, do the following:

- Select the objects
- Zoom into one corner of the display and put the mouse pointer over a distinguishing feature of the set of objects to be moved.
- Hit the m bindkey. Although Cadence draws a huge box leaving you in the dark, the feature of the objects being moved will be located under the mouse pointer when you click again. Simply move the mouse to where you want that feature to be, then click.

The C bindkey does a “layer chop”. Once activated, the rectangle that you select will be “cut out” of the layout. This option is useful for creating the slotted metal required in the ring of CMOS4S designs.

In the display options menu, the option “Instance Pins” causes pins of the instances one level down in the hierarchy to be displayed. This is useful to ensure that you cover the pins of the lower-level instances.

3. Extraction

3.1 Pins

Cadence offers two types of layout extraction - flat and hierarchical. Hierarchical extraction is much faster (approximately 100 times faster for large designs) but in order to use hierarchical extraction one must follow certain rules. First, connections between cells can only be made at pins. Cells can only “see” the pins of cells which lie one level below them in the hierarchy - if a connection is to be made to a cell n levels higher in the hierarchy, pin metal must be laid out in each of the intervening n-1 layers to propagate the pin up to that level. Always be sure that the metal connecting two lower-level pins actually overlaps each pin; the extractor performs a geometrical AND operation to see if there is a connection so merely butting metal up to a pin will not make a connection. In order to make sure one is connecting only to pin metal, it is useful to use Design->Options->Display to turn instance pins on and make the pins one level below visible.

Pins are normally of either input or output type; however there are cases where input-output pins must be used. Whenever a single net (electrically connected wire) is expected to be driven at one end by a higher-level cell and in turn the other end of this net drives some input at a higher level, all pins on this net must be of type input-output. This typically occurs for metal feedthroughs - a signal is not modified by a lower level cell, but merely passes through it.

So that connecting cells at pins is not too much of a hardship, it is a good idea to cover most of the likely points at which a cell will be contacted with pin metal - particularly for the lowest-level cells, since they are likely to be used in many different situations. One can place as many

distinct pin rectangles on a net as desired; just make sure you give each electrically common pin the same name.

Pins at the highest level of the hierarchy are the inputs and outputs to the outside world. The input pins are driven during a simulation, and the output pins are easy to monitor.

3.2 Using the Extracted View

The hierarchical extractor creates two cellviews for each cell in the hierarchy: an excell and an extracted cellview. The excell is a sort of template for each cell - it shows where the various pins are and is used to determine what the inputs to the cell are for each instance. The extracted view shows the interconnect and metal internal to the cell, and defines what a cell actually does. If your simulation is not working, probing the extracted cellview can be very useful to be sure that things are connected properly. Verilog complains about nets with no fanin, and HSPICE complains about nets with no dc path to ground - they have found floating wires in your logic. To find such a net in the extracted cellview, you must first determine the Cadence name for the net. Global nets like vdd! and gnd! keep their original name. Other nets have names like /number. The start of an HSPICE netlist contains a table for converting HSPICE node numbers to Cadence names. While reading the extracted cellview select Verify->Probe->Add Device or Net. If you know the net name, type the net name, in double quotes, on the CIW command line, e.g. “/18”, or you can click on any net with the mouse. The net will be highlighted in purple (hilite d1 layer). Sometimes this highlighting is difficult to see; if it is select Edit->Set Valid Layers in the LSW and make all the layers valid. The extracted geometries are drawn in the nt layers; turning off these layers will make the highlighting show more clearly.

To see how large parasitic capacitances are or to verify that the dimensions of a transistor are correct, click on the desired device in the extracted view and press q.

3.3 Extraction Procedure

You must enter the switch name “do_extract” into the extraction form in order for the extract to do anything. If you want to find parasitics enter “do_extract do_parasitics” as the switch names. Do not use the “do_parasitics” switch when you are extracting in order to do a Verilog simulation - Verilog does not have models for the capacitors. Finally, if there are some pins with the same name which are not electrically connected, but which will be connected at a higher level, you should turn on the “join nets with the same name” option. Be sure you do not do this during the final extractions of the whole chip - at that point you want to ensure that all connections have been physically made.

Hierarchical extract will not find shorts at places other than pins, so a flat extraction and simulation must be performed on the final layout to fully verify functionality. Incremental hierarchical extraction re-extracts only those cells whose layouts have changed since the last extract, and thus should usually be used. Use a full hierarchical extract to force the re-extraction of all cells.

After a layout is extracted, you should run a SKILL file which sets all signals ending in ! to be global signals. The CMC provides the CMOS4SfixExtract() routine to do this, but it becomes very slow (5 hours running time) for large extracted views. We have written a replacement routine which eliminates the useless code in CMOS4SfixExtract() and runs about 100 times

faster. The SKILL routine is called `our_globalize()` and is located in `/pc/vrg/d1/atm94/skill`. To invoke it, type `our_globalize` on the CIW command line while editing either the layout or extracted cellview (you may first have to load `our_globalize.il` if your initialization files do not do so automatically).

Large extractions consume huge amounts of disk space; it took 130 MB to do a flat extract of a 56 000 transistor chip. If you are running out of disk space, before starting Cadence setenv `DRCTEMPDIR` to a directory with plenty of unused space. You can specify several directories on different disks if you wish; Cadence will store some files on each.

4. Netlisting

The Session->Options->Netlisting form controls the netlister operation. The Netlisting View List lists the views, in order, which will be used to generate the netlist for each cell. The Netlisting Stop List lists the terminating views -- when the netlister encounters such a view it does not continue any further down this portion of the hierarchy. For example, if the Netlisting View List is “verilog extracted symbol” and the Netlisting Stop List is “verilog symbol”, the netlister will, for each cell, check if there is a verilog cellview. If there is, the information in this verilog cellview will be written out and the netlister will not netlist any submodules that are instantiated in this cell. If there is no verilog view, the netlister will look for an extracted view. If this view is found, the information in it is written out, and, since extracted is not in the stop list, the netlister will descend into any submodules instantiated in the extracted view in order to netlist them. If neither verilog nor extracted cellviews exist, the information in the symbol view will be written out.

If you are simulating pads, you will likely have `global vddring!` and `gndring!` nets -- make sure you add these to the Global Power and Ground Nets fields in the Netlisting Options form

5. Verilog Simulation

5.1 Overview

Verilog is a switch-level simulator - it models logic gates or transistors as perfect switches. The rise and fall delays of gates and transistors can be specified. As well, drive strengths (supply, strong, weak) can be given for gates; transistors, however, are always of “strong” strength. In the absence of any delay information, Verilog assumes an element has zero delay. There are two types of transistor models: unidirectional and bidirectional. The `nfet` and `pfet` models always assume that information flows from the source of the transistor to the drain, under control of the gate. The `tranif0` (nmos) and `tranif1` (pmos) models are bidirectional -- the source and drain are symmetric, just as they are in a real transistor. These bidirectional models are slower (up to 20 times) but they work much better with Cadence, since the Cadence extractor and netlister randomly assign one side of a transistor to be the source and the other side to be the drain. Even with the slower bidirectional models Verilog is very fast - it simulated a 56 000 transistor chip at the transistor level in 1 hour.

When netlisting schematics for Verilog, one’s netlisting options should be similar to:

Netlisting View List: verilog schematic symbol

Netlisting Stop List: verilog symbol

If you are netlisting extracted views, replace schematic in the above two lines with extracted.

5.2 Problems with CMC CMOS4S Library Setup

The CMC CMOS4S library is not set up properly to do Verilog simulations in which transistors are used. There are three problems:

1. The nmos devices are defined as nfets with the substrate tied to the source. This confuses Verilog because it has a built in model for nmos devices, which the netlist will now attempt to redefine.
2. Even if this redefinition of nmos (and pmos) problem is removed, Verilog treats nmos devices as unidirectional, but we have no guarantee that the netlister will always treat the drain as the transistor output -- half the time it uses the source as the output.
3. There are no built-in Verilog models for pfets and nfets, and the CMC release does not provide Verilog model files to define them.

The first two problems were solved in the library `/pc/vrg/d1/atm94/lib/cmos4s/our_cmos4s`. The nmos and pmos devices now both have a verilog cellview which ensures that nmos and pmos devices are netlisted correctly. Whenever you create a verilog cellview, add a property of type string called `hdlVerilogFormatInst`. In nmos this property has the value `hdlVerilogPrintBidiXfr("tranif1")` while in the pmos verilog cellview it has the value `hdlVerilogPrintBidiXfr("-tranif0")`. To solve the third problem, we created Verilog models for pfet and nfet; the contents of the model file `pfet.v` are listed below (both `pfet.v` and `nfet.v` are in `/pc/vrg/d1/atm94/lib/cmos4s/models/verilog`).

```
// Defines a pfet as a pmos (both are tranif0).
`resetall
`celldefine
`timescale 1ns/10ps

/* macromodule for speed */
macromodule pfet (B, G, D, S);
/* Note: need to use B, G, etc. since Cadence netlist
passes these ports by name. */
input B, G;
inout D, S;
/* drain and source are bidirectional */

tranif0 (D, S, G);
```

```
endmodule
`endcelldefine
```

5.3 Feedback Fights and Verilog Models

Verilog does not understand transistor widths or lengths -- it assumes all transistors are of equal strength (strong). In order to model structures like static latches, one must create Verilog models and verilog cellviews such that the netlisting of a latch will not go all the way down to the transistor level. For example, we created a verilog cellview for the half_latch cellview; the schematic of our half latch is shown below.

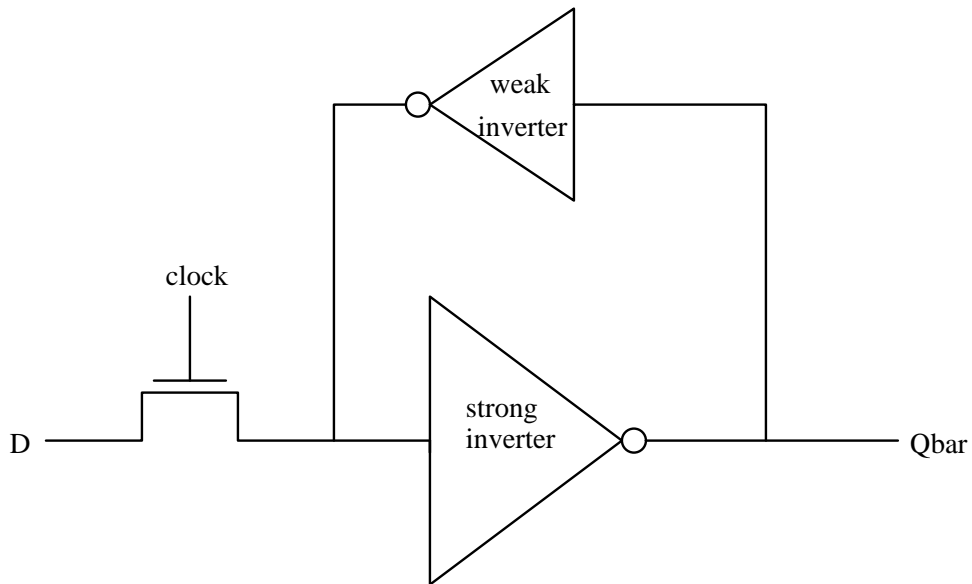


Figure 1. Figure 1: Half Latch Schematic

The `hnlVerilogFormatInst` property on this verilog cellview is set to `hnlVerilogPrintPrimitive("half_latch Q clk_a D vdd! gnd!")` and in the verilog models directory (which is set by the `simVerilogLibraryDirectory` variable in your `.simlocal` file) we have a file called `half_latch.v` which contains:

```
`resetall
`celldefine
`timescale 1ns/10ps
`delay_mode_path

module half_latch(q, clk, d, dumv, dumg);
    output q;
    input clk, d, dumv, dumg;

    not (weak0, weak1) (t, q);
    not (q, t);
    tranif1(d, t, clk);

    specify
        (d,clk *> q) = (0.91, 2.17);
    endspecify
endmodule
`endcelldefine
```

This model resolves the feedback fight by declaring that the feedback inverter (`not`) is weak. It also contains timing information in the `specify` block. The line `(d,clk *> q) = (0.91, 2.17)` specifies that the rise time (output rising) of this latch is 0.91 ns while the fall time (output falling) is 2.17 ns. The output will make an abrupt transition 0.91 ns (or 2.17 ns, depending on the transition) after either `d` or `clk` changes. Note that `vdd!` and `gnd!` are called `dumv` and `dumg` in the verilog model file -- using the names `vdd!` or `gnd!` will in the model will confuse Verilog. The Verilog cellview must use names identical to those of the schematic or extracted (and therefore layout) view, and the order of the pins in the verilog cellview defines the order in which the parameters of the verilog model should be listed in the model file.

There are two basic reasons for creating Verilog models: to model structures in which there are feedback fights, and to specify timing information for basic cells. You should give all Verilog model files a `.v` extension.

5.4 Running Verilog on Flat Netlists (hsp2vl Program)

Verilog models can be used to replace cells during the netlist operation if a schematic or an hierarchically-extracted layout is being netlisted. If a flat extracted view is being netlisted, the entire hierarchy has been collapsed down to the transistor level, and the Verilog netlist will consist only of `tranif0` and `tranif1` elements. This means that any feedback fights cannot be resolved; if two transistors attempt to drive the same node to different values, the result is an unknown.

The circuit used in our static latches contains a feedback loop consisting of a strong and a weak inverter. Verilog could handle this circuit, but it needs to know that the weak inverter is weak. This information is not provided by the extractor in Cadence. Thus, a translator (HSP2VL) was written which takes an hspice netlist and looks for pairs of n- and p-transistors whose gates are connected together and are very long where the p-transistor is connected to VDD and the n-transistor is connected to GND (and to each other). In other words, it looks for weak inverters and replaces those two transistors with an instantiation of a module called `weak_inverter`. All other transistors are converted to a Verilog notation and output otherwise unchanged. Note that only a subnet matching a weak inverter exactly will be modified. Non-conforming inverters or nets which are close will not match. In this way, the connectivity check that a flat simulation provides is not compromised by the use of the `weak_inverter` module.

HSP2VL was made to do as much automatically as was possible. This meant getting the names and nodes for the power supply correct, and also including the correct net names in the output as well. Some name conversion was necessary since names like `vdd!` and `5` are not valid Verilog net names. The program assumes that there is a list of net names at the beginning of the HSPICE netlist as generated by Cadence.

The names of the power supplies are assumed to be `"gnd!"`, `"vdd!"`, `"gndring!"`, and `"vddring!"`. The last two make the translator usable on a full chip design. There is a file `ihnl/globalmap` in every Verilog run directory which contains the mapping for the global signals such as the power supplies. HSP2VL reads in this file and finds the names that it should use in the output netlist for the power supplies. When any of the names `"gnd!"`, `"vdd!"`, `"gndring!"`, or `"vddring!"` are found in the list of net names at the beginning of the HSPICE netlist, it automatically considers the name of the corresponding node to be the name it found in the `globalmap` file.

The netlist that is output is suitable for inclusion in a netlist file generated by a flat Verilog netlist of the same cell. The `nfets` and `pfets` in the Verilog-generated netlist should be deleted and replaced with the output from HSP2VL. There will be a duplicate set of supply definitions in the Verilog-generated netlist: once it is verified that they are the same as those generated by HSP2VL, they can be safely deleted. Note that the `"endmodule"` command should follow the complete netlist.

Optionally, HSP2VL can output a full module definition. It will contain all the named nodes in the port definition, and a list of input/output declarations for the same names. Since HSP2VL cannot determine which nodes are inputs and which are outputs, it outputs both an input and an output declaration for each name. The incorrect one can be simply deleted. Also, the module name must be specified by a person. This version of HSP2VL was used to create the netlists for the low-level blocks in order to generate schematics from the layout.

5.5 Usage of HSP2VL

There are two methods of invoking HSP2VL. The first takes all arguments from the command line. The second assumes the arguments are contained in a file called `"hsp2vl.cfg"` in the current directory. Both versions write their output to the standard output. The command line version is outlined below.

```
hsp2vl <max_nodes> <threshold length> <HSPICE netlist file> <globalmap file> [-m]
```

The HSPICE netlist should have fewer nodes than the `max_nodes` argument. If it has more nodes, HSP2VL is not guaranteed to work. The `threshold length` argument is the dimension of the

shortest transistor gate which is to be considered weak. Any transistor with a gate at least as long as this number will be assumed to be half of a weak inverter. HSP2VL will output a warning if it finds transistors with gates which are greater than minimum length (1.2 microns) but less than the threshold length.

The HSPICE netlist file is the path of the file containing the HSPICE netlist. Similarly, the globalmap file argument is the path of the file containing the globalmap entries created by a Verilog netlist.

The -m argument is optional (denoted by the square brackets). If it is present, HSP2VL will output a full Verilog module definition. Otherwise, it will only output a Verilog netlist.

If HSP2vl is called with no arguments, then it looks for a file called “hsp2vl.cfg” in the current directory, from which it reads the arguments in the same order as in the command-line version, one line per argument. The only difference is that the -m option is specified using the gen_module keyword on the last line by itself.

5.6 Flat Verilog Simulations Using HSP2VL

There is a standard set of steps to follow when using HSP2VL to perform a flat Verilog simulation. They are detailed below.

1. Do a flat extract of the cell you want to simulate in Verilog.
2. Do an HSPICE netlist into some run directory. Call it hnetlistdir for purposes of illustration.
3. Do a flat Verilog netlist into another run directory. Call this directory vnetlistdir
- 4.

```
% cd vnetlistdir/ihnl/cds0
```

This is the directory where the flat Verilog netlist was generated.

5. Run HSP2VL.

e.g.

```
% hsp2vl 1000 2e-6 hnetlistdir/netlist ../globalmap >
netlist.vl
```

or: create a file called hsp2vl.cfg containing

```
1000
2e-6
hnetlistdir/netlist
../globalmap
```

and simply run

```
% hsp2vl >netlist.vl
```

6. Edit the file “netlist” in the current directory. This is the file that was generated by the Verilog netlister. Delete all the lines defining nfets and pfets. Read in the file netlist.vl just before the “endmodule” command. Verify that the supply commands in the “netlist” file and in the “netlist.vl” file are the same, then delete one set. Make sure you save this into the file called “netlist” so that the simulator will find this modified version of it. This substitutes the HSP2VL netlist for the netlist that the Verilog netlister generated, while keeping the names and port list consistent with the other files were generated at the same time.

7. Perform a Verilog simulation as usual. It will pick up the new flat Verilog netlist. Make sure that a module called `weak_inverter` is included in the netlist or that a file `weak_inverter.v` exists on the library path for the simulator. The outputs of the weak inverter must be specified as (`weak0`, `weak1`) or the simulation still won't work.

```
e.g. not (weak0, weak1) (OUT, A);
```

5.7 Generating Test Vectors and Clocks

Cadence Simulate and Test Language (STL) provides a simple way to generate test vectors. Use Stimulus->STL->Edit Stimulus to create an STL file. The `defformat` statement declares the ordering of inputs (and possibly outputs) used by the test vectors. The test vectors are of the form `xv(0 1 ...)` and are placed 1 per line between the `deftest` and `endtest` statements. The `deftiming` statement has three parameters; the first number defines the time step for the simulator; the second defines the time resolution used in declaring clocks and strobes; and the third defines the period of each test vector and clock. In order to create a clock, one must first change the type of the corresponding input from `in` to `clk`. Then one uses `defclock` statements to actually declare the clock. The statements below, for example, declare a 50 MHz two-phase nonoverlapping clock.

```
deftiming 1ns 2ns 20ns
defclock "1111....." clk50_ph0
defclock ".....1111." clk50_ph1
```

A 1 in a position indicates the clock is high for one time-resolution unit; a . indicates it is low.

Clocks can also be placed in the `testfixture.template` file for a Verilog simulation. This is necessary when one has clocks of different periods on the same chip (since the `deftiming` statement implies that we can only have one period for all clocks). The lines below show how to simultaneously define both 10 and 50 MHz two-phase clocks in the `testfixture.template` file.

```
initial
begin
  clk50_ph0 = 1;
  clk50_ph1 = 0;
end

always
begin
  #8 clk50_ph0 = 0;
  #2 clk50_ph1 = 1;
  #8 clk50_ph1 = 0;
  #2 clk50_ph0 = 1;
end

initial
```

```

begin
  clk10_ph0 = 1;
  clk10_ph1 = 0;
end

always
begin
  #40 clk10_ph0 = 0;
  #10 clk10_ph1 = 1;
  #40 clk10_ph1 = 0;
  #10 clk10_ph0 = 1;
end

```

Initial statements are executed only when the simulation first starts, while always statements are executed throughout the simulation. The #number in front of statements indicates how long to delay (in nanoseconds) before executing the following statement.

One often desires an output file containing the history of key logic signals. The lines below show what to insert in the testfixture.template file to achieve this.

```

integer outfile;

initial
begin
  outfile = $fopen("icsim.out");
  if (outfile == 0) $finish;
  $fdisplay(outfile, "%d %b %b", $time, eop, out0);
end

always
  $fdisplay(outfile, "%d %b %b", $time, eop, out0);

```

5.8 Verilog Simulation Directory Structure

Table 1: Important Verilog Files

File Name	Function
testfixture.template	Control and stimulus file for simulation.
input.stl	STL stimulus: compiled to form new testfixture.template.
ihnl/globalmap	Names of global signals used and modules netlisted.
ihnl/blockdirmap	Lists the directory in which the netlist of each module resides.
ihnl/cds*/netlist	Netlists for all the modules.

6. HSPICE Simulation

6.1 Creating Stimuli

When uses Cadence to create an HSPICE netlist, it creates three major files: netlist, hspice.sim and hspice.inp. As well, it will copy a control file (specified by the simControlFile variable set in .simrc or .simlocal) into this directory. One can create stimuli in one of two ways: by using STL or by editing the hspice.inp and hspice.sim files. Creating STL excitation is identical to the procedure described in section 5.5, except that the signal rise and fall times have to be defined for each input now, and the STL compilation fills in the hspice.inp and hspice.sim files for you instead of testfixture.template.

One can also write HSPICE excitations into the hspice.inp file. Instead of using hspice node numbers, however, one refers to Cadence pins using the notation [#pin_name]. For example, to define vdd! one would insert a line like

```
VDD [#vdd!] [#gnd!] DC 5V
```

into one's hspice.inp file. If you do not use STL, you also have to put a .tran statement into either the hspice.inp or hspice.sim file. When you select simulate in Cadence, the hspice.sim, hspice.inp, control and netlist files are concatenated together and the Cadence names are replaced by hspice nodes. The new HSPICE input file is si.inp.

Sometimes HSPICE runs out of memory. If this happens, one can rerun HSPICE from the command line, using si.inp as the input, and specify a larger memory allocation than the default. The control.hsp file supplied by the CMC contains an options .sda=2 line, which tells HSPICE to generate waveform output suitable for viewing through Cadence. The gsi waveform display is generally superior to that of Cadence; if you want to view the output through gsi instead, change

.options sda=2 to .options post in the control file.

6.2 Netlisting Bug

When one attempts to netlist to HSPICE after completing a Verilog netlist, Cadence may give you a stream of error messages about being unable to find the pfet and nfet masters, and fail to netlist. Cadence sometimes does not clean up properly after a Verilog netlist - before netlisting to HSPICE select the Options menu and be sure that hierarchical netlisting and incremental netlisting are off. If your troubles persist, logout of Cadence, restart it and then netlist in a different directory.

6.3 HSPICE Output Postprocessor

In order to save large volumes of disk space, and also to allow automatic testing of HSPICE simulations, a small utility called “prep” was written to convert HSPICE voltage outputs to the form that is expected by the test vector verification program mcamp. This program basically modifies voltage readings that are close to 5V and 0V into 1’s and 0’s.

6.4 Preparing the HSPICE Input File

The HSPICE input file must be modified slightly in order to output the correct format for the “prep” program. The input file should be created by doing a simulation through Cadence first (it will construct the si.inp file from the control, hspice.inp and hspice.sim files in the simulation directory). This file should then be modified as follows.

1) In order to save disk space, remove the “.options post” or “.options sda=2” command near the beginning of the si.inp file. This will stop HSPICE from writing out a trace file for the circuit, which can be extremely large for even moderately large circuits. The command “.options brief” will restrain the simulator from printing out large volumes of information about models and such that are being used, and result in a more readable output file.

2) Find the node numbers of the outputs which you want to be monitored. Then insert two commands near the top of the si.inp file: “.options ingold=1” and “.plot tran v(n1) v(n2) ...v(nn)” where n1, n2, ..., nn are the n nodes that you want to monitor. The first command instructs the simulator to write output values in scientific notation. The second merely states which nodes to print out. The simulator will print these values five or so to a line.

Run the simulation. When it is finished, the output file will contain a large table of data values for the nodes specified in the input file. Unfortunately, the simulator will not write out these values as they are determined, but only when the simulation is finished (so you have to wait until it’s done to see if it worked). Delete all superfluous lines from the output file (maybe make a copy of it first). Superfluous lines are any lines which do not contain a simulation time and a list of output values. This includes the title line for the output telling which signal is which, and any lines following the data. Running “prep” on this file will output a list of times and binary values for each node that was originally in the HSPICE output file.

6.5 Running Prep

Prep can be executed using the command:

```
prep <sample interval> <number of outputs> <input filename>
```

The HSPICE simulator will output values at the interval specified on the .tran line of the “si.inp” file. Larger intervals rarely seem to speed up the simulation much, so 0.1ns seems the “best” value. However, an automatic test vector verifying program won’t want to see data at such small intervals. Thus, “prep” is able to sample the output file at a given interval. This is the meaning of the <sample interval> argument, and it is expected to be in nanoseconds.

The <number of outputs> argument tells “prep” the number of values to expect after the time on each entry in the output file. Thus, in the discussion above, this would be equivalent to n.

The <input file name> argument is the file created as described above from the output file of the HSPICE simulation.

The output will be written to standard out.

e.g.

```
prep 20 2 sim.out >icsim.out
```

This will read the file sim.out, sampling it every 20ns, converting the two voltage levels into binary values and write the resulting data into the file icsim.out.

7. Awsim Simulation

7.1 How to Guide

Awsim presents a whole new set of challenges. Because Awsim is more or less a tool internal to the University of Toronto, Cadence does not have built-in support for it. It appears possible to write support code for any simulator so that it may be integrated into the Cadence. This approach was not taken since the immediate benefit was not evident. Rather, other in-house tools were used to support Awsim. Simulating a circuit under Awsim is a complex task as follows:

- Generate an HSPICE netlist, either with or without parasitics. See the preceding sections for detail on doing this.
- Use the sp2a program to convert the HSPICE netlist to an Awsim netlist:

```
sp2a -n xlctrl netlist >chip.aw
```

The -n option causes sp2a to extract the symbolic net names from the HSPICE netlist and use them in the Awsim netlist. By default, the Awsim netlist would contain numeric node names which convey little information.

xlctrl is a conversion control file which contains information that sp2a needs (definition of transistor models).

- Compile the Awsim netlist using preproc:

```
preproc chip.aw chip.net aw.param
```

This step creates a binary image of the netlist that Awsim will actually simulate. aw.param is a control file giving device and simulation parameters.

- Prepare the test vectors. Awsim test vectors consist of two files: a name file and a vector file. For the video switch chip, the name “awv.out” was used for the vector file and “awv.names”

was used for the name file. The Awsim documentation is actually correct when it describes the format of these files. For example:

Name file :

2:1

3:0

Vector file :

1X

0X

1X

0X

1X

0X

- Create a control script, for example:

```
zf 1
r inv.net
i 2
s 2 3
zs 0.1
zr 0.1
zo trace.nl trace.out
t 3
l 0
h 1
m inv.nod inv.vec
q
```

Consult the Awsim documentation to find out what the options do.

The 's' command gives Awsim a list of nodes to be traced. Ensure that any node whose value you want to know is present. Once Awsim starts running, you cannot add or remove nodes from the set being traced without restarting the simulation.

Also ensure that the power supplies are hooked up. The 'h' and 'l' commands set a node high or low respectively. You must set all vdd nodes high and all gnd nodes low. To determine the exact names of these nodes, issue the Unix commands:

```
grep vdd netlist
grep gnd netlist
```

- Simulate using Awsim:

```
/usr/bin/nice -19 time awsim <run.top >aw.res
```

This command directs Awsim's output to the file `aw.res`. You should inspect this file a few seconds after Awsim starts running to verify that all signal names were found. Also, Awsim will write any discrepancies between expected and actual results into this file.

- If desired, the `awv` program can be used to look at the waveforms.

7.2 Using the awv and stv graphical viewers

awv displays the file `trace.out` (Awsim's binary node trace) on an X display. stv displays the file `stl.out` (test vectors extracted from an STL file) on an X display. Arguments are as follows:

Table 2: awv and stv arguments

argument	function
-display name	Use the X display "name".
-geometry str	Use the given X window geometry. This option requires the cooperation of your window manager.
-Fname	Enable the print option, sending PostScript output to the file "name".
-Pname	Enable the print option, sending output to the given network printer.

The display has the following buttons, from left to right:

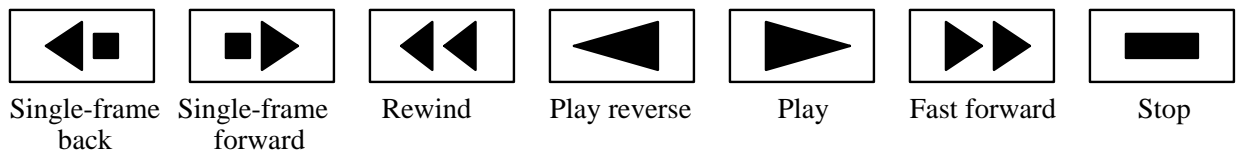


Table 3: awv and stv controls

control	function
Single-frame back	Scroll left one page.
Single-frame forward	Scroll right one page.
Rewind	Jump to time zero.
Reverse	Play the plot as a "movie", showing each page for 1.5 seconds before moving to the left one page.
Play	Play the plot as a movie, but move to the right (forward).
End	Jump to latest time simulated.
Stop	Stop the movie, if it is playing. You can also stop the animation by pressing any other button.

Table 3: awv and stv controls

control	function
Update	Re-sync the display to the simulation, including all results produced since the last update. (awv only)
Zoom in	Enlarge the display by a factor of two. (awv only)
Zoom out	Reduce the display by a factor of two. (awv only)
Next	Scroll down to see next bunch of signals. This button appears only if there are more than eight signals in the trace. (awv only)
Prev	Scroll up to see previous bunch of signals. (awv only)
Print	Print the current page to a PostScript printer, or file, as instructed on the command line. This button appears only if a print option was specified.
Quit	Quit the program.

stv is useful for viewing test vectors before simulation. Even if you are doing an Awsim simulation, you can create STL vectors using the same random number seed and view them while viewing the Awsim outputs.

awv is useful for examining the Awsim results during and after simulation. During simulation, selecting the “Update” button includes the latest results into the display. Please note that if you run Awsim and awv on different machines, then NFS data caching may result in awv lagging behind by up to 30 seconds. In the extreme case, at startup, awv may not be able to display any vectors. In this case, you will get a blank screen. Wait a while, then use the update button to fix.

The file “awv.nodes” is awv’s node preferences file. If this file is present, then only those nodes listed in the file are displayed. A ‘#’ character can be used to comment out a node.

7.3 Limitations of Awsim

Awsim was built for performing fast simulations on custom hardware. A logarithmic number system (LNS) is used on the custom hardware to reduce the hardware cost of multiplication and division, while making addition and subtraction only slightly more complex. From the Awsim user’s manual:

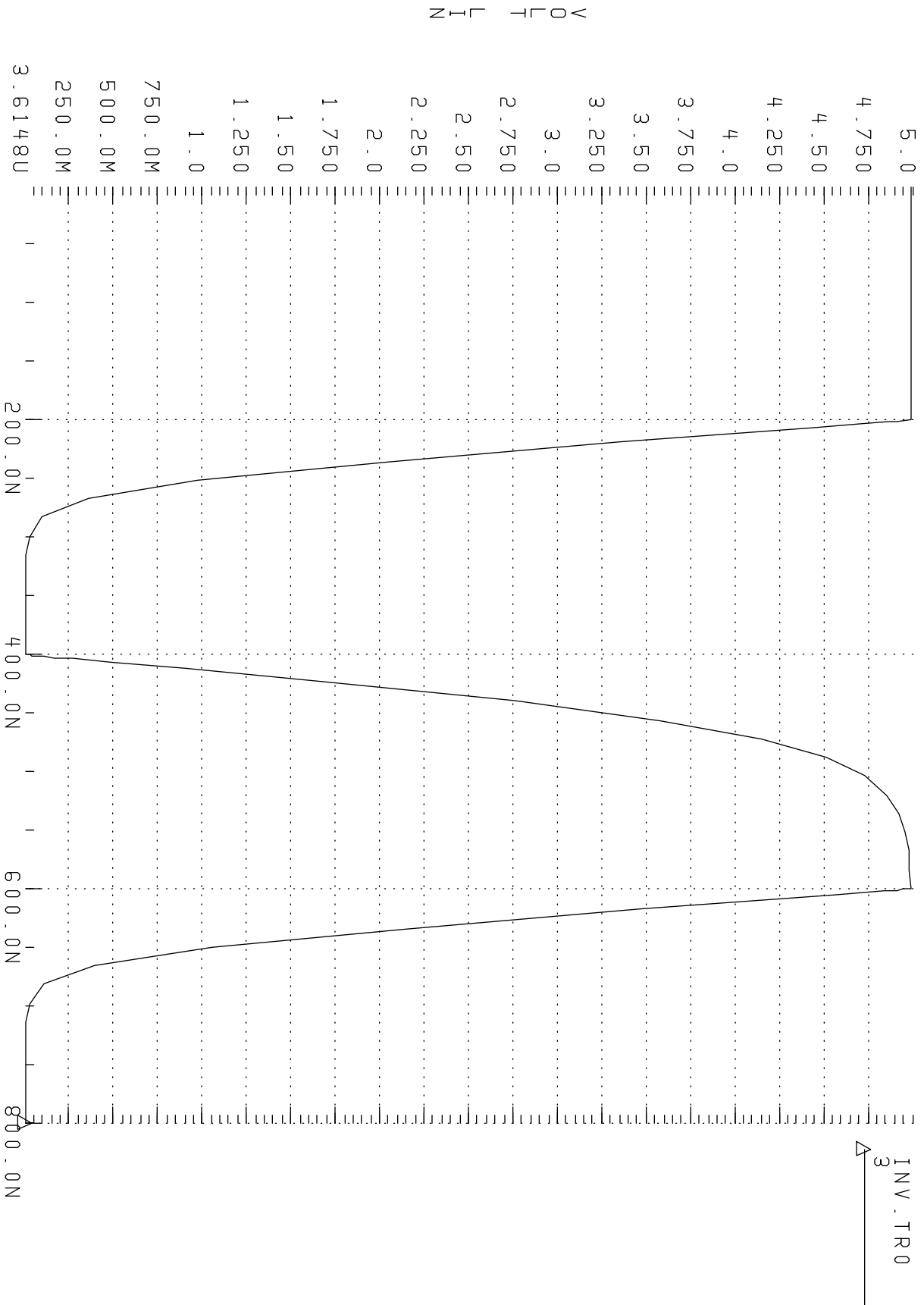
“The simulator performs the solution of the differential equations defined by the circuit in order to solve for the voltages as a function of time. The method used is forward Euler rectangular integration. If some node has a capacitance C , then $\frac{dV}{dt} = \frac{I}{C}$, where I is the sum of all current entering the node. The simulation engine solves this equation by using a small time step and integrating $V(t + \Delta t) = V(t) + \frac{I(t)}{C} \Delta t$. The engine uses lookup tables to calculate currents rapidly, and uses logarithmic representation of the quantities involved in some places to avoid the need for multiplication...”

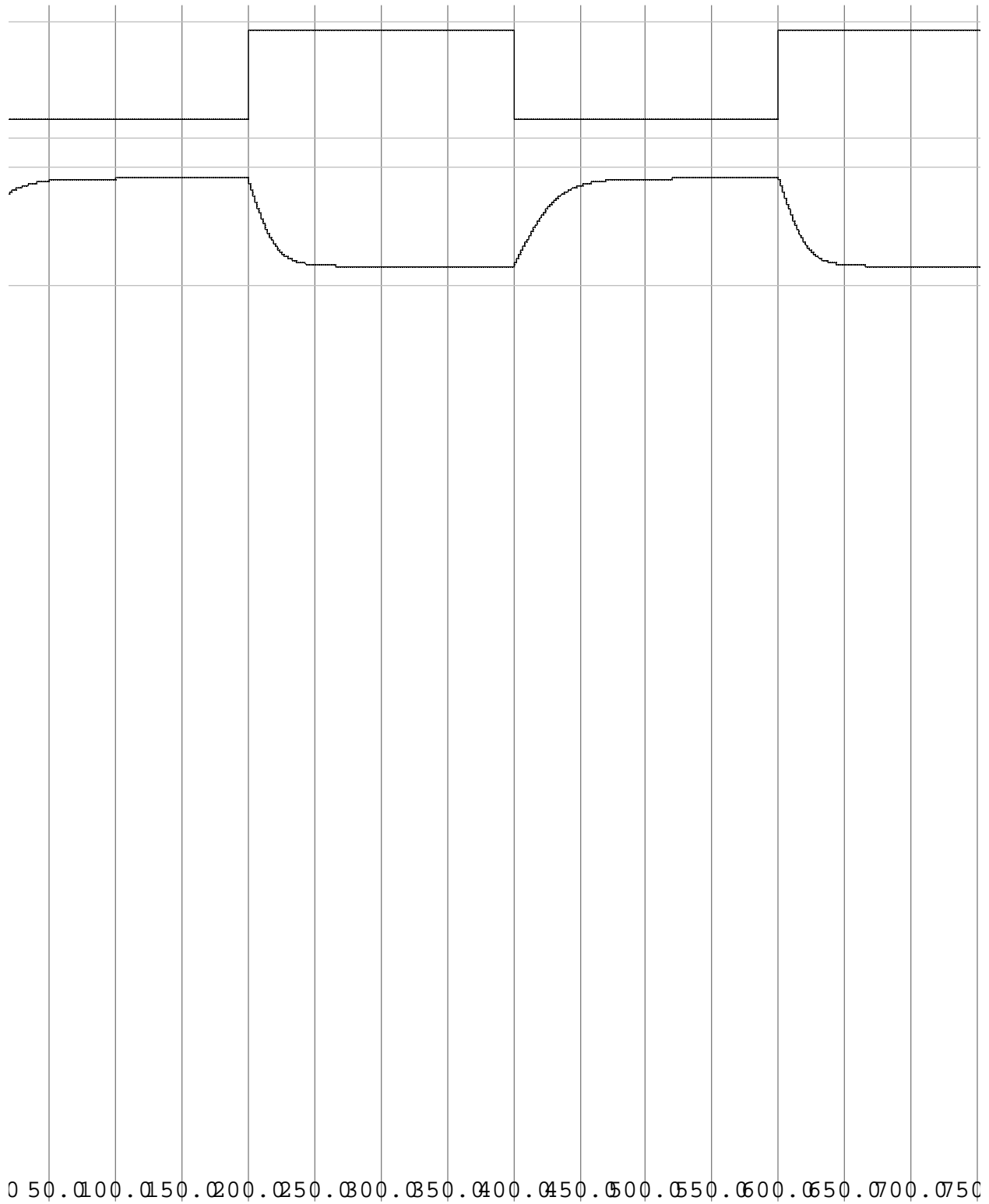
This method has several ramifications. First, since the node capacitance C appears in the denominator, small capacitances can lead to numerical instability. To prevent this, Awsim imposes a minimum capacitance on each node in the circuit. The minimum capacitance is proportional to the integration timestep such that $\frac{\Delta t}{C}$ is constant. It is expected that few nodes will require capacitance to be added to them to bring them up to the minimum. However, as the timestep increases, so does the minimum capacitance - to a point where the circuit slows down and storage elements such as latches fail to switch. Since the timestep controls the simulation speed, it is desirable to use the largest timestep possible that still allows the circuit to work. For the chip, a timestep of 0.05 ns was used. The next possible timestep (0.1 ns, the trace sampling period must be a multiple of the timestep) failed to work.

Second, since the transistor models used are much simpler than those used by HSPICE, some timing inaccuracy can be expected. We found that when properly configured, Awsim devices operate about twice as fast as the same devices simulated using HSPICE. This observation corroborates the observations of the StarBurst design team. Example runs of an inverter charging a 10 pF capacitor are shown in fig. X.

On the bright side, Awsim simulation is fast. Simulation of a 56 000 transistor chip on zeep (a SPARC 10 class machine) proceeds at a rate of 9.3 clock seconds per nanosecond. Awsim was run at a nice level of 19, and no apparent swapping took place. The memory requirements of Awsim are nice too - the chip simulation requires only 8048K of core.

21-MAY94 13: 9:29





8. CMOS4S Pads

The CMOS4S pad family has several problems in Cadence 4.2. The most troubling is that they will not hierarchically extract properly. While they may extract properly on their own, a hierarchical extract of a cell in which pads have been instantiated will usually fail to complete, and, even worse, will corrupt the system memory so that Cadence behaves erratically. Consequently all cells containing pads must be extracted flat.

There are several invisible layers in the pads which can cause trouble; if you don't understand why something is happening, make sure you have used Set Valid Layers to turn on all the normally hidden layers. The vddcore.sn pad contains a DRC exclusion layer which completely cuts the core off from vdd! when it was extracted (extract uses much of the same code as DRC). In order to simulate properly, this exclusion layer must be deleted. The pad will still DRC without errors.

There are also several resistive layers in the pads -- they are used to electrically isolate multiple gnd! and vdd! pads from each other (i.e. they will not be extracted as the same net). If these resistors are left in the pads, one should give each ground pad a different name. We found these resistors problematic -- Verilog does not understand resistors and one resistor would not extract properly -- so we deleted them. Now all the gndcore pads are electrically connected, so all pins on this net should have the same name. The same is true for vddcore!, vddring! and gndring!. When these resistors are removed, the gndcore! and gndring! nets are also connected through the substrate, so the input pins on both of these nets should have the same names. Thus, if one deletes all the pad resistors, there will only be three different names for the power supply input pins -- say gnd!, vddcore! and vddring!.

Regardless of what you call the various power supplies, be sure that they end in an exclamation mark so that they are flagged as global signals, and make sure they are all listed on the global power and ground nets lines in the Verilog Netlisting Options form before netlisting to Verilog. The pads which we modified are contained in the /pc/vrg/d1/atm94/lib/cmos4s/our_cmos4s library.

9. Design Rule Checking (DRC)

The CMC design rules for the CMOS4S technology were incomplete at the time this guide was written. The ndope and pdope width and spacing rules (rules 7.3 and 7.4) were not checked by the CMOS4S technology file, but are checked by the CMC Dracula final layout verifier. As well, there were no checks that pin metal or pin poly was covered by a drawing layer, which can be very dangerous if pin layers are not converted to CIF in order to be sent to the CMC for fabrication. One may wind up with a layout that simulates correctly, but the layout which is sent to the CMC for fabrication is missing some of the geometry. Consequently we wrote five new DRC rules to make all these missing checks; the modified technology file is /pc/vrg/d1/atm94/lib/cmos4s/cmos4s.tf.

DRC's of large circuits consume significant amounts of hard disk space (130 MB for a 56 000 transistor chip). If you are running out of hard disk space, you should setenv DRCTEMPDIR to be a directory (or directories) with lots of free disk space before you start Cadence. If you specify multiple directories Cadence will store some files in each directory.

10. Dracula Final Layout Verification

DRACULA is used as the official Design Rule Checker of the CMC. Unfortunately, Cadence 4.2 cannot fire off a design to CMC for DRACULA testing. Even more unfortunately, it cannot spit out CIF, the format you need to send off CMC (for DRACULA and for implementation). You must convert between CIF and Cadence using the intermediate format called 'Stream'.

10.1 Cadence to Stream

- start at the CIW
- Translators->Physical->Stream Out...
- press 'Library Browser' to select a cellview for checking
- select 'Stream DB'
- choose an 'Output File' name with the extension '.strm'
- press 'OK'
- view errors (using vi) in the "Run Directory"/PIPO.LOG
 - ignore warnings which say "layer not translated" as this is likely to be normal (not all Cadence layers are translated into physical layers -- pins, for example)
 - to check layer numbers, go to CIW and select:
Technology File -> Layers -> Layer Information

10.2 Stream to CIF to DRACULA

- the stream to CIF conversion utility is in /cds/bin, and is licensed to run only on Edge-licensed machines
- run the script file \$ATM/bin/dracula outfile (don't add the .strm, the script file does this automatically)
- errors, if any, will be reported on screen. the script may abort if an error occurs.
- new file outfile.cif is created
- new file outfile.cmc is created
- the 'xxx' identifier will be gNN, where NN is an integer determined from \$ATM/bin/.dracula
- mail the file to CMC
 - mail vlsiic@cmc.ca <outfile.cmc
- wait for a reply from CMC

Note: you don't have to use the dracula script file - you can do everything manually:

- run \$ATM/bin/strm2cif outfile (again, no .strm extension)
- new file outfile.cif is created
- preview the CIF file using cifxvu application by typing:
 - cifxvu outfile.cif
 - paint

- view 0
- if everything looks ok, add to the top of the file the line:
(\$DCATRxxx);
where D means dracula, CA means CMOS4S, TR is UofT, and xxx is a unique design number (so you can distinguish replies and send queries about it later)
- mail the file to CMC
 - mail vlsiic@cmc.ca <outfile.cmc

10.3 What Comes Back From DRACULA

- three or four replies, usually in the following order
 - acknowledgment from CMC that your design was submitted for DRACULA testing (this message is always sent)
 - long article describing errors and DRACULA rules file (always sent)
 - First a list of the 'ERROR CELLS'. An error cell is really a DRC rule being checked for.
 - A long list of error cells with the comment "SKIPPED BECAUSE OF NO OUTPUT-DATA" *is a good thing*, it means that error does not occur in the design.
 - An "OUTPUT CELL SUMMARY" describing which ERRORS occurred in the design (these are bad). It consists of an error cell name (ERR50230, for example), layer number, window (error region in centiDSMs), and the number of polygons involved. If no errors were in your design, this section will appear but no error cells will be listed.
 - An area describing any problems with strange geometry. Since I have never seen an error here, I don't know what it looks like. If there are no errors, the "NUMBER OF ACUTE ANGLE INPUT POLYGONS" should read 0.
 - The rest of the mail can be ignored. It is simply the DRACULA rules file in some strange programming language. It should help trace problems in case CMC changes the DRACULA rules file on you late in your design process (hey, that error didn't show up before!)
 - CIF output file from DRACULA with errors flagged (always sent). There are two ways of overlaying this onto the original design. One way is to overlay the error CIF rectangles onto the original CIF and use cifxvu to view it. The steps are as follows:
 - copy the lines in the error file from "L CERR;" down to (but not including) the "DF;"
 - paste these lines into the original CIF file, just before the "DF;" in that file
 - when you use cifxvu, the errors are boxed in purple
 - this technique is not effective for large designs since the errors will be lost in the crowd. Instead, import it into Cadence as described in the next section.
 - A text description of the rules (sent only if an error occurred). Each error cell is named ERRxxx30, where xxx is a 3 digit number. The 3 digits correspond to a rule number listed in this text; just read the plain English beside the rule number.

10.4 DRACULA CIF to Cadence

- use \$ATM/bin/cif2strm on a machine licensed to run Edge:
rsh edge_machine.vrg \$ATM/bin/cif2strm /fully/qualified/path/filename
(no .cif extension!)
- the script writes a file filename.strm
- make a new library in Cadence, named dracula_results, for example, specifically for importing these CIF files in
- from the CIW do Translators->Physical->Stream In...
- set the “Run Directory” and “Input File” as appropriate
- set the “Library Name” to dracula_results
- set the Scale UU/DBU to 0.1
- press OK
- a number of cells will be created in dracula_results:
 - topsymbol, this is an instance of outecadtoplevel
 - outecadtoplevel, this contains instances of the error cells
 - numerous errxxx30, one for each rule violated
- you may wish to fully flatten topsymbol, since you will be overlaying it on top of your design
- bring up your design layout and press i to grab an instance of topsymbol
- move the MOUSE POINTER to the ORIGIN (ignore the corners of the topsymbol Cadence is drawing!) and place the instance down. press Esc to stop dropping instances.
- **THE ORIGIN OF THE DESIGN MUST NOT HAVE CHANGED SINCE THE CIF FILE WAS CREATED AND SUBMITTED FOR TESTING**
- to view the errors, you might need to add a layer to the techfile. make sure layer #35 exists. You will probably want it blinking and have a large priority so it is drawn last. Also, be sure to set the layer as VALID in the LSW. Be sure to save the techfile so you don't have to add this layer again!!!
- press Shift-F to view all levels of the hierarchy -- you can probably see the blinking layer #35
- if your design is too complex or big, you may have to flatten the topsymbol and view only levels 0 to 1 in the hierarchy. Do this by changing Design->Options->Display->Display Levels.

11. Hierarchical Design

A hierarchical design methodology is one in which extracted layouts can be substituted into a global schematic of a chip, replacing the schematic representation of some subblocks. This allows one to simulate the entire chip, using a fixed set of test vectors and expected responses, at each stage of layout. If the chip no longer works after a layout/extraction step, the last layout created is incorrect. This approach avoids the tedious procedure of determining appropriate test vectors and the expected outputs for each subblock, and because the test vectors used to test the chip are generated only once and are hence more complete, hierarchical design tends to find errors in subblock

layouts that testing the subblocks by themselves would likely miss.

In order to use an hierarchical design methodology

- use exactly the same hierarchy of cells in the layout and the schematic;
- use exactly the same pin names (including case) in the layout and the schematic;
- include vdd! and gnd! pins in each schematic cellview.

The vdd! and gnd! pins must be included in the schematic because the extracted view contains them, and if the schematic view does not there will be pin mismatches when extracted and schematic views are mixed. The top level of the schematic contains the vdd! and gnd! supplies. Hierarchically extract the layout(s) to be substituted into the schematic. To netlist a design in which extracted views will be used wherever possible, one's netlisting options should include (note that the order of the views is important):

Netlisting View List: verilog extracted schematic symbol

Netlisting Stop List: verilog symbol

The above methodology never worked completely for us because of a bug in the Cadence schematic editor. Each schematic cellview contains vdd! and gnd! inputs. When one must connect another input to vdd! or gnd! (say an enable input) Cadence will not make the connection, presumably because vdd! and gnd! are global signals. The schematic looks correct, but will not simulate properly. Use the probe command available in the schematic editor to see if the different hierarchical levels are being connected properly by Cadence.

We have thought of a possible workaround to this problem, but it has not been tested due to time constraints. This potential workaround is to never tie any input pin (except vdd! and gnd!) to a supply. If you need to tie some other input high or low, create a special cell whose output is always high or low and use this to generate the input signal.

12. Schematic Generation from Verilog Netlists and Layouts

It is possible in Cadence 4.2 to generate schematic cellviews from layout cellviews, although it is a laborious and error-prone process, fraught with much pain and anguish (at least the first time through). The basic idea is to create transistor-level schematics and symbols for all the basic cells in the layout, then the translator can take a hierarchical verilog netlist and create schematic views for all the higher-level blocks from the netlist. The netlists must be massaged a little first, however, and the schematics need to be checked (although not at a great level of detail) after generation.

12.1 Doing a Translation

First of all, a distinct library is probably a good idea to do all schematic conversion in. This library needs a few basic cells before translations to it will work properly. First, the cells ipin, iopin, and opin must be copied from the "basic" library. The transistor cells "tranif0" and "tranif1" must also be copied from the "sample" library. With these files, pins and transistors will have the correct symbols when a translation is done (otherwise the translator will generate a symbol for each of the

transistors--a box).

From the CIW, select Translators -> Verilog in... This will pop up a dialog box with much information. In order for the translation to work properly, the following procedure is recommended:

Remove all library names from the Reference Libraries field. Put the library where the schematics should be generated in the Target Library field. In the field Verilog Files, fill in the names of the files containing the netlists that are to be translated. You may want to verify that the search path is correct, just in case.

Press OK, and the translation should proceed. If the netlists have been properly primed (see the sections below), there should be no errors, and the schematic and symbol cellviews for all the netlists should be created in the target library. The translator will not overwrite any existing cells in the target library. If any errors occur, their existence will be reported in the CIW, and a file called verilog.log will be created in the directory where the Cadence session was started.

12.2 Translator Input Requirements

The translator, while claiming to take a Verilog netlist to a schematic cellview, falls somewhat short of this ideal. Certain features of Verilog confuse it to the point that it fails. A Verilog netlist that is to be translated must be massaged prior to actual submission for translation. The changes that we are aware of are listed below:

Power supplies must be removed and replaced by wires with the same names. This requires that the supply pins must be re-inserted into the schematic after it is created by the translator. Modifying the schematic is discussed in a later section.

```
e.g.  
supply0 global_0;  
supply1 global_1;
```

These must be replaced by:

```
wire global_0, global_1;
```

Any specifications in the Verilog netlist must be deleted. When Verilog netlists are generated by Cadence, they typically have several lines bracketed by the “specify...endspecify” commands. These must be completely removed from the netlist.

```
e.g.  
specify  
specparam CDS_LIBNAME = "input_chip"  
specparam CDS_CELLNAME = "or2"  
specparam CDS_CELLNAME = "extracted"  
endspecify
```

All five lines must be completely removed from the netlist in order for the translator to work properly.

Global signals can/should be removed from the port list and the list of inputs/output. Though this may not be necessary (we found that it helps), it makes the schematic generated a little less congested since the gnd and vdd signals would be routed to every instance in that view (if it was translated from a layout).

These three modifications should be sufficient to ensure correct translation of Verilog netlists to schematic and symbol cellviews.

12.3 Necessary Schematic Massaging

Once the schematic is generated, some inspection is required to ensure that the translator didn't do something stupid (which it's been known to do). First of all, because the supplies were turned into wires, the ground and vdd pins must be inserted and connected in the schematic. Every wire in the schematic will be labelled with the name of the node that was used in the Verilog netlist, so finding these wires should be easy. Once vdd and gnd are connected, Check and Save the schematic view (key 'X' if the bindkeys stay the same). There should be four errors: gnd! and global_0 are shorted, and vdd! and global_1 are shorted (each twice--don't worry it's the same error). The names global_0 and global_1 may not be exact, but they should be the same as were in the original Verilog netlist. Make sure that the appropriate name is connected to vdd! and to gnd!. Once this has been verified, the wire labels for the global_0 and global_1 should be deleted wherever they occur in the schematic. This will eliminate the errors. There may also be some other errors or warnings for the schematic. A common mistake that the translator makes (though not that often) is to incorrectly short two wires together. This will produce an error of the form "net1 shorted to net2" or some such. Most warnings are simply regarding solder dots and can be safely ignored. Other warnings regarding nets with no fanout should be inspected carefully. They often arise because of debugging (i.e. superfluous) pins that were put on internal nodes of a layout that aren't actually used in upper-level cells. These can also be ignored.

12.4 Creating Low-level Cells

There are a variety of ways of creating the low-level schematics and symbols. One is to manually enter them into the schematic editor. However, in the event that this is an unpleasant alternative, it can be done semi-automatically. The approach that was taken with this project was to take flat HSPICE and Verilog netlists for the cell being created. A flat Verilog netlist was generated from the HSPICE netlist using the hsp2vl utility to replace the body of the module (as was done in order to do flat verilog simulations). This netlist was then modified using the rules above to make it suitable for translation. The translator was then used to create a schematic and a symbol cellview for the cell, and then some more massaging (as described above) was done on the schematic to make it usable.

12.5 Creating Higher-level Cells

Once all the low-level cells have been created, the translation of all higher-level cells can be done in one pass. A hierarchical Verilog netlist is required of the top-level cell, massaged as described above. The Verilog netlists that were used to create the schematics are also required. Otherwise, they must be generated from the schematics (by doing a Verilog netlist from the schematic view), and massaged as described above. Put all the netlists (both lower-level, and all the netlists generated from the hierarchical netlist) into one file. The hierarchical netlists can be found in the ihnl/cds*/netlist files in the run directory where the netlist was made. The easiest way to do this is:

```
% cat ihnl/cds*/netlist low_level_blocks >new_blocks
```

where low_level_blocks is a file containing the concatenation of all the lower-level-block

netlists. All the modules must be modified as described above or the translation will not work. A translation of the file `new_blocks` will create schematics and symbols for all the modules that are defined in the file unless a cell with that name exists already (i.e. it will not overwrite cells).

The pins will be exactly the same as in the layout. This applies to buses as well. In the layout, the wires of a bus are all distinct, and each of these pins will be created in the schematic as well. This may not be the desired effect when simulating however, if a bus is expected. The problem is that the pins won't be combined into one input or output for the simulator. For example, a signal `dest<3:0>` will appear in the schematic as pins `dest<0>`, `dest<1>`, `dest<2>`, and `dest<3>`. The port list for the schematic and the symbol will contain these four names, not `dest<3:0>`. Thus, when simulating, the `input.stl` file will contain `dest<0>`, `dest<1>`, `dest<2>`, `dest<3>`, and not a bus `dest<3:0>`. Each bit will have to be specified individually. The only way to fix this is to modify the schematic so that there is a pin called `dest<3:0>` (replacing the four individual pins) from which the `dest<0>`, `dest<1>`, `dest<2>`, and `dest<3>` wires are extracted. Be careful, because the translator tends to name bus wires that it creates from low bit to high bit (e.g. `dest<0:3>`, not `dest<3:0>`). The symbol pins will also need to be modified to reflect this change. Less obviously, the port lists for the schematic and symbol views need to be changed. To change these, first deselect everything in the view. Then type 'q' to see the cellview properties. There will be a property called `ihlFormatInst` which contains the list of inputs in a specific order. This is the order that the netlister uses when it generates modules from the view. The individual pins must be deleted and the new bus pin inserted. The notation must be exactly as on the pin (i.e. the ordering of the wires low-to-high or high-to-low must be kept consistent). Both the schematic and the symbol have port lists, and both must be updated and be identical or a netlist of the cell will fail.

13. Time/Space Requirements of Tools

The table below lists the time and disk space required for various circuit simulators and Cadence procedures performed on a 56 000 transistor chip designed in the CMOS4S process. All programs were executed on SPARC 10 machines with 32 MB of physical RAM except for the Hierarchical Verilog simulation of the chip, which required 40 MB of RAM to run and was therefore run on a SPARC 10 with 64 MB of RAM, and the HSPICE simulation which was run on zeep because of its large time requirement. The Virtual Memory size includes temporary files written out (such as the `DRCTEMPDIR` files).

Table 4:

Operation	VM size (MB)	Time (h:mm)
Awsim simulation of chip	8.4	70:50
Hierarchical Verilog (gate-level) of chip	50	0:20
Flat Verilog (transistor-level) of chip	30	0:30
HSPICE of control and 1 output slice (approx. 1/10 of chip)	52	217:20
Hierarchical extract	-	0:03
Flat extract	130	2:00
Flat DRC	130	2:30
cmos4sFixExtract() SKILL function on chip	90	5:00
our_globalize() SKILL function on chip	80M	0:04