Random SPICE Simulation Flow

Introduction

The Random SPICE Simulation flow runs user-specified circuits through the SPICE simulator. The goal is to run both random and custom test vectors while monitoring test-vector output mismatches and delay time violations. Those values are compared against timing models or user-defined timings for each circuit, and violations are reported.

A wide variety of run parameters can be varied. Timing models that aren't available for a particular corner can be scaled from ones that are. And it runs continuously on all circuits so that maximum coverage is achieved.

Flow Cabilities

The flow has two major compelling capabilities versus doing things the old way (manually). First, it can run jobs while you're not around, scheduling them to run repeatedly and forever without worrying about hogging the queue. This means that you will have a greater amount of random vector coverage for your circuit. It also makes it easy for you, since you don't have to look at the results each time, total up how many test-vectors you ran, keep track of total run time, manage the actual job submission, etc. All of that is done for you by the flow.

Second, the flow provides a high-level interface to your circuit and timing model. This allows you to derive the timing information from a timing model rather than having to calculate it yourself. It also allows you to do things like specify different SPICE technology files, specify unique input slews for signals, add loads to the output pins, etc.

The effect of both of these capabilities is to free you up to do more important things and to reduce human error.

The flow has many capabilities already and will likely grow in the future. You can take a look at the control file to see all of the different properties you can set to control your runs.

Here's a list of some of the flow's many features:

- The flow runs forever. It will never be stopped unless it's the end of the project and we no longer need to run random vectors. The goal is to maximize spice simulator license usage and to always have random vectors simulating on all of the cells.

- Automated jobs are typically run using a "batch" queue. These queues are setup so that jobs running in them get killed immediately and rescheduled if someone needs to run a job in the normal queue. This way, the automated system doesn't get to monopolize the licenses. It's only when a license is idle (when nobody wants to use a license) that the automated system will wake up and use it.

- There's also a manual-mode operation of the flow for when you need immediate feedback. It lets you yourself run the flow rather than waiting for the automated flow to eventually run it.

- You can vary the SPICE technology file ("tt_tt", "chgsh", etc.).

- Specify the SPICE simulator (hsim, hspice, xa, etc.).

- Vary clock speeds.

- Specify whether to run a setup-time analysis or a hold-time analysis.

- Specify whether or not to check delay times.

- The timings come directly from a timing model which you can specify or let it find on its own. It does all the work for you by reading the timing model and deriving everything from it. Pearl timing models (PTM) and Synopsys Liberty timing models are supported. Cadence TLF timing models are still supported but are now deprecated.

- Define your own timings if you don't have a timing model. This includes arrival times of all inputs, output check windows, setups/holds, delays, and so on. You could use this feature to override timings found in the timing model as well.

- If you do have a timing model, you may need to scale it to another corner, and that's a capability built-into the flow. You can multiply timings by a certain scale factor (a different one for setup times, hold times, and delays if you want). You can add or subtract a constant from each timing value. You can do this using Perl regular expressions to select which signals you want to apply it to, or you can specify the names of the signals and the edges explicitly.

- Add margin to the analysis anywhere: delays, setups, holds, input arrival times, output check windows, etc.

- Specify input slews on a pin-by-pin basis or set general defaults for clock and data input pins.

- Specify output pin loads. These get added to your SPICE netlist and taken into consideration during the timing analysis calculations.

- The flow is completely customizable. All of the run parameters are completely under your control through the

control file properties. Every script that gets run is actually an option you can control. That means if you have your own script for running hsim, you can use that script instead of the default command to run hsim. Everything is customizable. It's done this way so that there's always a way around any issue that might come up specifically for your circumstance.

To Do:

- Presently the flow doesn't work for timing models that aren't at the same level as the ECKT level. (Even though this documentation implies it works, it doesn't.) This is the case for cells where your ECKT has the clock regen circuit in it and drives the main timing model cell. Well, this is a bit tricky to handle automatically, considering that the flow would need to use Pearl to extrapolate the embedded timing model to the ECKT boundary. That part of the flow is coming. For now, you need to have your eckt_cellname property defined to be the same as your cell property. And so you'd have to define your timings by manually entering them into the control file since there is no timing model available at the ECKT level.

  This is pretty easy to do. You could simply take the timing model that you do have, copy it somewhere, modify it, and then point to it in the control file by setting the timing_model property. In the copied timing model, you would rename the cell to the ECKT cell. Then you'd rename the clock from "ph1" to "eph1". Then you need to measure the eph1 to ph1 delay time (manually outside of the flow perhaps). Then use the arrival_add property in the control file to adjust the arrival times for all non-clock inputs, subtracting the eph1 to ph1 delay amount from each. It's not very difficult at all, but an automated way of doing it would be better.

- Half-cycle paths are not understood by the flow very well. I still need to investigate this. It would affect the way setup and hold times are measured.

- I'm open to suggestions for future enhancements.

Flow Diagram

The main part of the flow is shown below as a flow diagram. This is what happens inside of one of the "rundirs":

$CHIP/random_spice/running/$userdir/$rundir/

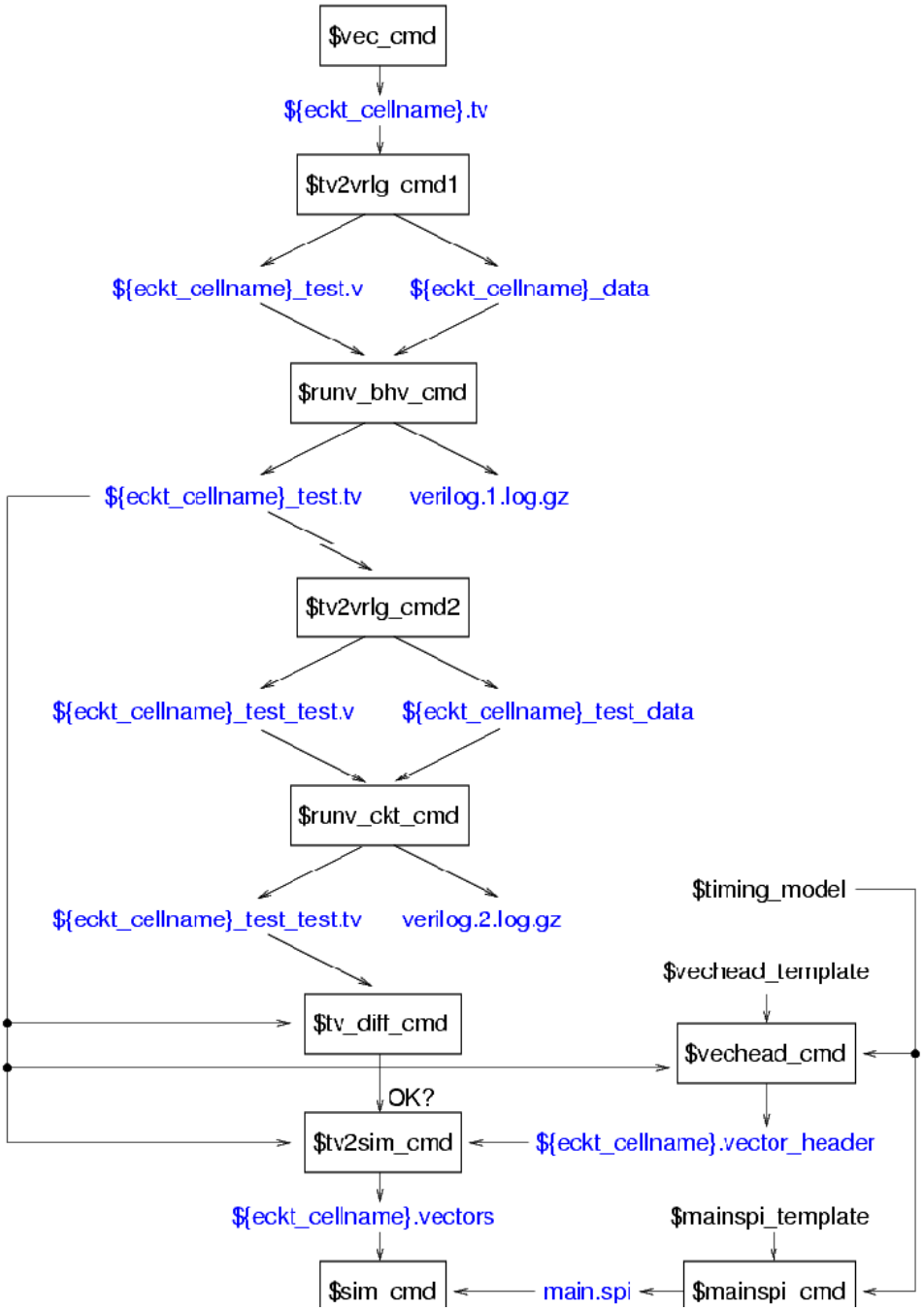
It doesn't show that the flow looks at your control file to figure out which rundirs it should run. That part of the flow is outside of the scope of this diagram.

Keep in mind that this is just showing the basic idea behind the flow. There are some other little things that go into the flow which aren't shown. For example, a CRC-32 checksum is performed on the "${eckt_cellname}.vectors" file and the "main.spi" file to determine if anything has changed since the previous run thereby causing the simulator to rerun, but this isn't shown on the diagram.

In addition to this, there may be input files that aren't labeled on the flow diagram. And not shown is the fact that some of the scripts in the flow make direct use of the control file itself as an input file, even though the control file isn't shown explicitly as an input file.

So the flow diagram below just shows the basic idea behind the flow and most of the flow's target files. One thing to note is that every command that is run is completely customizable. See the control file description for more information about customizing the flow and to find out what all of the properties refer to.

Items colored black are all under your control and represent the flow's inputs. Items marked in blue are the required outputs of the flow. See the control file section for a description of all of the "$" property names.



So the basic idea behind the flow is to generate some input vectors using a user-supplied vector generator script ($vec_cmd). Output vectors are generated by simulating the input vectors using the behavioral level model in a verilog simulation ($runv_bhv_cmd). Output vectors are also generated a second way by using a circuit level verilog simulation

($runv_ckt_cmd), and the resulting test vector output file is compared against the output based off of the behavioral level netlist ($tv_diff_cmd). Any important differences (outputs which aren't X's) causes the flow to abort. Then the resulting behavioral level input/output vectors, along with timing information from a timing model and/or the control file, are turned into a SPICE vector file ($vechead_cmd, $tv2sim_cmd) and a main SPICE file ($mainspi_cmd). At that point, a SPICE simulation is done ($sim_cmd) to compare the input vectors against the output vectors. And lastly, a summary program looks at the results and creates a summary file ($sum_cmd).

Using the Manual Version of the Flow

There are 2 versions of the flow: Manual Mode and Auto Mode.

Sometimes you need immediate feedback, and you're not willing to wait for the flow to detect your new control file and run with it. Instead, you want to debug your circuit right now. This is exactly what the manual flow is for.

The manual-mode operation of the flow works just like the automated version, except it runs only the control files you want it to run. It submits your jobs to remote machines and waits until they return. And it does it immediately.

Here's what you need to do in order to operate the manual flow.

1, ***source /c0319/jasonchen/Central/RSS/cnq/cshrc_rss*** , which define $CHIP, $CHIPBRF, $CENTHOME, and $CADHOME environment variables used by RSS flow.

2, If you have a cell you'd like to run through the flow, create a directory for it at:

$CHIP/random_spice/user/$userdir/$cell

The name of the directory ($userdir) can be any name you want it to be. I recommend naming it the same as your username.

3, prepare source files

3.1 technology files;

3.2 load file

3.3 behavior verilog model

3.4 ckt level verilog model

3.4.1 For ncverilog issue, may be you should modify the verilog model.(about this issue, please see FAQ).

3.4.2 If no verilog available, you can create a fake one only include IO pins and put it under dir:

/c0319/keithm/cnq/random_spice/vmd

Note: you should set *$rss{$rundir}{"skip_tv_diff"} = 1*; in you control file

3.5 circuit netlist

3.6 random vector script file and test vector header file

3.6.1 generate the test vector header file and a temperate random vector script file use maketv

/vlsi/cad2/bin/maketv -i /vlsi/centaur/cnq/lib/vnet/$cell.v -o $cell.tvhead -t -n -e $clk_name

-i input verilog file name;

-o output TestVect header file

-t "template" option. Creates an executable PERL script.

-n Set up template to do input stimulus only (no expected outputs)

-e specify "early clock" pin(s).

3.6.2 modify the temperate random vector script file to meet your requirement

…

4, create a control file and put it under dir:

(you can copy this file from ***/vlsi/centaur/cnq/random_spice/user***)

$CHIP/random_spice/user/$userdir/$cell

Note: (1) The control file is at the heart of the flow. It allows you to set variables which tell the flow exactly what you want it to do. You need to specify which cells you want to run, where the random vector script file is, which SPICE corner to use, etc. Details please see **The Control File**.

(2) make sure all files defined in your control file is available.

5, before you run the simulation, please first run the flow's control file checker program on it:

rss_flow_chkcf.pl control.pl | & tee check.out

The command line above will check the " control.pl" file for any mistakes. It also shows you all its calculations so that you can determine if there are any obvious problems before running with it for real. It's okay if the rss_flow_chkcf.pl program shows a "WARNING" message. Those messages are there to caution you, but they're not necessarily pointing out that something is wrong. The flow will run just fine with these messages. It's only when you get an "ERROR" message that it's actually broke, and so the flow wouldn't attempt to run with it.

6, run the simulation.

rss_flow.pl –ctrl $userdir/tmp.control.pl

The example above would cause the flow to look for your control file at:

$CHIP/random_spice/user/$userdir/tmp.control.pl

Eventually, it completes, and the results should be at:

$CHIP/random_spice/manual_results/$userdir /

This is the default location where all manual flow results go. And it runs in this default location:

$CHIP/random_spice/manual_running/$userdir /

If you do not want to run the flow under the default directory, you can specify the location as below:

rss_flow.pl -ctrl /c0319/keithm/cnq/sim/presim/glregi_x4_10/control.pl #specify the control file
-workdir /c0319/keithm/cnq/sim/rss/manual_running #manual running dir
-resultsdir /c0319/keithm/cnq/sim/rss/manual_results #manual results dir

The flow is divided into 4 parts, when the flow running crash, a directory begin with "part1", "part2", "part3", or "part4" will create under dir: $CHIP/random_spice/manual_results/$userdir/crashed_run/. This indicates which part in the flow it crashed at. Part 1 is just using the rss_flow_chkcf.pl program to dump out your control file values. If you're crashing in part 1, then it's telling you that something is wrong with your control file. If you're crashing at part 2, then it's telling you that there's something wrong when it tries to do the data prep stuff to create the test-vector file and such. If it crashed at part 3, it's telling you the HSIM simulation command had a problem. And if it crashed at part 4, it's telling you that the summary command and post-processing scripts had a problem.

You can generally see the error messages by looking through your trace files:

$CHIP/random_spice/results/$userdir/crashed_run/part*/*.trace.txt

If the control-file-dump jobs crash, the output will be copied back to the "results" directory at:

$CHIP/random_spice/manual_results/$userdir/crashed_run/part1/

So you should always check to see if there's a "crashed_run" directory and if there's anything in it. Crashed runs occur due to two reasons. Either the job really did "crash" due to some problem getting the job to run in the queue (a very rare malfunction), or it crashed in the sense that the exit status code was non-zero after running "rss_flow_part1.pl". That program just runs the "rss_flow_chkcf.pl" program and monitors its exit status. The exit status is non-zero if there's an "ERROR" message anywhere in the output. "WARNING" messages are okay and will result in exit status code 0.

So the flow will first take a snap-shot of all of the control files. Then it reads the results to figure out the names of all the rundirs. Then it spawns off a data prep job to build the "main.spi" and "${eckt_cellname}.vectors" file for each rundir. This occurs in parallel and is distributed to as many machines as needed using the PSub program.

If the data prep jobs crash, the output will be copied back to the "results" directory at:

$CHIP/random_spice/manual_/results/$userdir/crashed_run/part2.$rundir/

Note that the "$rundir" value comes from the control file.

The "part2.$rundir" directory indicates that the job crashed at some point or one of the commands it executed returned a non-zero exit status code. You should look at the files in this directory (if it exists) and make any changes necessary to ensure an exit status code of 0 for all commands. If you're not sure how to do this, see the flow owner, Steve Weigand.

If it didn't crash and everything ran okay, it then looks through the data prep results to determine if anything is different compared with the results of the previous run's results contained in the results directory:

$CHIP/random_spice//manual_results/$userdir/$rundir/

- vs. -

$CHIP/random_spice//manual_running/$userdir/$rundir/

It looks for changes of three types:

1. There was no previous rundir by this name. This is the first time it was run.

2. The circuit or timing information has changed since the last run.

3. The test-vectors have changed since the last run.

If any of the above changes occurred, it will rerun the flow for the given rundir. This means that the simulation (hsim) job will be rerun, followed by the summary command.

The order of the changes listed above is important. The first type of change takes priority over the second type of change. And the second type of change takes priority over the third. This means that a rundir that has never run before will be run before any other jobs are run. Similarly, if there's been a netlist or timing model change, then that takes priority over a rundir where the only change is the test-vectors used.

7 The results

After the flow runs your cells which you've defined in the control file, the results will be available at:

$CHIP/random_spice/manual_results/$userdir/$rundir/

A summary file will be located at:

$CHIP/random_spice/manual_results/$userdir/$rundir/$eckt_cellname.sum.gz

Here's an example summary file:

Summary Date : Wed Jul 16 21:53:41 2003
Simulation Date : Wed Jul 16 21:53:35 2003
Simulation Machine: fb129

	Now	Ever
Total vectors	106	1166
Passing vectors	59	651
Failing vectors	47	515
Failing timing checks	10	110
Sim CPU time	0:22:40	3:58:19
Sim peak memory use (MB):	101.4	101.6
Sim transient time (s)	2.65000e-08	2.91500e-07

Reason(s) for new run:
- Different vectors.

This run's test vector mismatches:
DOUT Error: t=0.95 ns expect=1 state=0 node=rddat6
DOUT Error: t=0.95 ns expect=1 state=0 node=rddat4
DOUT Error: t=0.95 ns expect=1 state=0 node=rddat2
...

This run's timing check violations:
checkdelay_ph1_r_qb1_r: Ref=v(ph1) Sig=v(qb1) Tref=1.9013e-08 Tsig=1.913e-08 (Tsig-Tref)=1.17e-10
checkdelay_ph1_r_qb1_r: Ref=v(ph1) Sig=v(qb1) Tref=2.6513e-08 Tsig=2.6634e-08 (Tsig-Tref)=1.21e-10
checkdelay_ph1_r_qb0_r: Ref=v(ph1) Sig=v(qb0) Tref=2.7513e-08 Tsig=2.7633e-08 (Tsig-Tref)=1.2e-10
...

The example summary file above is pretty self explanatory. The column labeled "Now" represents the current run. The column labeled "Ever" represents the current run and all of the previous runs. It's a running total of whatever is in the "logfile" file.

The "logfile" file is kept here:

$CHIP/random_spice/results/$userdir/$rundir/logfile

This log file isn't the HSIM log file. It's the flow's special log file. It details when the flow was run, why it ran, and what the summary information showed for it. Here's an example logfile:

@ Thu Jul 14 17:01:33 2003
Machine: "fb131" CPU time: 0:00:31 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/219/187
Timing Check Violations: 15
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- First time this rundir was ever run.

@ Thu Jul 14 17:41:36 2003
Machine: "fb127" CPU time: 0:00:32 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/218/188
Timing Check Violations: 10
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- Different vectors.

@ Thu Jul 14 18:21:37 2003
Machine: "fb131" CPU time: 0:00:31 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/220/186
Timing Check Violations: 0
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- Different vectors.

@ Thu Jul 14 18:58:34 2003
Machine: "fb132" CPU time: 0:00:32 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/216/190
Timing Check Violations: 12
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- Different vectors.

Notice that each run begins with "@ $date". This date is actually when the flow completed the simulation run, not when it started the run.

Each time a new run is performed, the previous run is archived to the "previous_violations" subdirectory if the previous run had any vector mismatches or timing check violations. You can have multiple "previous_violations" subdirectories:

$CHIP/random_spice/results/$userdir/$rundir/previous_violations/archive_1/
$CHIP/random_spice/results/$userdir/$rundir/previous_violations/archive_2/
$CHIP/random_spice/results/$userdir/$rundir/previous_violations/archive_3/
... etc.

These "previous_violations" archive directories are numbered starting at 1. Each time a new run is made, the previous failed run will get archived. The number of archived directories is limited by the limit_kept_logs control file setting which defaults to 4. If this limit is reached, no new runs will occur.

At this point, if you have any failing test-vectors or timing-check violations, then you can modify the control file and/or the circuit to fix the problems and rerun.

IMPORTANT: When you change the circuit, or the SPICE technology information changes, or the timing information changes, then the "previous_violations" directory and all its archives will be deleted. The "logfile" file will also be deleted. This is just like starting over from scratch. This doesn't occur if just the test-vectors change. There must be some change in the circuit or the timing information for the logfile and previous_violations directory to be wiped clean.

NOTE: Remember to set your $CHIP, $CHIPBRF, $CENTHOME, and $CADHOME environment variables appropriately! The flow needs these variables to determine where the flow templates and SPICE technology information

is kept(for RSS flow these variables defined in */logic/keithm/.cshrc_rss)*.

An Example:

/c0319/keithm/cnq/random_spice/user/keithm/gdyn_or8x8_x64_40
Control.pl: A control perl file example;
My_commands: commands file to run rss_flow_chkcf and rss_flow

Using the Auto Version of the Flow (not available now)

Below we will explain Auto Mode.

Where is the flow located?

The flow is housed in the following directory:

$CHIP/random_spice/

All user-input and control files are specified in:

$CHIP/random_spice/user/

All of the automated runs take place in:

$CHIP/random_spice/running/

And all of the automated results are stored in:

$CHIP/random_spice/results/

When the flow runs, it runs in the "running" directory. When it's done, the results are copied into the "results" directory where they can be viewed. Then the "running" directories get deleted.

You are only permitted to modify files that are in the "user" directory, not the "running" or "results" directories. Do not attempt to modify anything that's in the "running" or "results" directories, even if all you're doing is uncompressing a gzipped file.

A big summary report file is reported at:

$CHIP/random_spice/Summary/all_cells.sum

That summary file summarizes the results for all "rundirs" (individual jobs) that were run by the flow. It's at most 2 minutes out of synch with the "results" directories, so you can rely on it being up to date.

In addition to those directories, a "templates" directory is used by the flow and should only be modified

by the owner of the flow (Steve Weigand):

$CHIP/random_spice/templates/

The "templates" directory contains files to set default settings as well as template files for generating things like the main SPICE file and the SPICE vector file.

How do I create and run my jobs?

If you have a cell you'd like to run through the flow, create a directory for it at:

$CHIP/random_spice/user/$userdir/

The name of the directory ($userdir) can be any name you want it to be. I recommend naming it the same as your username. If that doesn't sound good, you can give it the name of a module or a cell. Whatever you want. It doesn't matter to the flow. It's merely used to contain cells you want to run. You must limit it to the following characters: a-z, A-Z, 0-9, "_" (underscore), and "." (dot).

You should not create a corresponding "results" or "running" directory for your runs. The flow will automatically create them for you when it detects your new "user" directory.

The $userdir directory is entirely under your control. No automated tasks will output to any "user" directory. If you no longer want the flow to run on the cell, simply remove the entire directory, and the flow will eventually delete the corresponding "results" directory.

Next, you'll need to create a perl script which will generate the random vectors for each cell. If you're used to using the old manual random spice simulation flow, your old script can be used with little or no modifications here. You can give this file any name you want, and you would point to it in the control file.

To keep it simple, just name this random vector generator script as follows:

$CHIP/random_spice/user/$userdir/$cell.random_vecs.pl

Next, you'll need to create a control file which contains some variable settings which are used by the flow. The variables you define will setup all of the runs that you want to perform. The existence of this file is what is used by the flow to start running your jobs. You can run any number of different cells with varying permutations of run parameters. This file is in Perl language format. You would place this control file at:

$CHIP/random_spice/user/$userdir/control.pl

The control file will be described in depth later on.

Before you actually name the control file "control.pl", it might be best to first call it something else, like

"tmp.control.pl". This is because the flow specifically looks for a file called "control.pl" in any $userdir directory. If it finds one, it will use it. But what if you're just in the middle of creating the file and you're not too sure if it's good or not? Then you'll want to first call it "tmp.control.pl" and run the flow's control file checker program on it:

rss_flow_chkcf.pl tmp.control.pl

The command line above will check the "tmp.control.pl" file for any mistakes. It also shows you all its calculations so that you can determine if there are any obvious problems before running with it for real. If everything looks good (no "ERROR" messages), then rename the file to "control.pl". At that point, the flow will eventually detect it and will run it.

It's okay if the rss_flow_chkcf.pl program shows a "WARNING" message. Those messages are there to caution you, but they're not necessarily pointing out that something is wrong. The flow will run just fine with these messages. It's only when you get an "ERROR" message that it's actually broke, and so the flow wouldn't attempt to run with it.

When do my jobs get run?

The flow works by taking snap-shots of all of the control files it sees at the same time. It evaluates them all more or less at the same time and dumps the results to output files at:

$CHIP/random_spice/running/$userdir/control_file_dump.txt

This file is the result of simply running the "rss_flow_chkcf.pl" program on the control file and dumping its output to that file. So what you see when you run this program by yourself at your command line is exactly what the flow uses.

If the control-file-dump jobs crash, the output will be copied back to the "results" directory at:

$CHIP/random_spice/results/$userdir/crashed_run/part1/

So you should always check to see if there's a "crashed_run" directory and if there's anything in it. Crashed runs occur due to two reasons. Either the job really did "crash" due to some problem getting the job to run in the queue (a very rare malfunction), or it crashed in the sense that the exit status code was non-zero after running "rss_flow_part1.pl". That program just runs the "rss_flow_chkcf.pl" program and monitors its exit status. The exit status is non-zero if there's an "ERROR" message anywhere in the output. "WARNING" messages are okay and will result in exit status code 0.

So the flow will first take a snap-shot of all of the control files. Then it reads the results to figure out the names of all the rundirs. Then it spawns off a data prep job to build the "main.spi" and "${eckt_cellname}.vectors" file for each rundir. This occurs in parallel and is distributed to as many machines as needed using the PSub program.

If the data prep jobs crash, the output will be copied back to the "results" directory at:

$CHIP/random_spice/results/$userdir/crashed_run/part2.$rundir/

Note that the "$rundir" value comes from the control file.

The "part2.$rundir" directory indicates that the job crashed at some point or one of the commands it executed returned a non-zero exit status code. You should look at the files in this directory (if it exists) and make any changes necessary to ensure an exit status code of 0 for all commands. If you're not sure how to do this, see the flow owner, Steve Weigand.

If it didn't crash and everything ran okay, it then looks through the data prep results to determine if anything is different compared with the results of the previous run's results contained in the results directory:

$CHIP/random_spice/results/$userdir/$rundir/
- vs. -
$CHIP/random_spice/running/$userdir/$rundir/

It looks for changes of three types:

1. There was no previous rundir by this name. This is the first time it was run.

2. The circuit or timing information has changed since the last run.

3. The test-vectors have changed since the last run.

If any of the above changes occurred, it will rerun the flow for the given rundir. This means that the simulation (hsim) job will be rerun, followed by the summary command.

The order of the changes listed above is important. The first type of change takes priority over the second type of change. And the second type of change takes priority over the third. This means that a rundir that has never run before will be run before any other jobs are run. Similarly, if there's been a netlist or timing model change, then that takes priority over a rundir where the only change is the test-vectors used.

The flow is actually a cron job that runs every 30 minutes. If it detects a run already in progress, it kills it if the flow has been running more than 4 days in a row. Otherwise, it lets the original job continue to run and will simply exit. Chances are, the flow will take less than a day to complete, depending on how many jobs there are in total and how long each job takes.

When the flow is restarted, it will take another snap-shot of all of the control files and goes through the flow again.

The goal is to maximize HSIM license usage at all times. So it launches all of the data prep and summary jobs to generic queues and launches the simulation jobs to the hsimbat queue.

Speaking of the "hsimbat" queue: The hsimbat queue is marked as a PEON queue and is paired with the hsim queue which is marked as a GOD queue. This means that jobs launched manually to the hsim queue can kill jobs in the hsimbat queue if the hsimbat queue jobs are using all of the licenses. This will not affect the flow at all. The flow recognizes when this occurs and gladly steps aside to let the manual jobs run.

Where are the results?

After the flow runs your cells which you've defined in the control file, the results will be available at:

$CHIP/random_spice/results/$userdir/$rundir/

A summary file will be located at:

$CHIP/random_spice/results/$userdir/$rundir/$eckt_cellname.sum.gz

Here's an example summary file:

Summary Date : Wed Jul 16 21:53:41 2003
Simulation Date : Wed Jul 16 21:53:35 2003
Simulation Machine: fb129

	:	Now	Ever
	:	===	====
Total vectors	:	106	1166
Passing vectors	:	59	651
Failing vectors	:	47	515
Failing timing checks	:	10	110
Sim CPU time	:	0:22:40	3:58:19
Sim peak memory use (MB):		101.4	101.6

Sim transient time (s) : 2.65000e-08 2.91500e-07

Reason(s) for new run:
- Different vectors.

This run's test vector mismatches:
DOUT Error: t=0.95 ns expect=1 state=0 node=rddat6
DOUT Error: t=0.95 ns expect=1 state=0 node=rddat4
DOUT Error: t=0.95 ns expect=1 state=0 node=rddat2
...

This run's timing check violations:
checkdelay_ph1_r_qb1_r: Ref=v(ph1) Sig=v(qb1) Tref=1.9013e-08 Tsig=1.913e-08 (Tsig-Tref)=1.17e-10

checkdelay_ph1_r_qb1_r: Ref=v(ph1) Sig=v(qb1) Tref=2.6513e-08 Tsig=2.6634e-08 (Tsig-Tref)=1.21e-10
checkdelay_ph1_r_qb0_r: Ref=v(ph1) Sig=v(qb0) Tref=2.7513e-08 Tsig=2.7633e-08 (Tsig-Tref)=1.2e-10
...

The example summary file above is pretty self explanatory. The column labeled "Now" represents the current run. The column labeled "Ever" represents the current run and all of the previous runs. It's a running total of whatever is in the "logfile" file.

The "logfile" file is kept here:

$CHIP/random_spice/results/$userdir/$rundir/logfile

This log file isn't the HSIM log file. It's the flow's special log file. It details when the flow was run, why it ran, and what the summary information showed for it. Here's an example logfile:

@ Thu Jul 14 17:01:33 2003
Machine: "fb131" CPU time: 0:00:31 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/219/187
Timing Check Violations: 15
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- First time this rundir was ever run.

@ Thu Jul 14 17:41:36 2003
Machine: "fb127" CPU time: 0:00:32 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/218/188
Timing Check Violations: 10
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- Different vectors.

@ Thu Jul 14 18:21:37 2003
Machine: "fb131" CPU time: 0:00:31 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/220/186
Timing Check Violations: 0
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- Different vectors.

@ Thu Jul 14 18:58:34 2003
Machine: "fb132" CPU time: 0:00:32 Peak Memory Usage: 70.0MB
Vectors Total/Pass/Fail: 406/216/190

Timing Check Violations: 12
Transient time: 1.015e-07 (seconds)
Reason(s) for new run:
- Different vectors.

Notice that each run begins with "@ $date". This date is actually when the flow completed the simulation run, not when it started the run.

Each time a new run is performed, the previous run is archived to the "previous_violations" subdirectory if the previous run had any vector mismatches or timing check violations. You can have multiple "previous_violations" subdirectories:

$CHIP/random_spice/results/$userdir/$rundir/previous_violations/archive_1/
$CHIP/random_spice/results/$userdir/$rundir/previous_violations/archive_2/
$CHIP/random_spice/results/$userdir/$rundir/previous_violations/archive_3/
... etc.

These "previous_violations" archive directories are numbered starting at 1. Each time a new run is made, the previous failed run will get archived. The number of archived directories is limited by the limit_kept_logs control file setting which defaults to 4. If this limit is reached, no new runs will occur.

At this point, if you have any failing test-vectors or timing-check violations, then you can modify the control file and/or the circuit to fix the problems. The flow will detect any changes and will rerun.

IMPORTANT: When you change the circuit, or the SPICE technology information changes, or the timing information changes, then the "previous_violations" directory and all its archives will be deleted. The "logfile" file will also be deleted. This is just like starting over from scratch. This doesn't occur if just the test-vectors change. There must be some change in the circuit or the timing information for the logfile and previous_violations directory to be wiped clean.

You can also see your results along with the results of all other peoples' runs at:

$CHIP/random_spice/Summary/all_cells.sum
Here's an example:

DIR	DATE	VECS TOTAL	VECS PASS	VECS FAIL	TCHECK FAIL	TRANSIENT TIME (s)
l2_lru/tt_tt.setup	07/22/03 18:10	106	58	48	0	2.6500e-08
weigand/gemux3kreg_x4_20.tt_tt.hold	07/22/03 17:57	406	406	0	407	1.0150e-07
weigand/gemux3kreg_x4_20.tt_tt.setup	07/22/03 17:55	406	295	111	321	1.0150e-07

| weigand/wcmps_x5_20.tt_tt | 07/22/03 17:49 | 404 | 404 | 0 | 304 | 1.0100e-07 |

| weigand/wxregi_x4_16.tt_tt.hold | 07/22/03 17:51 | 406 | 217 | 189 | 0 | 1.0150e-07 |

| weigand/wxregi_x4_16.tt_tt.setup | 07/22/03 17:53 | 406 | 218 | 188 | 0 | 1.0150e-07 |

+----------+----------+----------+----------+----------+----------+----------+----------+

| | | | | (EVER) | (EVER) | (EVER) | (EVER) |

| | CPU | MAX MEM | | VECS | VECS | VECS | TCHECK |

| DIR | TIME | (MB) | MACHINE | TOTAL | PASS | FAIL | FAIL |

+----------+----------+----------+----------+----------+----------+----------+----------+

| l2_lru/tt_tt.setup | 0:21:11 | 100.9 | fb127 | 318 | 175 | 143 | 0 |

| weigand/gemux3kreg_x4_20.tt_tt.hold | 0:01:04 | 80.5 | fb105 | 1218 | 1218 | 0 | 1189 |

| weigand/gemux3kreg_x4_20.tt_tt.setup | 0:01:02 | 80.4 | fb105 | 1218 | 863 | 355 | 977 |

+----------+----------+----------+----------+----------+----------+----------+----------+

| weigand/wcmps_x5_20.tt_tt | 0:00:17 | 0.0 | fb106 | 1212 | 1212 | 0 | 907 |

| weigand/wxregi_x4_16.tt_tt.hold | 0:00:48 | 82.1 | fb105 | 1218 | 650 | 568 | 0 |

| weigand/wxregi_x4_16.tt_tt.setup | 0:00:47 | 82.3 | fb105 | 1218 | 655 | 563 | 10 |

+----------+----------+----------+----------+----------+----------+----------+----------+

+----------+----------+----------+----------+

| | (EVER) | (EVER) | (EVER) |

| | TRANSIENT | CPU | MAX MEM |

| DIR | TIME (s) | TIME | (MB) |

+----------+----------+----------+----------+

| l2_lru/tt_tt.setup | 7.9500e-08 | 1:03:31 | 101.2 |

| weigand/gemux3kreg_x4_20.tt_tt.hold | 3.0450e-07 | 0:02:44 | 83.8 |

| weigand/gemux3kreg_x4_20.tt_tt.setup | 3.0450e-07 | 0:02:30 | 83.8 |

+----------+----------+----------+----------+

| weigand/wcmps_x5_20.tt_tt | 3.0300e-07 | 0:00:42 | 0.0 |

| weigand/wxregi_x4_16.tt_tt.hold | 3.0450e-07 | 0:01:58 | 82.1 |

| weigand/wxregi_x4_16.tt_tt.setup | 3.0450e-07 | 0:01:54 | 82.3 |

+----------+----------+----------+----------+

It's a good idea to look at the "all_cells.sum" file to see how long everyone else is running their simulations. This way you can compare your run times versus theirs. If your run times are larger, you should decrease the number of vectors you run at any one time so that the overall run time is decreased. This way you're fair to others. Occasionally the flow owner (Steve Weigand) may suggest you do this so that the flow is fair to all users.

NOTE: The "all_cells.sum" file is updated once every 2 minutes, so it should be very up to date.

The Control File

The control file is at the heart of the flow. It's in the Perl 5 language format. It allows you to set variables which tell the flow exactly what you want it to do. You need to specify which cells you want to run, where the random vector

script file is, which SPICE corner to use, etc.

You need to install your control file in the following location:

$CHIP/random_spice/user/$userdir/control.pl

The $userdir name can be anything you want, limited to the following characters: a-z, A-Z, 0-9, "_" (underscore), and "." (dot). Most people will just use their username or the name of a module or cell. It has no significance to the flow. The control file, however, must be named "control.pl".

If you don't install a control file, the flow will not run for the given $userdir directory. If you delete a control file that used to be in use, but you don't delete the $userdir directory, then the flow ignores the $userdir directory in its current pass through the flow and keeps the corresponding "results" directory if it ran before. If, however, you delete the $userdir directory altogether, the flow will eventually delete the corresponding "results" directory. So it's okay to temporarily move the control file so that your runs won't happen and eventually move it back after you've made some changes to it.

The Basics

There's just a single control file variable (%rss) that you use to control the flow. But it has a lot of different parts. Below is its format:

1) $rss{$rundir}{$property} = $value;
2) $rss{$rundir}{$property}{$subproperty} = $value;

In the Perl language, the variable "%rss" shown above has the structure known as a multi-level associative array (in other words, a hash table). The first line above shows a two level associative array, while the second line shows a three level associative array. Both of these formats are used to specify every type of property you'll encounter in the flow.

When the flow runs, it runs in a unique directory:

$CHIP/random_spice/running/$userdir/$rundir/

The $userdir directory in the "running" directory is created automatically by the flow and is the same as the $userdir directory you created in the "user" directory. The $rundir directory is also created automatically by the flow.

The name of the $rundir directory is set using the %rss variable in the control file. It can be any name you want to assign to it, so long as it makes sense to you. But you must limit it to the characters: a-z, A-Z, 0-9, "_" (underscore), and "." (dot). For example:

```
$rss{"btac_cnt.tt_tt.hold"}{"run"} = 1;
$rss{"btac_cnt.tt_tt.hold"}{"cell"} = "btac_cnt";
$rss{"btac_cnt.tt_tt.hold"}{"spice_techfile"} = "tt_tt";
```

```
$rss{"btac_cnt.tt_tt.hold"}{"setup_or_hold"} = "hold";
$rss{"btac_cnt.tt_tt.hold"}{"vec_cmd"} =
   "$CHIP/random_spice/user/btac/btac_cnt.rvec.pl";
   ...
```

The example above shows that a run directory is named "btac_cnt.tt_tt.hold". This name makes immediate sense. The intent is to run the "btac_cnt" cell's hold-time analysis using the "tt_tt" SPICE corner in this directory. But this name is completely meaningless to the flow itself. You could have just as easily named it something vague like "run1".

Continuing on the example above, the flow would run in the following directory:

$CHIP/random_spice/running/$userdir/btac_cnt.tt_tt.hold/

And when the run completed, its results would be copied to the following directory where they can be viewed:

$CHIP/random_spice/results/$userdir/btac_cnt.tt_tt.hold/

You can create as many runs as you want inside of a given $userdir directory. Each run has a unique set of run properties which are entirely under your control. The full list of these properties is shown later.

For example:

```
$rss{"run1"}{"run"} = 1;
   ...
$rss{"run2"}{"run"} = 1;
   ...
$rss{"run3"}{"run"} = 1;
   ...
```

The example above would cause the flow to create the following three directories:

$CHIP/random_spice/running/$userdir/run1/
$CHIP/random_spice/running/$userdir/run2/
$CHIP/random_spice/running/$userdir/run3/

For each run directory, you can set many different properties which tell the flow how to run. Some properties are required for you to set them. Other properties can be left blank, and the flow will automatically define them. Which ones are required and which are optional will become clear later.

Practical Matters

- Please note that wherever a variable is used to specify a filename or a script, it must begin with "/". In other words, you need to specify an absolute path to the file or script. There may be exceptions, such as the spice_techfile property,

where you can enter a filename relative to the hspice technology directory if you want.

- Leading and trailing spaces in strings are bad. For example:
  开头结尾空白

  $rss{"run1"}{"cell"} = "   btac ";

  The example above shows an incorrect way to specify a cellname. The spaces in the name of the values of property settings may or may not be stripped out by the program. Please make sure all property settings do not have spaces in them where they don't belong.

- The current working directory is considered to be the $rundir directory for each run. So if you use a script in the flow that creates a file without specifying an absolute pathname, it will be created there:

  $CHIP/random_spice/running/$userdir/$rundir/

- The $CHIP, $CHIPBRF, $CENTHOME, and $CADHOME environment variables will be set by the flow prior to using your control file. So your control file can use these variables. To make it easier for you, I've carried over these environment variables into Perl variables. So you don't need to reference them as "$ENV{CHIP}" and such. For example:

  $rss{"myrun1"}{"vec_cmd"} = "$CHIP/random_spice/user/weigand/vec.pl";

- The $CHIP, $CHIPBRF, $CENTHOME, and $CADHOME environment variables will be set by the flow prior to running any of the user-definable scripts. There are also these additional environment variables which are set: $RSS_DIR, $RSS_RUNDIR, $RSS_CELL, $RSS_ECKTCELL, $RSS_CORNER, $RSS_SETHLD, and $RSS_CTRLDUMPFILE. The $RSS_DIR environment variable contains the name of the directory ($userdir). The $RSS_RUNDIR variable contains the name of the run directory ($rundir). The $RSS_CELL variable contains the name of the cell (the cell property). The $RSS_ECKTCELL variable contains the name of the ECKT cell (the eckt_cellname property). The $RSS_CORNER variable contains the SPICE corner (the spice_techfile property). The $RSS_SETHLD variable is set to either "setup" or "hold" (the setup_or_hold property). The $RSS_CTRLDUMPFILE variable contains the absolute path to your control dump file (a dump of the rss_flow_chkcf.pl program which runs on your control file).

  Your script can make use of all these environment variable settings to make decisions about what to do. If you need more than these, let me know.

- Also, keep in mind that the control file is applied before the default values are set for all of the different variables. This is because the default values depend on what you set in the control file. Think of it this way: You set whatever variables you want, and the flow will then look at what you did and will fill in the rest.

- There is always the possibility that you entered something incorrectly into the control file. You might have forgotten to end the line with a semicolon. You might have forgotten to enter a required property. You might have misspelled the name of a variable. The flow will check this before it runs anything, and if there is a problem, it just won't run anything in that particular directory. It would keep its current results in the "results" directory the same until you fixed the control file. The way you can learn of these problems is by looking at the all-cells summary report, and it should show up in the

corresponding error column.

If you want to check your control file for any problems before you use it, simply type this at the command line:

rss_flow_chkcf.pl $controlfile

And replace $controlfile with the name of your control file. The control file can be named anything. This way you can check it before renaming it to "control.pl", where the flow will pick it up.

Search for any "ERROR" or "WARNING" messages that occur. Error messages cause the flow to skip running the given $rundir. Warning messages are not fatal, so the flow will still run.

The rss_flow_chkcf.pl command is very powerful. Not only is it checking syntax, it's also checking for the availability of files you specify. It also goes through a full analysis, pulling in the timing model information and generates information showing you exactly what it calculates. You can consider this a dry run prior to handing it over to the flow just to see if it will work.

排练

In fact, when you're editing your control file, it's best to edit it in another place or call it "control.working.pl" instead of "control.pl". This way the flow won't try to use it even though you've not completed your revisions. You would just copy it back to "control.pl" when you were done.

- Unless otherwise stated, the units for all of the properties are: picoseconds, femptoFarads, and milliVolts.

ps fF mV

- In many cases, you can give a Perl regular expression instead of the name of a pin, for example. These regular expressions are thought of as "default" settings for a given property. So if you give a regular expression and an explicit pin name for a given property, the explicit pin name wins.

For example, look at the arrival_set property. You might do the following:

```
$rss{$rundir}{"arrival_set"}{".*"} = 300;
$rss{$rundir}{"arrival_set"}{"i0"} = 250;
```

The example above will perform two searches for pin names. The first is for the regular expression:

```
/^.*$/
```

The second is explicitly looking for pin "i0" (not using any regular expression).

This example sets all of the input pins' arrival times to 300 except for pin "i0" which is set to 250. Since "i0" is not a regular expression, it takes precedence over the ".*" setting which is a regular expression.

非正则优于正则表达式执行

You can use multiple regular expressions, but if more than one matches a given signal name, then the first one in sorted order will win. For example:

```
$rss{$rundir}{"arrival_set"}{"i[0-7]"} = 250;
$rss{$rundir}{"arrival_set"}{".*"} = 300;
```

In the example above, there are two regular expressions which both match "i0", "i1", etc. So what will "i0" be set to? Since ".*" occurs before "i[0-7]" in sorted order, it will win. So "i0" gets set to 300. Same for i1 through i7.

NOTE: Regular expressions only work if you are using a timing model. If you don't have a timing model, then there's nothing there to do a pattern match, so obviously nothing will occur.

- It's possible to embed Perl language commands into the control file instead of just defining variables. You can setup a loop, for example, to help you define the variables if you want. You can even create subroutines if you want.

But you might be wondering if this could potentially cause problems, since the names of the variables you create might conflict with variables by the same name in the scripts that use the control file. Don't worry, because it turns out that the control file is always given its own "namespace" separate from everything else. The scope of all variables in the control file is therefore entirely local to the control file itself. So create any variable names or subroutines you want, and don't worry about the possibility of having namespace collisions. (FYI, it does this by merging your data into a Perl package called "rss_control" rather than merging it into the "main" package.)

Properties

Reference doc: /n/insideweb/html/users/weigand/public_html/rss_flow.html
Or: /c0319/keithm/cnq/doc/rss_flow (P16—P57)

Timing Details Behind the Scenes

When simulating test-vectors, there are just three things which are most important for timing consideration: the arrival times of all input signals, the slews for all input signals, and the output check windows for all output signals. In HSIM's vector format, these correspond with the "delay", "rise", "fall", and "check_window" statements.

So how do these get defined in the Random SPICE Simulation Flow? If you give the flow a timing model, the flow will attempt to derive arrival times from the setup or hold time values of all inputs. If there is no setup or hold time associated with a given input pin, then the arrival time is set to "0" (zero) by default. This would occur with all combinatorial paths.

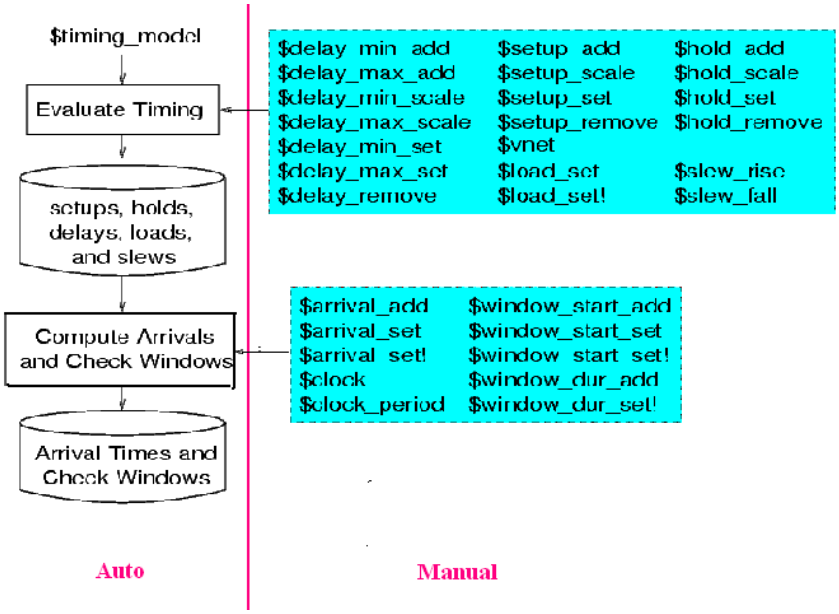
The output check windows are derived from the timing model by first deriving the arrival times for all input signals and then adding the timing model's delay values from all input pins to all output pins. The maximum absolute time that results is the time that is used for the start of the output check window. Then the duration of the window is

arbitrarily set to the difference between the start time and the end of the clock period.

If there is no timing model present, it's entirely up to you to use the control file properties to set the arrival times and output check windows. You can do this by directly setting these through the appropriate properties (arrival_set and window_start_set for example), or you can do it indirectly by defining setups, holds, and delays.

If the timing model is present but doesn't have a setup or hold time associated with a particular input pin, then again, it's up to you to set the arrival time for this input pin using the control file properties.

The whole process is summarized in the diagram below:



Notice that the diagram shows that the timing model information is used along with various control file properties to derive the final setup, hold, and delay values. You can use the timing model by itself without specifying any control file properties, or you can modify the result in whatever way by using the control file properties listed.

Once the setup, hold, and delay values are calculated, then that information flows into the part of the flow that computes the arrival times of all input pins and the output check windows for all output pins. Once again, it's possible not to use any control file properties here, but they are there if you need to modify the result.

It should be noted that the flow needs to calculate everything relative to the 50% VDD point for each signal. Then it will specify absolute times when signals begin to rise, fall, or transition. And if it sees that there are negative values for starting times, it slides all waveforms over until they are positive values. You'll see what this means in the examples later on.

Arrival Times

Arrival times are calculated by looking at the setup times if this is a setup-time run or hold times if this is a hold-time run (specified by the setup_or_hold property). Also affecting it are the input slew times, the manual properties that modify setup/hold times, the manual properties that modify slew times, and the manual properties that modify arrival times.

Let's take an example...

EXAMPLE 1:

Suppose you have the following conditions:

1. Clock is ph1 with a period of 500ps and arrives at time=0ps.
2. Slew (rising and falling) on pin a0 is 45ps (20-80%).
   Slew (rising and falling) on pin ph1 is 25ps (20-80%).
3. Setup a0 ^ wrt ph1 ^ = 40
   Setup a0 v wrt ph1 ^ = 20
   With respect to

That's what you give the flow. The rest is what happens behind the scenes by the RSS flow...

Remember that ph1 arrives at time t=0ps. This is actually the time at which ph1 is 50% VDD. So in reality, we're implying that ph1 began rising at some negative time value.

Also remember that the setup time of 40ps, for example, is the midpoint to midpoint time difference. So start with the 50% VDD point of ph1 rising, and then go back in time 40ps. That point will mark the 50% VDD point of a0 rising.

Then once we have the 50% VDD points of a0 and ph1, we can easily find their starting rise/fall and ending rise/fall time points by merely subtracting and adding half the 0-100% slew for each input.

So let's see how this works with actual numbers...

The flow first needs to convert the 20-80% slews into 0-100% slews. This is accomplished by using a simple scale factor of 1.66667:

The 0-100% slew for a0 = slew(a0) * 1.66667
= 45 * 1.66667
= 75ps

The 0-100% slew for ph1 = slew(ph1) * 1.66667
= 25 * 1.66667
= 41.67ps

The 50% VDD arrival time for pin "a0" is calculated:

Arrival of pin a0 = arrival_time_of_ph1
+ period_of_ph1
- max_setup_time_a0_wrt_ph1
= 0 + 500 - max(40,20)

= 500 - 40
= 460ps

But keep in mind that this represents the point at which the signal is at 50% VDD. That's what you and I refer to as "arrival time". But what Hsim and other SPICE simulators want is the time at which the signal begins to rise or fall. So we need to account for slews:

The 50% VDD arrival of pin ph1 is 0ps.
Therefore, ph1 begins to rise or fall at:

= 0ps - 41.67/2
= -20.835ps

The 50% VDD arrival of pin a0 is 460ps.
Therefore, a0 begins to rise or fall at:

= 460 - 75/2
= 460 - 37.5
= 422.5ps

But we're not done there just yet. Notice that ph1 is now beginning to rise at a negative time value. This is probably okay for Hsim, but it's better to start everything at time zero to be consistent. I'm not sure, but Hsim and perhaps other simulators do the DC characterization point at time zero, so it's a good idea to shift everything to start at time zero.

In this case, we only know about two pins, "a0" and "ph1". But there could be any number of pins in the design. So the flow will determine at this point which one of the pins is beginning to rise or fall at the least time value. For our case, it's ph1 which begins to rise at -20.835ps. So we'll need to add 20.835ps to all of the times...

Final arrival times:

ph1 = -20.835 + 20.835 = 0ps
a0 = 422.5 + 20.835 = 443.335ps

Note that it only slides all the values if there's a negative arrival time for any pin. If all the arrival times are positive, it doesn't slide anything.

Note also that these final arrival times are what you see when you run rss_flow_chkcf.pl. They represent the start of the signal's rise or fall, not the 50% VDD times. So you have to remember to assume every value has gone through slew adjustment and negative time value sliding. It can be confusing unless you know how it's done.

In the example above, you saw how setup times (presumably from a timing model) and input pin slews affected the arrival times of input pins. Let's see now how a hold time affects the arrival time. Remember, if this is a hold

time run, then setup times are ignored. And vice versa, if this is a setup time run, hold times are ignored.

EXAMPLE 2:

Given:
1. Clock is ph1 with a period of 500ps and arrives at time=0ps.
2. Slew on pin a0 is 75ps (20-80%).
   Slew on pin ph1 is 50ps (20-80%).
3. Hold a0 ^ wrt ph1 ^ = 100ps
   Hold a0 v wrt ph1 ^ = -20ps

Calculation...

The 50% VDD arrival of pin ph1 is 0ps (it was given above).

The 50% VDD arrival of pin a0 = arrival_time_ph1 + max(100,-20)
= 0 + 100ps
= 100ps

Now we calculate the *beginning* of the rise/fall for ph1:

= arrival_time_ph1 - (50*1.66667)/2
= 0ps - 41.67
= -41.67ps

And, a0 *begins* to rise or fall at:

= arrival_time_a0 - (75*1.66667)/2
= 100 - 62.5
= 37.5ps

(Note that 1.66667 is the conversion factor for converting
20-80% slews into 0-100% slews.)

Finally, we do have a negative valued arrival time, so we need to slide everything by 41.67ps to make the least valued arrival time to be at time 0. The final arrival times are:

ph1 = -41.67 + 41.67 = 0ps
a0 = 37.5 + 41.67 = 79.17ps

Now let's take an example of calculating the arrival time value with manual intervention by way of control file properties:

EXAMPLE 3:

Given:
1. Clock is ph1 with a period of 500ps and arrives at time=0ps.
2. Slew on pin a0 is 75ps (20-80%).
   Slew on pin ph1 is 50ps (20-80%).
3. Setup a0 ^ wrt ph1 ^ = 100ps
   Setup a0 v wrt ph1 ^ = -20ps
4. Property "setup_add" for arcs ".* ^ .* ^" = 10
   Property "setup_add" for arcs ".* v .* ^" = 25
5. Property "arrival_add" for pin a0 = -30

Calculation...

The setup times have been modified by the "setup_add" property,
so we calculate new ones at this time:
  setup a0 ^ wrt ph1 ^ = 100 + 10 = 110ps
  setup a0 v wrt ph1 ^ = -20 + 25 = 5ps

The 50% VDD arrival of pin ph1 is 0ps.

The 50% VDD arrival of pin a0 = 0 + 500 - max(110,5)
= 390ps

Now we adjust the 50% VDD arrival times by the amounts specified
with the "arrival_add" property:

  The 50% VDD arrival of pin a0 = 390 - 30 = 360ps

Now we calculate the *beginning* of the rise/fall for ph1:

= 0ps - (50*1.66667)/2
= -41.67ps

And, a0 *begins* to rise or fall at:

= 360 - (75*1.66667)/2
= 360 - 62.5
= 297.5ps

Finally, we do have a negative valued arrival time, so
we need to slide everything by 41.67ps to make the least
valued arrival time to be at time 0. The final arrival

times are:

ph1 = -41.67 + 41.67 = 0ps
a0 = 297.5 + 41.67 = 339.17ps

Output Check Windows

Output check windows are calculated by finding all delay arcs from all input pins to that output pin. The delay arc which results in the most time value is used. That represents the time at which the simulator should start comparing the simulation's value for a given output pin with the vector file's expected value. In other words, the "window start" value. It will do this for an amount of time called the "window duration". The window duration is calculated automatically by simply subtracting the clock period from the window start time. In other words, the window duration is just the remaining time in the clock cycle. Better values for it can be entered into the flow manually if desired.

The things that affect the output check window start times are the arrival times, the delay arcs, the manual properties that modify the delay arcs, and the manual properties that modify the window start times directly.

Let's take some examples:

EXAMPLE 1:

Given:
1. Clock is ph1 with a period of 500ps and arrives at time=0ps.
2. Slew on pin a0 is 45ps (20-80%).
   Slew on pin a1 is 45ps (20-80%).
   Slew on pin ph1 is 25ps (20-80%).
3. Setup a0 ^ wrt ph1 ^ = 340
   Setup a0 v wrt ph1 ^ = 320
   Setup a1 ^ wrt ph1 ^ = 360
   Setup a1 v wrt ph1 ^ = 330
4. Delay a0 ^ -> out0 ^ = min 125, max 150
   Delay a0 v -> out0 v = min 75, max 100
   Delay a1 ^ -> out0 ^ = min 105, max 135
   Delay a1 v -> out0 v = min 70, max 95
   Delay ph1 ^ -> out0 ^ = min 100, max 110
   Delay ph1 ^ -> out0 v = min 110, max 120

Calculation...

The 50% VDD arrival of pin ph1 is 0ps (given above).

The 50% VDD arrival of pin a0 = 0 + 500 - max(340,320)
= 500 - 340
= 160ps

The 50% VDD arrival of pin a1 = 0 + 500 - max(360,330)
= 500 - 360
= 140ps

Now we can look at the output check window start time. For this, we note the delays. We find the maximum valued delay arc from all inputs to each output. Pin out0 has 6 delay arcs to it. So we need to calculate each one separately:

A. Delay a0 ^ -> out0 ^ = min 125, max 150

The 50% arrival time of pin a0 is at 160ps. So the maximum output check time would be: 160 + max(125,150)
= 160 + 150
= 310

B. Delay a0 v -> out0 v = min 75, max 100

The 50% arrival time of pin a0 is at 160ps. So the maximum output check time would be: 160 + max(75,100)
= 160 + 100
= 260

C. Delay a1 ^ -> out0 ^ = min 105, max 135

The 50% arrival time of pin a1 is at 140ps. So the maximum output check time would be: 140 + max(105,135)
= 140 + 135
= 275

D. Delay a1 v -> out0 v = min 70, max 95

The 50% arrival time of pin a1 is at 140ps. So the maximum output check time would be: 140 + max(70,95)
= 140 + 95
= 235

E. Delay ph1 ^ -> out0 ^ = min 100, max 110

The 50% arrival time of pin ph1 is at 0ps. So the maximum output check time would be: 0 + max(100,110)
= 110

F. Delay ph1 ^ -> out0 v = min 110, max 120

The 50% arrival time of pin ph1 is at 0ps. So the maximum output check time would be: 0 + max(110,120)
= 120

The maximum overall time is:

max(310,260,275,235,110,120) = 310ps

So the output check window starts at time 310ps. That's the point in time when we're sure that out0 has a valid output value.

What is the output check window's duration? In this case, we note that the clock period is 500ps. So we subtract 310ps from 500. That gives a window duration of 190ps.

Now we calculate the beginning of ph1 rise or fall:

= 0ps - (25*1.66667)/2
= -20.83

And a0 begins to rise or fall at:

= 160 - (45*1.66667)/2
= 160 - 37.5
= 122.5

And a1 begins to rise or fall at:

= 140 - (45*1.66667)/2
= 140 - 37.5
= 102.5

But remember, we don't like negative arrival times. So we slide all the values by 20.83ps...

Final arrival times after sliding all signals by 20.83ps:

ph1 = -20.83 + 20.83 = 0ps
a0 = 122.5 + 20.83 = 143.33ps
a1 = 102.5 + 20.83 = 123.33ps

This means we also need to slide the window start time:

310 + 20.83 = 330.83

Final output check window start time is 330.83ps.

And the window duration stays at 190ps.

So what will be reported by rss_flow_chkcf.pl is:

Pin	Start Time	Duration
out0	330.83ps	190ps

And we're done. Whew! Luckily, the flow does all of this for you.

EXAMPLE 2:

Given:
1. Clock is ph1 with a period of 500ps and arrives at time=0ps.
2. Slew on pin a0 is 45ps (20-80%).
   Slew on pin ph1 is 25ps (20-80%).
3. Property "arrival_set" for a0 = 450
4. Delay ph1 ^ -> out0 ^ = min 100, max 110
   Delay ph1 v -> out0 v = min 110, max 120
   Delay ph1 ^ -> out0 v = min 90, max 100
   Delay ph1 v -> out0 ^ = min 80, max 90
   Delay a0 ^ -> out1 v = min 200, max 250
   Delay a0 v -> out1 ^ = min 220, max 260
5. Property "window_dur_set!" on out1 = 150;

Calculation...

The 50% VDD arrival of pin ph1 is 0ps.

The 50% VDD arrival of pin a0 = 450 (set by the user)

Now we calculate the beginning of ph1 rise or fall:

= 0ps - (25*1.66667)/2
= -20.83ps

And a0 begins to rise or fall at:

= 450 - (45*1.66667)/2
= 450 - 37.5
= 412.5ps

Final arrival times after sliding all signals by 20.83ps:

ph1 = -20.83 + 20.83 = 0ps
a0 = 412.5 + 20.83 = 433.33ps

Now we can look at the output check window start time:

For out0 ...
Delay ph1 ^ -> out0 ^ = min 100, max 110
Delay ph1 v -> out0 v = min 110, max 120
Delay ph1 ^ -> out0 v = min 90, max 100
Delay ph1 v -> out0 ^ = min 80, max 90

Since ph1 has a 50% VDD arrival time at t=0, then
the max value is just max(100,110,110,120,90,100,80,90) = 120.

So the output check window starts at time 120 for out0.

And we still have to slide it by 20.83ps due to the negative
arrival time of ph1. That makes it: 120 + 20.83 = 140.83ps.

And the window duration for out0 is just the clock period
minus the value before sliding: 500 - 120 = 380ps

For out1 ...
Delay a0 ^ -> out1 v = min 200, max 250
Delay a0 v -> out1 ^ = min 220, max 260

The 50% VDD arrival time of a0 was set by the user to
be 450ps. So this gives two delay arcs:

Arc 1: 450 + max(200,250) = 700ps
Arc 2: 450 + max(220,260) = 710ps

And so the maximum value of those two is chosen to be
max(700,710) = 710. This is the window start time. But
remember we need slide it by 20.83ps due to the negative
arrival time of ph1. That gives the final window start time
as 710 + 20.83 = 730.83ps.

The user has a "window_dur_set!" property for out1, and it's set to 150. So that will be the window duration time for out1.

But what if the user didn't set a window duration and instead let the flow calculate one? In this case, it would calculate the window duration by subtracting 710 from the clock period:

500 - 710 = -210

But because it's a negative valued number, this will cause the rss_flow_chkcf.pl program to generate an error message. And so the flow would fail. At that point, you'd be required to use the "window_dur_set!" property.

And so that completely describes the timing for this example.

So what will be reported by rss_flow_chkcf.pl is:

Pin	Start Time	Duration
out0	140.83ps	380ps
out1	730.83ps	150ps

Reference doc: /n/insideweb/html/users/weigand/public_html/rss_flow.html
Or: /c0319/keithm/cnq/doc/rss_flow (P57—P64)

FAQ

Q: Should I just give it a timing model and assume that it's going to be able to successfully derive the arrival times and output check windows from it?

A: No. The timing model could have combinatorial paths that don't have setup times or hold times associated with them. This means the flow assumes the arrival times for those input signals will be at time 0 (zero) relative to the clock period. This may not be an accurate assessment of the arrival time. It's possible in this case to have a false positive result.

It's also possible that your timing model has half-cycle paths or maybe other conditions that the program hasn't anticipated. In this case, it's not going to calculate the values properly.

The flow is there to help you, but it's not infallible.

Ultimately, it's up to you to use the rss_flow_chkcf.pl program to see what the flow calculates for all of the arrival times and output check windows. If what you see doesn't look right, then you need to modify their values using

control file properties.

Q: When do you think you'll have the capability to derive higher level timings for modules that have embedded timing models in lower level blocks? Also, what should we do to get around this problem in the meanwhile?

A: This is a tricky problem. Unfortunately, it affects the majority of modules in our design.

There are a few ways this can be handled, and I'm considering them all. Probably the best way to handle this is to use Pearl to analyze the main module which has embedded timing models for lower level blocks. We'd use Pearl to evaluate the timing at the main module level. This would give us a timing model for it. Whether this part of the flow would be built-in or whether it would be something stand-alone that you could use to create a temporary timing model, I don't know just yet. I'm a little hesitant on doing this because this will likely open a can of worms. Now the flow would need to handle setting up and running Pearl, which could be problematic. I don't know the issues in doing this just yet.

A second way of doing this would be to use a SPICE simulator to detect the eph1 to ph1 delay. Then once that's known, the embedded submodule's timing model can be extrapolated to the main module level in a straight forward way. It would be a two-pass flow, running HSIM first to get the eph1 to ph1 delay, and a second time to do the actual test-vector checking. The "ph1" clock signal in the submodule's timing model would be renamed "eph1" at the top level. The setup and hold times would be skewed to adjust for the eph1 to ph1 clock insertion delay. But the problem with this is that the main module might include some additional logic instead of just the clock regen circuit. That would mean it would be incorrect to simply skew the lower-level timings by the clock insertion delay value. It might be a rare thing that this happens, however, so maybe it would be okay to do.

A third way of doing this would be to use the timing models of the clock regens as well as any combinatorial logic. This approach would derive the timing through the clock regen as well as any other logic blocks that happen to be in the main module. This is basically like reinventing the wheel. I'd be making my own timing analyzer. It would be simpler than Pearl, but the problem is that there may be stuff that doesn't have timing models and requires PCD files in Pearl to time correctly. So we're back to using Pearl if we want to handle the general case. But this may be okay if it's rare that we'd need a PCD file. If we're just dealing with a clock regen circuit, it may be a simple matter to grab the timing model for it and evaluate the eph1 to ph1 clock insertion delay. So this is still a possibility.

First, before I start to program all of this into the flow, I want us to get familiar with the tool as it is now. The tool is at least as good as what you've been doing (manually), although I hope you'll agree that it has a lot of features that makes it compelling to use. Later on we can decide what additional things can be done to help us.

In the meanwhile, I suggest doing the following to get around the problem. First, copy the existing lower-level timing model to your RSS flow userdir. Rename the timing model to the main module's name. Edit the timing model and change the cell name to the main module's name. Change the clock signal name from ph1 to eph1. Save the timing model. In your control file, point to this timing model by using the timing_model property. Then use the arrival_add property to skew the arrival times by the eph1 to ph1 insertion delay value. (You would need to calculate this value outside of the flow somehow, either by simulation or by using the clock regen's timing model.) You can also use the setup_add and hold_add properties to accomplish the same thing. Then you're ready to run the flow.

Q: What's the deal with half-cycle paths? Why can't it figure out how to deal with them?

A: Half-cycle paths cause problems for the flow in a couple ways. First, it doesn't understand the timing

properly. So you might get an extra clock cycle added to the calculated setup or hold time value.

Second, half-cycle paths can cause difficulties when you're trying to create your random vector command file (vec_cmd). This is because when you run the verilog simulator, it doesn't know anything about these paths. So it generates output vectors that it thinks are stable for the entire cycle when in fact the signal could only be stable for half of the cycle. So the output vectors may need to be replaced with "X" (don't care) marks.

At present there is no solution for this built into the flow. The way around the first problem would be to use the control file properties to set the arrival times and output windows yourself rather than let the flow figure them out.

The way around the second issue is to modify the tv2sim_cmd property so that you can run an additional script to post-process the vector file and set "X" (don't care) where necessary. Or I can add a command line option to the existing command file to allow for this, but I'm still not sure of the issues involved in doing this. So for now it's up to you. I am going to look into the issue.

Q: Can I add a property or a command to the flow to do something I need to do?
A: Sure thing. The flow is easily expandable. Tell me what you need, and I'll see if I can get it added.

Q: The flow is crashing for my cells, and I don't know why.
A: There are a number of reasons why the flow can "crash". A crash could mean that there was a network filesystem glitch at some point. Or it could mean that there was a non-zero exit status code after running one of the many scripts that the flow runs.

To pinpoint the cause, look at your "crashed_run" directory:

$CHIP/random_spice/results/$userdir/crashed_run/

There should be at least one directory in there. The name of the directory will begin with "part1", "part2", "part3", or "part4". This indicates which part in the flow it crashed at. Part 1 is just using the rss_flow_chkcf.pl program to dump out your control file values. If you're crashing in part 1, then it's telling you that something is wrong with your control file. If you're crashing at part 2, then it's telling you that there's something wrong when it tries to do the data prep stuff to create the test-vector file and such. If it crashed at part 3, it's telling you the HSIM simulation command had a problem. And if it crashed at part 4, it's telling you that the summary command and post-processing scripts had a problem.

You can generally see the error messages by looking through your trace files:

$CHIP/random_spice/results/$userdir/crashed_run/part*/*.trace.txt

If you need help deciphering these files or you think it's something outside of your control, then see me for help.

Q: Can I get my stuff to run immediately without waiting for everyone else's jobs to run?
A: It's shouldn't be long before the flow finishes running on the other cells. But in some cases, you need results now, and you're not willing to wait for them.

See the manual flow. The manual flow lets you do what you want.

Q: Do I have to use this flow?

A: Generally, the RSS flow is required on most multibit datapath and custom cells. But if you have your own way of doing the equivalent, you can use that instead.

There are a few reasons you might want to start using this flow. First, it runs stuff for you, maintaining a running count of job statistics. At the end of the project, you can simply show the flow's log file to prove that your circuit has a sufficient amount of test-vector coverage and has no violations. And by running stuff for you, it means your circuits will run more often than if you were doing it manually, which means greater vector coverage for your circuit.

Second, it can help you out with tedious timing model evaluation for determining things like arrival times and output check windows. You can also use it to add loads or vary your input slews real easily. It even allows you to use Perl regular expressions to save you from having to do much data entry.

And by using the flow, it reduces the possibility of having human error.

Also, whenever capabilities are added to the flow in the future, you'll be in a position to take advantage of them.

Q: What if I want to reapply a set of vectors that used to fail?

A: When you have a failing set of vectors, you usually make a change either to the timing of the signals or to the circuit netlist itself to correct it. But rather than letting the flow create a new set of random vectors to run on it, you would maybe like to give it the same set of vectors that caused it to fail in the previous run. So how do you do this?

The way you do it is simple. Recall that your random vector command file (vec_cmd) generates all of the random input vectors. So what you would want to do is to modify that script to reproduce the same set of vectors as the previous run. So what you'd do is copy the "$eckt_cellname.tv" file to a safe location (maybe in your userdir). Then simply cat the file and exit with exit status 0.

For example:

```
$CHIP = $ENV{"CHIP"};
system("/bin/cat $CHIP/random_spice/user/weigand/mycell.tv");
exit(0);
```

Remember that the flow won't rerun unless something has changed. So since you're not changing the test vector set, something else must change in order for it to rerun. So you'd need to modify the timing or the circuit netlist, and then it would rerun.

When you're satisfied that your circuit is working with this set of vectors, you can simply delete those 3 lines from the top of your random vector script. Then it would work like it would normally work, creating new random vectors each time it runs instead of reproducing that same set of vectors.

Q: How do I debug a circuit once I know it has violations?

A: You could use the manual flow combined with the FAQ suggestion above to go about all your debugging

if you wanted to. But sometimes it's easier just to copy the RSS flow results directory over to a private workarea you've setup. Then you would gunzip all of the ".gz" compressed files. At that point you can modify the "main.spi" file, the circuit netlist, etc. Then when you're ready to rerun, type:

```
hsim -i main.spi
```

This is probably the easiest way to debug.

Q: How do I and why would I use the clkgen_delay_chk.pl subroutines?

A: We have a special subroutine that people use in their control files to tell the flow that it should abort mid-simulation if it ever detects a clock regen delay outside of some tolerance range. This is a good idea, because all of your timings are affected by the regen delay. You generally have to assume the regen delay you simulate is the delay you expect to get. But this is not always the case. Sometimes the simulation shows a regen delay far outside the normal tolerance range. Why does that happen? It can happen if your regen device is not sized properly, for example if you have too much load on the clock tree.

So what you would do is this. At the top of your control file, insert the following Perl code:

```
require "$CHIP/random_spice/user/clkgen_delay_chk.pl";
```

Now, within your main loops in your control file, you can call the following subroutines as follows:

```
&SetCheckForClkgenDelayRise($rundir,"eph1","ph1",$regen_delay_rise);
&SetCheckForClkgenDelayFall($rundir,"eph1","ph1",$regen_delay_fall);
```

The first argument is the name of the rundir. The second is the name of the clock that comes before the regen. The third is the name of the clock that comes after the regen. And the fourth argument is the center point of the expected regen delay (rise or fall delay).

That defines the center point delay value, and the function assumes a default tolerance of +/- 4%. During the simulation, if Hsim measures a regen delay greater than or less than the 4% margin, it aborts the simulation without letting it complete. That saves valuable simulation time, since it will likely occur at the very beginning of the simulation.

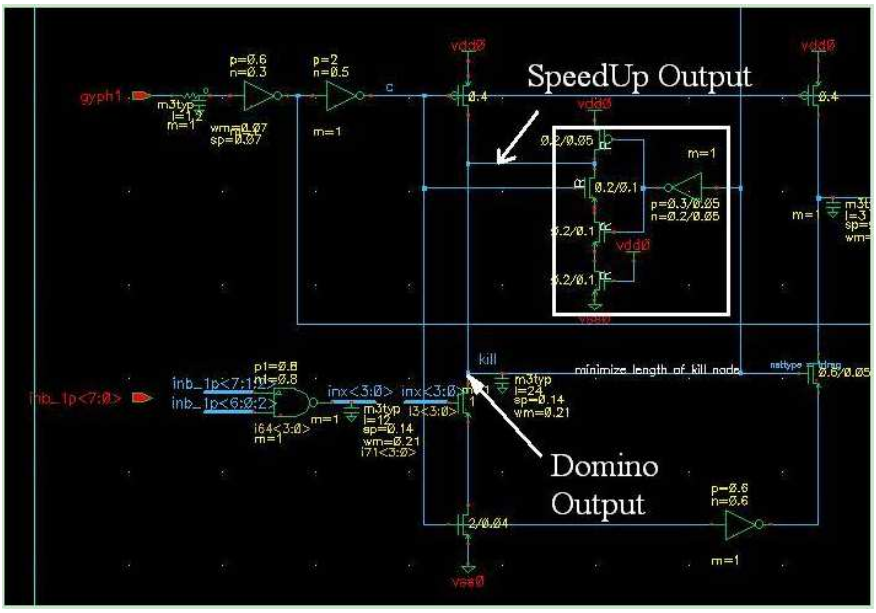
If you want more or less than the default 4% margin, you can specify your own percentage to use:

```
&SetCheckForClkgenDelayRise($rundir,"eph1","ph1",$regen_delay_rise,6);
&SetCheckForClkgenDelayFall($rundir,"eph1","ph1",$regen_delay_fall,6);
```

In the example above, it's the same as the previous example except that this one specifies an additional argument. This fifth argument is to specify a default percent margin. In this case, it's +/- 6%. You can specify whatever margin you want. Be careful when using this, however, since the default 4% margin is pretty reasonable. Increasing the margin generally defeats the purpose of the check. You're better off just commenting it out of your control file if your goal is simply to ignore any regen delay that doesn't match your expected delay.

CKT level verilog simulation has some issue with ncverilog when simulation positive feedback domino logic, the reason can be explain as below:

Signal “kill” is driven by two driving source “Domino Output” and “SpeedUp Output” (the logic in the white line range is the “SpeedUp Logic”, it can be regarded as a buffer when clock “c” is high). When clock “c” is high and some bits in inx<3:0> are one, the domino output “kill” should be driven to 0 and the speedup logic will also drive “kill” to 0 to accelerate “kill” transition. NCverilog is a simulation tool driven by events. When “kill” transit to 0, it will regard “domino output” transition to 0 is earlier than “speedup logic” transition to 0 (if it will happen) although the speedup logic delay is zero. So when it handle “domino output transition to 0” event, NC will think the “speedup output” is 1 at that time. Because both driving sources have same driving strength, no one can win the competition, the NC will treat “kill” is UNKNOWN at that time. UNKNOWN will make the “speedup logic” output unknown from that time, so “kill” is unknown at clock “c” high time. The solution method is cut off the connection with feedback logic.



Control Properties

Property Name	Description
arrival_add	
arrival_set	
arrival_set!	

The arrival time of an input signal (data or clock) is the time at which the signal first transitions. In other words, when it arrives. This is the mid-point (50% VDD) transition time, not the point at which it begins to rise or fall, by the way.

The arrival time for a given pin can be set either manually or automatically. Normally it's calculated by the flow automatically by looking at that pin's setup and hold times. If this is a setup time run (as specified by the setup_or_hold property), then it uses the setup times, otherwise it uses the hold times.

For setup time runs, it calculates the arrival time for a given input pin by getting the maximum setup time on that pin and subtracting that from the clock period. For example if this is a setup time run and the input signal needs to be setup 50ps before the clock rising edge, then its arrival time is actually set to the period of the clock minus 50ps. If the clock period was 500ps, then the arrival time would be at 450ps.

If you had two setup times on a given input pin referenced to different reference pins, then it uses the maximum of the two. So for example if a pin had a setup time of 100ps and another one of 200ps, it would use the 200ps value as the setup time.

For hold time runs, the arrival time for a given input pin is calculated by getting the maximum hold time on that pin and using that as the arrival time (no clock period subtraction). For example, if you have a hold time of 100ps on a given input pin, then its arrival time is set to 100ps.

If you had two hold times on a given input pin referenced to different reference pins, then it uses the maximum of the two. So for example if a pin had a hold time of 100ps and another one of 200ps, it would use the 200ps value as the hold time.

The flow knows the difference between a ph1, ph1b, ph2, and ph2b clock signal. It's able to determine its phase using a variety of methods (see the clock property). Knowing the phase of the clock allows the flow to pick the appropriate reference edges with setup/hold times.

NOTE: Positive setup time values run in the negative time direction, whereas positive hold time values run in the positive time direction. This is true in PTM models as well as TLF models:

Not all input signals have setup and hold times associated with them. Some can be completely combinatorial, or you may not have a timing model at all. So those input pins will have an arrival time of 0. For these input pins, you can use the arrival_set or arrival_set! property to specify their arrival time values. Or if you wanted to add margin to the

analysis, you could do that with the arrival_add property.

The arrival_add property will add (or subtract) time from the calculated arrival time value for a given input pin.

The arrival_set property will set the default arrival time value for a given input pin. If the flow can't calculate the arrival time (due to a lack of setup/hold time values for it), then it will apply this property. But if it can calculate the arrival time, then it disregards this property. This property is only used to set a default value if it can't calculate the arrival time.

The arrival_set! property will set the arrival time value for a given input pin. It overrides both the arrival_add and arrival_set properties as well as the internal calculation of the arrival time. So, even if the flow could calculate the arrival time from the setup/hold times, it ignores them and applies the arrival_set! property instead.

NOTE: These are all interpreted as mid-point (50% VDD) times. The relationship between all arrival times is preserved regardless of differing slew values for each signal.

NOTE: The use of Perl regular expressions is allowed. But if you use one that matches the name of a clock, it ignores the clock pin. This is because the clock pins should arrive at time 0, since they provide a reference for all other input signals. However, if you don't use any "magic" characters (such as "*", "[ ]", ".", etc.), then it considers your property to be a literal property, not a regular expression, and will honor it. So you can specifically apply an arrival time to a clock signal if you want, but not during a regular expression.

For example:

```
$rss{$rundir}{"arrival_set"}{"*"} = 300;
$rss{$rundir}{"arrival_set!"}{"i0"} = 250;
$rss{$rundir}{"arrival_add"}{"i1"} = 20;
```

The example above sets the default arrival time to 300ps for all input signals (except the clocks). It sets the arrival time to 250ps for the "i0" input signal, and this overrides the default value of 300ps and the calculated arrival time (if present). And 20ps have been added to the arrival time for input pin "i1".

But if you don't have a timing model, and if you've got a clock that's not named like a clock, then you will want to do something like this instead:

```
$rss{$rundir}{"arrival_set"}{"*"} = 300;
$rss{$rundir}{"arrival_set"}{"my_clock"} = 0;
```

The example above just shows how you could set a specific clock's default arrival time value to 0, yet all other inputs get an arrival time value set to 300ps.

If you want to specify certain signals, you can do something like the following:

```
$rss{$rundir}{"arrival_set"}{"d"} = 125;
```

```
$rss{$rundir}{"arrival_set"}{"en"} = 50;
```

The examples above show default arrival time settings for the "d" and "en" input signals. You could also use Perl regular expressions to match input signal names:

```
$rss{$rundir}{"arrival_set"}{"a[0-9]+"} = 125;
$rss{$rundir}{"arrival_set"}{"brt.*"} = 50;
```

NOTE: You can also use the setup/hold time properties (such as setup_add) to affect the calculated arrival time if you don't want to use the arrival time properties directly. That's up to you.

arrival_skew_limit

This property sets the maximum difference between the min and max arrival times for a given pin. This difference is typically the result of different rise and fall data edge setup or hold constraints, which means there's a good probability that you are simply not checking the minimum setup or hold value (this is a fairly dangerous hole in the RSS flow).

Assume this is a hold run. Suppose you have the following in your timing model file:

```
hold i0 ^ wrt ph1 ^ = 40
hold i0 v wrt ph1 ^ = 10
```

What will the RSS flow do with this? The setup and hold constraints are used to determine the arrival times for all input pins. The arrival times are set so that they just barely meet the setup or hold constraint.

The RSS flow can only select one arrival time. It can't set one arrival time for the rising edge and a separate arrival time for the falling edge of an input pin. Why? Because hsim doesn't allow it. Arrival times are just the time skew between inputs, that's all.

In the example above, the 40ps hold time is selected for the arrival time for pin "i0" instead of the 10ps hold time, because 40ps would satisfy both. If it selected 10ps instead, it would not satisfy the 40ps constraint.

But you can see there's a problem here. In the example above, the 10ps falling constraint will not get checked. What if the actual hold constraint on the falling edge of i0 is 20ps? You'll never know it, because it's still less than 40ps. You won't see the need to change anything in your timing model, because the RSS flow isn't actually checking this. Then when the timing model is used in Primetime, it will use the 10ps constraint. If Primetime then calculated a hold time of 15ps on the falling edge, it would say everything is fine, because it's meeting the 10ps constraint. But this is wrong, because the real constraint is 20ps.

So this is a potentially dangerous hole in the RSS flow.

The arrival_skew_limit property is actually set by default to some value for each chip. It's just the maximum allowable difference between the min and max arrival time value for a given pin. So in the example above, there are two

hold times for pin "i0", 40ps and 10ps. The difference between them is 30ps. This is the "arrival skew". If the arrival_skew_limit property is set to anything less than 30ps, it will trigger a fatal error in Part 1 of the flow (and in rss_flow_chkcf.pl).

Usage:

```
$rss{$rundir}{"arrival_skew_limit"}{$inpin} = $value;
```

Units are in picoseconds.

Where $inpin is an input or inout pin. It can also be a regular expression which matches on all input and inout pins except clock pins.

Example:

```
$rss{$rundir}{"arrival_skew_limit"}{"i0"} = 20;
```

The example above sets the arrival_skew_limit for pin "i0" to 20ps. If the difference between the min and max arrival times for this pin exceed 20ps, it will trigger a fatal error.

Shouldn't it be set to zero? The answer is yes and no. This problem is very likely to exist, but the arrival skew will be small enough in most cases that the setup and hold margins in Primetime timing analysis will cover it. If there's a large arrival skew, we want to fix those. That's why there's a default, non-zero valued arrival_skew_limit for each chip (see the defaults.pm file in the templates directory for the flow).

You can set the arrival_skew_limit to 0 if you want. This will cause it to error out whenever there's any difference between the min and max arrival times. This is a safe thing to do, but you may find that it reports a lot of things. Example:

```
$rss{$rundir}{"arrival_skew_limit"}{"*"} = 0;
```

NOTE: In the future, this hole in the flow will be closed possibly by using piecewise-linear waveforms instead of vector files. Doing so will allow more custom waveform shaping to allow both the rising and falling constraints to be checked at the same time. Until then, the only solution is to make the rising and falling data constraints the same exact values. The synLibModify.pl script is currently in development and will soon allow us to automate this task.

cell Specify the name of the cell you want to analyze. This should be the same cell as the timing model you're using, even if the ECKT and behavioral verilog model are for higher levels which instantiate this cell. If you are not using a timing model, then this cell should be the same name as your ECKT and behavioral verilog model.

clock This property can be used to specify the names of clock pins and their "sense".

The flow would normally be able to calculate this on its own using a number of methods. First, it attempts to gather a list of clock signals without worrying about the "sense" of the signal. It does this by doing the following:

o A regular expression based on the name of the signal is used to detect if it's a clock signal. This expression comes from the $REGEX_ALL_CLK_NETS variable set in the $CHIP/bin/Defaults.csh script. At the time of this writing, it was set to:

/(ph[12]|bclk|clkdiv|[ai]picclk|^itck$)/i

o From the timing model's setup and hold statements, all of the reference signals are flagged as clock pins. This may not be correct for some situations. I'm debating whether to remove this step.

o From the TLF model, there is a CLOCK_PIN statement in TLF version 4 and a PINTYPE statement in TLF version 3. Both indicate clock pins. So the flow looks for this and adds any additional pins to the list of clocks. If a pin lacks this property in the TLF, yet one of the above methods already determined it to be a clock pin, then it keeps it as a clock pin rather than removing it from the list of clock pins. There is no corresponding statement in PTM models.

o From the TLF model, there are REGISTER, LATCH, TEST_REGISTER, and TEST_LATCH statements. Each of these contain specific entries for clock names. These are actually clock "expressions" in TLF format, so they can be complex; I only take simple expressions into consideration, so they must be the name of a pin or a negated pin name ("ph1" or "!ph1b" for example); anything more complex than that will be ignored. There are no similar statements in PTM models.

The sense of the clock is thought to be positive by default. However, the flow will apply the following methods to determine if it's actually a negative edge:

o A Perl regular expression based on the name of the signal is used. The regular expression is set to:

/ph1b\b|ph2\b|\bclkb\b/i

o If this is a TLF version 3 model, it looks to see if the PINTYPE statement specifies a "CLOCK NEGEDGE" or "CLOCK LOW" value. If so, it's added to the list of inverted clocks.

Generally, it should pick up all of the clocks and determine their senses automatically. However there could be times when you want to override it and specify your own clocks. You can do that with this property.

The syntax of this property is:

$rss{$rundir}{"clock"}{$clockname} = 0, 1, or 2;
0 = Not a clock.
1 = Positive edge / level clock.
2 = Negative edge / level clock.

The $clockname variable can be the name of the clock pin or a Perl regular expression. If it's a Perl regular expression, it takes a lower precedence than if you used a specific clock name. This allows you to setup default values.

Setting the clock property to a value of 0 causes it to disregard a clock if it has automatically detected a clock. The flow will simply not consider it a clock pin if you do this.

Setting the clock property to a value of 1 forces the flow to tag the pin as a positive edge or positive level clock signal. Setting it to 2 forces it to tag it as a negative edge or negative level clock signal.

For example:

```
$rss{$rundir}{"clock"}{"*"} = 0;
$rss{$rundir}{"clock"}{"ph1"} = 1;
$rss{$rundir}{"clock"}{"ph1b"} = 2;
```

The example above will remove the clock tag on all input pins that the flow determined were clocks automatically. Then it sets the "ph1" pin as a positive edged clock and the "ph1b" pin as a negative edged clock. This is the style you'll want to use if you don't trust the way the flow automatically detects clocks. It allows you to specify the clock names (and their senses) explicitly, rather than letting the flow detect them.

Here's another example:

```
$rss{$rundir}{"clock"}{"wrtph.*"} = 0;
```

In the example above, you're deleting the clock property for all of the signals which match the regular expression "/^wrtph.*$/". Presumably these signals aren't really clocks, but the flow thinks they are.

Why is it important that the flow knows the names and senses of the clocks?

The clock pins and their senses are needed in order to calculate the arrival times properly (see arrival_set). It considers the positive edge of the clock to occur at time 0 if the sense of the clock is positive. If the sense of the clock is negative, it considers the negative edge of the clock to occur at time 0. This information is needed to accurately interpret the setup and hold times with respect to these clocks. And that information then goes into the function that calculates the arrival times of all inputs.

clock_period Specify the clock period in picoseconds. By default it's set to whatever the default is for this chip's "typical" SPICE corner. However, the default value may also have a dependency on the SPICE corner that you specify with the spice_techfile property.

delay_check Set this to 1 if you want to check that the simulation delay arcs match the timing model delays. Set it to 0 if you don't care about delays. By default, it's set to 1, but it might have a dependency on the SPICE corner.

delay_check_xa_bug If you are using the XA simulator, there is a bug in it whereby if the number of timing checks is greater than 100, it crashes with a segmentation violation. So the rss_flow_chkcf.pl program will check for this and errors out in this case.

Set this property to "1" to tell the flow that you understand the problem. It will turn off the error message and will just comment out all of the timing check statements after the 100th one.

In the future, this bug will be fixed by Synopsys. At that point, this property will be defunct.

Example:

```
$rss{$rundir}{"delay_check_xa_bug"} = 1;
```

NOTE: Alternatively, you can set the delay_check property to 0 to avoid delay checks altogether. You can also use the tcheck_remove property to selectively remove the timing checks until you get to 100 or less. You shouldn't try to use the delay_remove property, however, because removing delay arcs prevents output check windows from being calculated correctly.

delay_min_add
delay_max_add
delay_min_scale
delay_max_scale
delay_min_set
delay_max_set
delay_remove

These properties act on the delay arcs in the timing model. You can scale them, add to their values, explicitly set their values, create new delay arcs, or remove delay arcs. Since each delay arc has a minimum and a maximum delay value associated with it, you'll be able to treat them individually with these properties. Keep in mind that the flow will ensure that the minimum delay value really is the minimum delay value if you accidentally modified it to be greater than the maximum delay value (ie, it swaps the values in that case).

delay_min_add and delay_max_add: Add or subtract time from the min and max delay values seen in the timing model. This might be used to add extra margin in the simulation results. It does not apply to the delay arcs explicitly set by using the delay_min_set and delay_max_set properties.

delay_min_scale and delay_max_scale: Multiply the min or max delay value by this amount (scaling it). This occurs before delay_min_add and delay_max_add are applied. And it does not apply to the delay arcs explicitly set by using the delay_min_set and delay_max_set properties.

delay_min_set and delay_max_set: Explicitly set the delay value. This overrides the delay_min_add, delay_max_add, delay_min_scale, and delay_max_scale properties. If you specify a delay arc that isn't in the timing model, it creates one and sets it to the value you specify.

delay_remove: Removes a delay arc entirely. This has priority over all the other properties except delay_min_set and delay_max_set.

If you're confused about which properties take precedence over which other properties, this may help. The flow will execute these properties in the following order: delay_remove, delay_min_scale, delay_max_scale, delay_min_add, delay_max_add, delay_min_set, and lastly delay_max_set.

All of these properties require you to specify the input pin followed by its edge and then the output pin and its edge. The edges can be either "^" for rising, "v" for falling, or "*" for both rising and falling. Each of these four items must be separated by a single space character. And each of these four items may contain regular expression characters. If you use a regular expression character, the delay arc must already exist in the timing model for it to match. And all delay values are in units of picoseconds.

Here's an example:

```
$rss{$rundir}{"delay_min_add"}{"*"} = 50;
$rss{$rundir}{"delay_max_add"}{"*"} = 75;
$rss{$rundir}{"delay_min_set"}{"i0 ^ out1 v"} = 200;
$rss{$rundir}{"delay_max_set"}{"i0 ^ out1 v"} = 250;
$rss{$rundir}{"delay_min_set"}{"i1 * out2 *"} = 180;
$rss{$rundir}{"delay_max_set"}{"i1 * out2 *"} = 210;
$rss{$rundir}{"delay_min_scale"}{"i[0-7] * out[1-8] *"} = 0.5;
$rss{$rundir}{"delay_max_scale"}{"i[0-7] * out[1-8] *"} = 0.5;
```

The example above first scales delay arcs from i0, i1, i2, etc. to out1, out2, out3, etc. by one half their original value set in the timing model. Then it adds 50ps to all min delay arcs and 75ps to all max delay arcs. Then it explicitly sets the delay value of the i0 rising to out1 falling arc and the arcs from i1 rising or falling to out2 rising or falling. Remember, the "set" properties take priority over the "add" and "scale" properties (or you can consider "set" properties as coming last in the order of execution).

The delay_remove property is special. It occurs first in the order of execution before all other delay properties are applied. You simply have to set the delay arc to 1 (one). That will remove it. For example:

```
$rss{$rundir}{"delay_remove"}{"i[1-7] ^ * v"} = 1;
$rss{$rundir}{"delay_min_set"}{"i0 ^ out1 v"} = 200;
$rss{$rundir}{"delay_max_set"}{"i0 ^ out1 v"} = 250;
```

The example above first removes all delay arcs from i1 through i7 rising to any output pin falling. Then it sets an arc from i0 rising to out1 falling.

Another example:

```
$rss{$rundir}{"delay_remove"}{"*"} = 1;
$rss{$rundir}{"delay_min_set"}{"i0 ^ out1 v"} = 200;
$rss{$rundir}{"delay_max_set"}{"i0 ^ out1 v"} = 250;
```

The example above removes all delay arcs that were in the timing model and then creates a new one. This is the style you'd use if you just didn't want to use the timing model values and wanted to set your own instead.

NOTE: If you have conflicting regular expressions, it will choose the first one it sees in alphanumeric order.

delay_window_set This property lets you specify the delay "window" for the hsim ".tcheck" command. The delay window specifies the maximum delay difference that can be reported. This comes in handy when you have combinatorial delays. Combinatorial outputs sometimes have many inputs, and the simulator can't figure out if the change in the output was caused by a change in one input versus another input. So you can get false delay arcs seen in the simulator. To avoid this, you can specify a maximum delay of, say, one clock period. Any delay violation over this limit is discarded.

The usage of this property follows that of the "delay_min_set" property. Examples:

```
$rss{$rundir}{"delay_window_set"}{"*"} = 50;
$rss{$rundir}{"delay_window_set"}{"i0 ^ out1 v"} = 200;
$rss{$rundir}{"delay_window_set"}{"i[0-9]+ * out[0-9]+ *"} = 200;
```

The resulting .tcheck commands will have the "window" parameter set to the value you specify here.

delay_zero_margin When the flow encounters a single delay for a particular timing arc, it says that the min and max value for the delay arc are the same. But in order to check any delay in hsim, there needs to be some min and max margin around that delay arc. So the default behavior in this case is to print out a warning message in rss_flow_chkcf.pl and to add the delay_zero_margin value to it to get the max value and subtract it to get the min value.

So this artificially creates some margin when the min and max delay values are exactly equal to each other, and that allows it to be checked in hsim.

By default, the margin is set pretty small (less than a picosecond), so this means that if you accidentally missed the warning message, hsim's tcheck command will almost certainly trigger a timing check violation for this delay. At that point, you can determine the appropriate margin to add and just use the delay_max_add and delay_min_add properties to add and subtract from the max and min values.

Set this delay_zero_margin property if you want to change the flow's default value for this. But most people should leave this alone.

Example:

```
$rss{$rundir}{"delay_zero_margin"} = 1.5;
```

The example above adds 1.5ps to the max delay and subtracts 1.5ps to the min delay for any timing arc where the calculated min and max delay values are the same.

eckt By default, the flow will look for the eckt file at the following path for the given cell (eckt_cellname):

```
$CHIP/lib/eckt/${eckt_cellname}.ckt
```

However, if you explicitly specify the ECKT file here, it will use the one you specify. This is the extracted SPICE circuit from actual layout. It should not be a subckt file and should be flat and use silicon dimensions. It should not end in ".end".

eckt_cellname If you don't specify this, it assumes the cellname that the eckt circuit represents is the same as the cell property. In some cases, you might not have layout for the cell you want to analyze, but you do have layout for a cell that instantiates it. You can specify the name of the ECKT cell you want. This variable is also used to specify the name of the behavioral verilog model cell and it's this cell to which your random vectors should target.

WARNING: For now, the flow can't handle it if this property is set to anything other than the cell property. So it's best not to use it for now. But this means you need to enter a set of timings by hand instead of using a timing model. Later on the lower-level timing model will be extrapolated to the ECKT boundary, so you wouldn't have to do it by hand. For now that functionality is not yet ready.

env_* This allows you to create environment variables which get set prior to running most parts of the flow (parts 2, 3, and 4). Just name the property "env_$varname" where $varname is the name of the environment variable

For example:

```
$rss{$rundir}{"env_HSIMVER"} = "C-2009.06";
```

The property above causes the flow to set the environment variable "HSIMVER" equal to "C-2009.06".

By default, the flow might already set some environment variables automatically. The list of these variables can be found in the flow's defaults file:

```
$CHIP/random_spice/templates/defaults.pm
```

Look for the "envs" property which lists the environment variables that will be set automatically.

You can override these flow defaults in your control file. If you want to actually delete an environment variable setting, you can set it equal to "-" (the minus sign). For example:

```
$rss{$rundir}{"env_HSIMVER"} = "-";
```

The example above unsets the environment variable HSIMVER just in case it was set somewhere (like maybe in the flow's defaults).

ground This property lets you specify the ground node's name if you want something different from the default. The ground node is used by the flow when creating capacitors used by the load_set and load_set! properties.

Example:

```
$rss{$rundir}{"ground"} = "vss";
```

It would be rare that you would need to use this property. The flow has a default ground node name already, which varies based on the chip. For CN, the default ground node is "vss0". For chips prior to CN, it is "vss". The flow will also check to make sure the default ground node is defined as a supply (by the supply property), and if not it will change the default ground node to "gnd" which is always defined as a global ground regardless of the chip.

NOTE: If you specify a ground node name that isn't defined as a supply (with the supply property), then it will error out in part1. You can run the rss_flow_chkcf.pl script to make sure it will run prior to running the flow.

hold_add
hold_scale
hold_set
hold_remove

These properties operate on the hold-time values specified in the timing model. You can scale them by a certain value, add or subtract amounts from them, explicitly set their values, create new hold-time requirements, or remove existing ones.

hold_add: Add or subtract time to the hold-time values seen in the timing model.

hold_scale: Multiply the timing model's hold-time value by a number. This occurs before the hold_add property is applied.

hold_set: Set a hold-time to a value explicitly. If the hold-time requirement already exists in the timing model, this value overrides it. If the hold-time requirement does not exist already in the timing model, you will create one by using this property without regular expressions. NOTE: This property overrides the hold_add and hold_scale properties.

hold_remove: Remove a hold-time requirement. This occurs first before all other properties.

If you're confused about which properties take precedence over which other properties, this may help. The flow will execute these properties in the following order: hold_remove, hold_scale, hold_add, and lastly hold_set.

All of these properties require you to specify the input pin followed by its edge and then the reference pin and its edge. The edges can be either "^" for rising, "v" for falling, or "*" for both rising and falling. Each of these four items must be separated by a single space character. And each of these four items may contain regular expression characters. If you use a regular expression character, the hold-time requirement must already exist in the timing model for it to match. And all hold-time values are in units of picoseconds.

NOTE: If there are multiple hold-time requirements inside of the timing model for the same input pin/edge and reference pin/edge, then the maximum hold-time value is selected.

Here's an example:

```
$rss{$rundir}{"hold_add"}{"*"} = 10;
$rss{$rundir}{"hold_scale"}{"*"} = 1.25;
$rss{$rundir}{"hold_set"}{"a1 ^ ph1 ^"} = 200;
$rss{$rundir}{"hold_set"}{"a2 * ph1b v"} = 130;
```

The example above first scales all hold-times in the timing model by 1.25. Then it adds 10ps to them. Then it sets the a1 rising hold-time with respect to ph1 rising to 200ps. And it then also sets the a2 rising or falling hold-time with respect to ph1b falling to 130ps.

The hold_remove property is special. It occurs first in the order of execution before all other hold-time properties are applied. You simply have to set the hold-time arc to 1 (one). That will remove it. For example:

```
$rss{$rundir}{"hold_remove"}{"i.* * * *"} = 1;
$rss{$rundir}{"hold_set"}{"i0 ^ ph1b v"} = 200;
$rss{$rundir}{"hold_set"}{"i0 ^ ph1b v"} = 250;
```

The example above first removes all hold-time requirements on all pins beginning with "i". It does this regardless of which pin they're referenced to and which edges the input or reference pins have. Then it sets a hold-time requirement on pin i0 rising with respect to ph1b falling.

Another example:

```
$rss{$rundir}{"hold_remove"}{"i1 ^ ph1 ^"} = 1;
```

The example above removes a specific hold-time requirement on pin i1 rising with respect to ph1 rising.

insert Use this option to insert anything into your main.spi file. This is a little like using the spice_include property, but instead of having to create an additional file, you can just use this property to tell the flow exactly what you want inserted into the main.spi file.

Here's an example:

```
$rss{$rundir}{"insert"} = ".probe v(gh*) level=1";
```

The example above inserts the following directly into your main.spi file:

```
.probe v(gh*) level=1
```

If you have more than one line you'd like to insert, you can do it by using the "\n" (newline) character:

```
$rss{$rundir}{"insert"} =
".probe v(i0)\n.measure tran i0 when v(ph1)=vth rise=1";
```

The example above inserts the following directly into your main.spi file:

```
.probe v(i0)
.measure tran i0 when v(ph1)=vth rise=1
```

An easier way to do this would be to actually insert carriage returns directly into the string. You can do this in Perl:

```
$rss{$rundir}{"insert"} =
".probe v(i0)
.measure tran i0 when v(ph1)=vth rise=1";
```

Notice in the example above, the "\n" character wasn't needed, since you inserted a carriage return into the string directly.

limit_kept_logs The flow will run continuously on a given rundir. Each time a new run occurs, if the old run had any test-vector mismatches or timing check violations, it copies the important files and logs to the "previous_violations" directory inside of the rundir, making sure not to overwrite any of the existing files in that directory. This is done so that previous log files showing any violations are archived for later inspection. It will not archive the results of a clean run, by the way.

But this causes the number of files to increase with each run that has violations. These files can accumulate over time. They get deleted only when the netlist or vector header file has changed. So instead of letting them build up out of control, you can use this property to set a limit on how many previous runs can be in the previous_violations directory. By default, it's set to 4. You can set it to a higher limit or a lower limit if you want.

If the number of runs in the previous_violations directory exceeds the limit, then the most recent runs are kept and the oldest runs are removed.

Also note that this controls whether a new run will happen at all. If the limit has already been reached, no new run will occur. This saves on run time.

Example:

```
$rss{$rundir}{"limit_kept_logs"} = 10;
```

The example above limits the number of failing runs to 10.

limit_logfiles Whenever the SPICE file changes or when the vector header file changes, the flow interprets this as a brand new design to check. It may even be just a minor change. Whenever this happens, it will archive the current logfile off to a file starting with:

```
logfile.1.gz
```

Then it begins a new logfile called "logfile". The next time such a change occurs, it archives the logfile to the next highest number. In this cased "logfile.2.gz".

You can use this option to limit the number of logfiles you want to keep around. By default it's set to 20. As a result, only the 20 highest numbered logfiles will be kept each time. The others will be destroyed. Set this to 0 if you don't want any logfiles archived at all (so you'll just have the "logfile" file and no archived logfiles). Set this to -1 if you don't want to delete any of the old archived logfiles ever (so they'll just continue to be created and never get deleted).

Example:

```
$rss{$rundir}{"limit_logfiles"} = 3;
```

The example above will tell the flow to keep just the last 3 logfile archives.

limit_vec_size Normally, the number of vectors you run comes straight out of your "$eckt_cellname.vectors" file which is generated by the tv2sim_cmd command. It specifically looks for the comment in that file:

```
; There were 1600 vectors
```

When it finds that comment, it's able to get the total number of vectors. And this number is used to determine how many nanoseconds to simulate.

But in some cases, you may want to override this number by setting a limit on the number of vectors to simulate. You'd do this if your vector command file generated, say, 1000 test vectors, but you only care about the first 100. So you'd do:

```
$rss{$rundir}{"limit_vec_size"} = 100;
```

Note that this is a limit, not an actual vector size. If the number of vectors is actually less than this number, then the lesser number is used. But if it's greater than this number, then the limit's value will be used.

An alternative method of doing this would be to read a command line argument in your vector command file and simply pass the number of vectors to it instead:

```
$rss{$rundir}{"vec_cmd"} =
   "$CHIP/random_spice/user/weigand/myvec.pl 100";
```

The example above passes "100" to your vector command file. At this point your command file should be reprogrammed to take the command line argument and limit the number of vectors it outputs to this amount. That's something you'd have to do.

load_set
load_set!

These properties allow you to set an output load for a given output pin. The output load is added to the SPICE circuit in the form of a capacitor with the positive terminal on the pin in question and the negative terminal on ground (as specified by the ground property). It's also used to determine the delay arc values from input pins to that output pin. The units are in femptoFarads.

Please note that the total load on any output pin is this user-given external load plus the timing model's internal output pin cap. These properties will not override the timing model's internal output pin caps, they add to them.

load_set: Use this property to set the external output pin load cap.

load_set!: Same exact behavior as load_set.

The syntax for this property is the same as the syntax for the arrival_set property. For example:

```
$rss{$rundir}{"load_set"}{"*"} = 50;
$rss{$rundir}{"load_set"}{"q"} = 125;
$rss{$rundir}{"load_set"}{"qb"} = 100;
```

The example above shows an external load cap value of 50fF for all output pins except for the "q" pin which has a load of 125fF and the "qb" pin which has a load of 100fF.

mail You can specify a list of email users who will be emailed in the eventuality of this rundir having a failing vector or a timing check violation. Separate usernames by spaces. For example:

```
$rss{$rundir}{"mail"} = "weigand stanho khoi";
```

The example above will cause the flow to send email to users "weigand", "stanho", and "khoi" when the flow detects a problem. They will get a message from user "bot" telling them the name of the rundir which is failing. Usernames can include addresses as well.

By default, it will not email anyone.

NOTE: Crashed jobs will always trigger an email to be sent to the person who owns the control file if this is an automated run (not manual). Or if this is a manual run, crashed jobs will always trigger an email to be sent to the person who is currently running the manual job (not necessarily the owner of the control file). This occurs regardless of whatever is specified for the "mail" property. In fact, you could leave the "mail" property undefined, and it would still send emails in the event of a crashed run. The "mail" property is just used when there's a failing vector or a timing check violation, not for crashes.

mail_success You can specify a list of email users who will be emailed when the flow finishes a job successfully. A successful run has no vector mismatches and no timing check violations. Separate usernames by spaces. For example:

```
$rss{$rundir}{"mail_success"} = "weigand stanho khoi";
```

The example above will cause the flow to send email to users "weigand", "stanho", and "khoi" when the flow finishes running a job successfully. They will get a "congratulations" message from user "bot" telling them the name of the rundir which ran successfully. Usernames can include addresses as well.

By default, it will not email anyone.

mainspi_cmd The "main.spi" file is the main SPICE file that the simulator uses. It loads the technology file, reads the circuit definition, sets some parameters, and applies the test-vectors as well as other simulation analysis commands.

The "main.spi" file is based on the mainspi_template template file. The mainspi_cmd command is run to

generate the "main.spi" file based on the template file as well as variables set in the control file. Generally there should be no reason for you to have to use an alternate command to do this, but if you do need to use it, you can. By default, it's set to:

/vlsi/cad2/bin/rss_flow_genmainspi.pl

Even though the default shown above doesn't show any command line arguments, it doesn't mean that you can't give them. If you want, you can include any number of command line arguments along with your command.

The requirement is that it produces this output file in the working directory:

./main.spi

It logs stdout and stderr to file:

./main.spi.log.gz

If your script exits with any status code other than 0, it will cause the flow to abort.

mainspi_template The "main.spi" file is the main SPICE file that the simulator uses. This variable can be used to specify the template file which is used to create the "main.spi" file. The mainspi_cmd script uses this template file to generate the "main.spi" file.

By default, the template file is set to:

$CHIP/random_spice/templates/main_spi_template

There may be other default template file settings depending on the SPICE corner and such.

See also: sim_type.

param_* You can set any parameter to anything you want. This is a general way of creating ".param" lines in your main SPICE netlist. For example:

$rss{$rundir}{"param_hsimalloweddv"} = "'vcc/10'";

The property above causes the flow to create this line in your SPICE file:

.param hsimalloweddv='vcc/10'

You could also make up your own parameter names:

$rss{$rundir}{"param_SteveWuzHere"} = 1;

The property above causes the flow to create this line in your SPICE file:

```
.param SteveWuzHere=1
```

The flow has default parameters which will be set automatically if you don't specify them. They are: hsimalloweddv, hsimcc, hsimspice, hsimsteadycurrent. Their defaults are:

```
.param hsimalloweddv='vcc/10'
.param hsimcc=1
.param hsimspice=2
.param hsimsteadycurrent='1nA'
```

The hsimalloweddv parameter: A larger number results in less accuracy and greater speed. A smaller number results in more accuracy but less speed.

The hsimcc parameter: Setting it to 1 causes Hsim to use the charge conservation models thereby increasing complexity of the MOSFET calculation. Set to 0 to disable. You'll want to use 1 if you set hsimspice to 2 or higher.

The hsimspice parameter: Setting it to 1 causes Hsim to use the simple MOSFET model. Setting it to 2 results in a more complex model. And so on.

The hsimsteadycurrent parameter: Recommendation is to set this to "50pA" when you're simulating at low voltages and when hspimspice is set to 2 or higher.

See also: sim_type.

probe_level Use this option to set the voltage probe level during an HSIM simulation. Setting it to "1" causes it to probe just the first level of hierarchy. Setting it to "2" causes it to probe the first and second levels of hierarchy. And so on. Setting it to "all" will probe all levels of hierarchy. Setting it to "0" causes it to skip probes. Setting it to "all_io" will have it probe just the IO pins.

NOTE: If you specify "all_io", then you must provide either a verilog netlist or a timing model.

Example:

```
$rss{$rundir}{"probe_level"} = "all";
```

The example above results in this command added to main.spi:

```
.probe v(*)
```

Here's another example:

```
$rss{$rundir}{"probe_level"} = 1;
```

The example above results in this command added to main.spi:

```
.probe v(*) level=1 filter="*/*"
```

Notice that the filter is there to remove any flattened hierarchy if this is a flat netlist. With flat netlists, there is no hierarchy (or the hierarchy has been flattened to a certain level already). And so the "/" character is used to indicate flattened hierarchy. The filter will remove it.

Another example:

```
$rss{$rundir}{"probe_level"} = 2;
```

The example above results in this command added to main.spi:

```
.probe v(*) level=2 filter="*/*/*"
```

Another example:

```
$rss{$rundir}{"probe_level"} = "all_io";
```

The example above results in this command added to main.spi:

```
.probe v(i0)
.probe v(i1)
.probe v(i2)
.probe v(o1)
```

In other words, it uses the verilog netlist or the timing model to determine the names of all IO pins. Then it probes each of them specifically instead of using a wildcard.

The default is set to 1 for this property.

queue_prep
queue_sim
queue_sum

These properties let you specify the names of the queues in which your jobs run. There are default values for these specified by the flow, but you can change them if you need to.

The queue_prep property specifies which queue the data-prep jobs will run in. These jobs are for running part 2 of the flow which typically runs the runv commands which run Verilog-XL simulations and some minor other stuff.

The queue_sim property specifies which queue the SPICE simulation jobs will run in. (Part 3 of the flow.) Please note that this should no longer be necessary. Specify the sim_type property instead.

And the queue_sum property specifies which queue the summary jobs will run in. (Part 4 of the flow.)

The properties can be set according to the following format:

```
@{$rss{$rundir}{"queue_prep"}} =
    ($qname1,$qname2,...);
```

So for example, suppose you wanted to run the prep jobs in both the linux queue and the unixfast queue. You could do this by using the following code:

```
@{$rss{$rundir}{"queue_prep"}} = ("linux","unixfast");
```

The example above shows that all data-prep jobs will run in either the linux queue or the unixfast queue. Because "linux" is listed first, the flow will try to send jobs to that queue first. If it fills up, then jobs will be submitted to the unixfast queue.

Keep in mind, though, that the data prep portion of the flow should be run in the "vxl" queue, because that queue is for Verilog-XL simulations. Running the runv command outside of this queue may give you a crashed run.

Remember, you must always specify a list of values, but that list can include just a single element if you want. For example:

```
@{$rss{$rundir}{"queue_sim"}} = ("hsimms");
```

Oh, and if you do specify the hsimms queue, make sure you also do this:

```
$sim_cmd = &main::rssGimmeDefault($rundir,"sim_cmd");
$sim_cmd =~ s/\-lco \S+/-lco hsim-ms/;
$rss{$rundir}{"sim_cmd"} = $sim_cmd;
```

That tells the hsim command itself to use the hsim-ms license. So not only do you need to use the queue_sim property, but you also need to modify the sim_cmd property.

The flow knows the names of all valid queues and will report an error if you list a queue name that's not supported. The flow also knows how many jobs can run at any given moment in each queue. So all you would do is specify the names of the queues you want and list them in the order you want.

The names of the queues are the same as those used in:

```
/vlsi/cad/queue/current/b.conf
```

Why would you want to set these properties instead of letting the flow decide their default values? Consider this. Suppose your circuit has over 100,000 transistors. In this case, the Hsim tool requires you use the "hsim-ms" license. But you should only run that license in the "hsimms" queue. So you would specify the queue_sim property to use the

hsimms queue instead of the default hsim queue.

There could be other reasons you want to send your jobs to different queues than the default ones. For example, if you decide you want to use Hspice instead of Hsim, you would set the queue_sim property to use the hspice queue. Or if you had to use a special tool with a special license in the data-prep stage (maybe to run verilog simulations), then you'd specify a different queue name using the queue_prep property.

See also: sim_type, sim_cmd.

run Specify whether or not you want to run the flow. Set to 0 if you don't want to run or 1 if you do want to run. If you set it to 0, it prevents any further runs of the flow, but keeps the old results around.

There are no default values. It's a required setting. If it's blank or not defined at all, then the $rundir will be deleted if it ran previously. You can achieve this by commenting out those settings in your control file which setup the given $rundir.

runv_bhv_cmd This allows you to specify an alternative command for "runv" which is used to generate behavioral level test vector output. If you don't specify this, it defaults to:

```
$CHIP/bin/runv \
   -top ${eckt_cellname}_test.v \
   -noverror
```

The requirement is that it produces these output files in the working directory:

```
./${eckt_cellname}_test.tv
./verilog.log
```

NOTE: The flow will rename and compress this log file to "verilog.1.log.gz".

Also, the flow will create a log of stdout and stderr to:

```
./${eckt_cellname}_test.tv.log.gz
```

If your script exits with any status code other than 0, it will cause the flow to abort.

runv_ckt_cmd This allows you to specify an alternative command for "runv" which is used to generate circuit-level test vector output. If you don't specify this, it defaults to:

```
$CHIP/bin/runv \
   -top ${eckt_cellname}_test_test.v \
   -y $CHIP/lib/fundamental \
   -y $CHIP/lib/vnet
```

The requirement is that it produces these output files in the working directory:

./${eckt_cellname}_test_test.tv
./verilog.log

NOTE: The flow will rename and compress this log file to "verilog.2.log.gz".

Also, the flow will create a log of stdout and stderr to:

./${eckt_cellname}_test_test.tv.log.gz

If your script exits with any status code other than 0, it will cause the flow to abort.

setup_add
setup_remove
setup_scale
setup_set

These properties operate on setup-time values. They work just like thier hold-time counterparts:

hold_add
hold_remove
hold_scale
hold_set

setup_or_hold Specify whether this is a setup-time or hold-time analysis run. For setup-time runs, you'd set this property to "setup". For hold-time runs, you'd set it to "hold".

This is used by the flow to determine when inputs transition. The setup-time and hold-time values in the timing model are used to set this. So you need to select which of the two you want to use.

If you're not using a timing model, whatever you set it to here doesn't really matter. But you still need to set it to "setup" or "hold". Or else the flow will not run.

sim_cmd This allows you to specify an alternative command for the SPICE simulator. If you don't use this, then the command is determined from looking at the sim_type setting to figure out which simulator it should use and then looking up the default command line for that simulator type.

You should only set this option if you're trying to manually override the flow. In the past it was necessary to use this option in order to specify the hsim ms license or the "nostress" version of the hsim binary. Now this is all handled automatically for you, based on the options you specify with the sim_type property.

For hsim, this defaults to something like this for the regular binary and the "sc" license:

/Server/bin/hsim_stress \

```
-lco hsim-sc \
-i main.spi \
-o main
```

The "-lco hsim-sc" option specifies that you only want to use the hsim-sc license. If you don't specify this, it grabs any available license to run hsim, which means you could grab the special hsim-ms license. Since the hsim-ms license is special, and we only have one of them, you must specifically change this property to have "-lco hsim-ms" as the option. And you must use the queue_sim property to send it to the "hsimms" queue instead of the default queue.

The "main.spi" file is actually generated by the flow in the working directory using the mainspi_cmd. It's based off of a template SPICE file. This main SPICE file is used to setup the simulation, load the technology file, load the circuit file, and run the vectors.

You can change this property to run a script or some other SPICE simulator besides hsim. This may be needed in case you need to hack some input files before running hsim. You can also add command line options by using this variable.

This script is required to create the following files, although it is free to create others:

```
./main.log
./main.fsdb
```

The flow will dump stdout and stderr to the following file:

```
./sim.log.gz
```

If your script exits with any status code other than 0, it will cause the flow to abort.

See also: sim_type, queue_sim.

sim_step_time This lets you control the SPICE simulation's transient analysis step time (ie, the resolution). It's set to some default value for the project. For example, for the c5i project it's set to 1ps.

Example:

```
$rss{$rundir}{"sim_step_time"} = 5;
```

The example above sets the step value to be 5ps. This value will result in a SPICE file being generated with this line:

```
.tran 5ps "vectors*per/2"
```

It may be useful to override this if your output database file (main.fsdb) is getting to be too large, for example. Like if you have a corner you're running at which requires very long clock cycles, you could safely increase this number to save disk space and increase run times.

sim_type In the past you might have had to set various SPICE simulation properties to control things such as which hsim licenses (sc or ms) to use or which queue to use (the hsim or hsimms queue). Now however, the sim_type property controls all of this for you. You can think of it as the master property, and the other properties are its slaves. Like a kind of auto-pilot capability for SPICE simulations.

The properties this affects are:

- o sim_cmd
- o queue_sim
- o mainspi_template
- o vechead_template
- o param

All of those properties control the execution of the SPICE simulation: the SPICE simulator command line execution, the queue which runs the SPICE simulation, the SPICE template files, and the default SPICE parameter settings. The sim_type property provides a simple way to set all of these correctly.

You no longer have to set these properties yourself unless you want to. If you do set any of them, they will override the sim_type property. It's best to leave them blank if you're using sim_type.

The sim_type property is entirely optional. If you don't use it, the flow will behave the same as it has in the past prior to introducing this new property. The defaults for all of these properties are contained in the flow's defaults.pm file:

$CHIP/random_spice/templates/defaults.pm

Usage:

```
$rss{$rundir}{"sim_type"} = "$option1 $option2 ...";

Where $option1, $option2, etc. are space delimited keywords.

Simulator selection (required, must choose one):
   hsim        -> Use the hsim SPICE simulator.
   hspice      -> Use the hspice SPICE simulator.
   xa          -> Use the XA SPICE simulator.

Queue / license selection options (choose zero or more):
   bat         -> Force all jobs to batch queue.
   nobat       -> Force it to never use batch queues at all.
   sc          -> Force "sc" queue / licenses (only for hsim).
   ms          -> Force "ms" queue / licenses (only for hsim).
   nostress -> Force it to use the no-stress queue and
                    special no-stress hsim binaries and licenses
      (only for hsim).
```

```
bigmem      -> Use only "big memory" machines.
nobigmem -> Specify that you do not want "big memory"
                machines.
monte_#    -> Specify montecarlo run (hsim and hspice only),
                the number after it is the number of simulations.
32             -> Specify 32-bit binary executable (for xa only).
```

You don't have to specify the sim_type property, but if you do use it, you must set it equal to at least one of: hsim, hspice, or xa to begin with. Then you can follow it with a list of additional options separated by spaces.

Example:

```
$rss{$rundir}{"sim_type"} = "xa";
```

The example above sets the SPICE simulator type to "xa". That tells the flow how to format the SPICE command files. It also determines the queue to which the job will be submitted. The flow will then set the following properties for you automatically:

```
$rss{$rundir}{"sim_cmd"} =
   "/Server/bin/xanq -hspice main.spi -out main -wavefmt hspice";
$rss{$rundir}{"queue_sim"} = ["xa","xabat"];
$rss{$rundir}{"mainspi_template"} =
   "$CHIP/random_spice/templates/mainspi_template_xa";
$rss{$rundir}{"vechead_template"} =
   "$CHIP/random_spice/templates/vechead_template_xa";

... and then a series of "param_*" parameter settings
      unique to xa.
```

See how much work that saves you?

You should be able to see exactly what each of those property settings get set to when you use the rss_flow_chkcf.pl command.

Here's another example:

```
$rss{$rundir}{"sim_type"} = "hsim monte_100 nobat nobigmem";
```

The example above specifies an hsim montecarlo run with 100 simulation iterations in the non-batch, non-big-memory queues only. The "nobigmem" option was necessary in this case, because montecarlo runs automatically go to the big-memory machines by default without it. Also, the flow will automatically determine whether to use an "sc" or an "ms" license, since that was not specified by the user.

Keep in mind that some combinations are not allowed. You will get an error message during

rss_flow_chkcf.pl (part 1 of the flow) if that is the case. For example, you can't combine the "xa" and "nostress" options, because there isn't a no-stress version of the "xa" binary.

Options: hsim, hspice, and xa

For the main SPICE simulator, you can only specify one of: hsim, hspice, and xa. Currently, those are the only ones officially supported by the flow. This list will change in the future. The default simulator is "hsim" if you don't specify the sim_type property at all.

Some things to note about each simulator:

+ Timing checks (delay checks) are not supported at all for hspice just yet. I hope to allow these soon. No errors are given, but you'll see a warning in the log files.

+ Timing checks are handled properly for both hsim and xa. In hsim, it is handled through the ".tcheck" command. In xa, it is handled through the "check_timing_edge" command.

+ Output check windows are not supported in hspice, even though it supports the vector file format. This means that running vectors through hspice is currently broken. I plan to support hspice check windows soon. No errors are given if you do use hspice with output check windows, just a warning.

+ If you use XA, the vhth and vlth properties must be a function of vcc or just a numeric value. There should be no other parameter names, supply names, function names, or symbols in it. Only "/", "*", "+", "-", and "()" are allowed as math operators. For example, "vcc/2" is good, but "myvhth" is not. And "(vcc+0.5)/2" is good, but "vcc/sqrt(3.14)" is not. An error will result if this is violated. The default values are "vcc/2" for both, which is acceptable. In addition to this, if "vcc" is used, it must be defined in the property xa_vcc, or it must appear somewhere in the SPICE technology file (or any one of the included files or lib files inside of it). If it is defined in the SPICE technology file, it must be defined as a simple parameter name and value, such as:

.param vcc = 1.0

The reason for this limitation is that XA needs to know specific values for the vhth and vlth properties when used in the XA_CMD options. It can't use SPICE expressions. So the RSS flow needs to evaluate the vhth and vlth expressions, and for that it needs to know the vcc value.

An error will result if anything is not right with any of the above.

Options: bat and nobat

The "bat" and "nobat" options allow you to control whether or not the jobs will go to batch queues. By default, your jobs will go to both the regular queues and the batch queues.

The "bat" option means that you want all jobs to go to batch queues (they are low priority jobs that can be killed and rescheduled).

The "nobat" option means that you want all jobs to go to the non-batch queues (but you are limited in the number of jobs that can be run at any given time).

There are batch queues for hsim, hspice, and xa.

Currently there are no batch queues for the "bigmem" machines, so don't specify "bat" and "bigmem" together.

Note that the rss_flow.pl command has its own -bat and -nobat command line options which override the control file's options if they are given.

Options: sc and ms

These options control the hsim license usage, which also implies which queues the jobs will run in. If you want to use an ms license and run in the ms queues, specify "ms". If you want it to use an sc license, specify "sc". The "ms" licenses are only to be used when your transistor count is 100,000 or more. Use "sc" if it's less.

You can specify "ms" even if you have less than 100,000 transistors in your netlist. This is sometimes done to force the job to use the ms queue machines which might have more memory than the regular queue machines. But it still uses the ms licenses in that case, so only do this as a last resort if memory is an issue.

Keep in mind the flow now counts how many transistors you have in your eckt netlist on the fly. So you generally won't have to use these options. The subckt name it uses when it counts transistors, by the way, is determined by the cell property. If it can't find a subckt by that name, it assumes your cell is actually not a subckt cell but is instead the "global" (top) level of hierarchy in the SPICE file. If the count is ever zero, then it results in an error - it would indicate a potential bug in the transistor counter algorithm.

You would only use "ms" if you knew ahead of time that your transistor count is 100,000 or more, or "sc" if you knew it was less. This would save the flow some time trying to count transistors, but it's only a small amount of time. You're better off not using these options, because your netlist's transistor count could change at some point, and you might not realize it before the flow runs again. (And keep in mind extracted netlists often have many times more transistors in them than schematic netlists.)

You can not use these options with "xa" or "hspice". It only applies to hsim runs.

Options: bigmem and nobigmem

These options control whether or not the jobs are going to "big-memory" machines. The "bigmem" option specifies that your jobs are going to big-memory machines only. The "nobigmem" option specifies that you don't want them to go to big-memory machines.

The default is to send jobs to the regular, non-big-memory machines, except if you specify the "monte_#" option (for montecarlo jobs). For montecarlo jobs, they will go to big-memory machines by default, but you

can specify "nobigmem" to force them to go to the regular, non-big-memory machines instead.

Presently, only "hsim" jobs have a big-memory queue. These options do not work for "xa" and "hspice" jobs. The hsim no-stress queue also lacks a big-memory queue at present.

How do you know whether or not to use the "bigmem" option? If your job crashes, you can look at the bottom of the sim.log.gz log file to determine if there was "Killed" message in it. If so, it usually means it was using too much memory. Also, you can look at your job's memory statistics (which are printed to the screen after your job completes), and it will tell you.

Here's the present (Jan 2010) list of minimum memory for all hsim queues:

hsim = 6-8GB
hsimms = 28-32GB
hsimbg = 64GB
hsimmsbg = 64GB

So if your hsim-sc job is taking up more than 8GB of memory, it might be best to specify "ms" to send it to the ms queue where there are 32GB machines. Whereas if your hsim-ms job is taking up more than 32GB of memory, specifying "bigmem" would take it to 64GB machines.

Options: monte_#

This option can be specified if you're doing a montecarlo run. Specifying this causes the flow to format your main SPICE file a certain way and to use certain montecarlo options and parameter settings. It also causes more information to be reported in the summary and log files.

You must specify the number of simulations in the option itself. This must be an integer number greater than 0. For example, "monte_10" would do 10 simulations. Whereas, "monte_500" would do 500 simulations. The more you do, the more memory it uses.

This presently only works for hsim and hspice. It will error out if you try to do this with xa. No ETA on xa montecarlo support in the future.

Note that montecarlo runs for hsim and hspice set a random number seed to the current time (in seconds). This is limited to numbers between 1 and 259200, according to the hspice spec. So the algorithm it uses for seeding is:

.option seed = $val
Where $val = (time % 259200) + 1

Options: 32

This only applies to XA. The XA binary executable is 64 bit by default. Specifying "32" here will

force the flow to use the 32 bit binary instead. There should be no reason to use this, but it's here if you do need it. There may be run speed or memory utilization differences, but they should be small.

It should be noted that this list of options can be expanded in the future to handle special needs, new simulators, etc. Just come to me if you have anything you'd like for the flow to automatically understand.

See also: queue_sim, sim_cmd, mainspi_template, vechead_template, param.

skip_tv_diff This property can be used to specify that you don't want to run the runv_ckt_cmd and instructs the flow not to perform the test-vector comparison script tv_diff_cmd.

This is useful because sometimes there may be a problem with the verilog simulation of the circuit netlist which invalidates the comparison against the verilog simulation of the behavioral netlist. Rather than worrying about it, you can set this property to skip it. This comparison would normally be used to detect differences between the circuit and behavioral netlist early on before going to SPICE simulation (which takes a long time). But since your SPICE simulation uses the behavioral-level test vector output, this comparison is not necessary. The SPICE simulation will detect any test-vector problems vs. the expected behavioral-level output.

Here's what you would do to disable comparisons (ie, to use this property):

```
$rss{$rundir}{"skip_tv_diff"} = 1;
```

Setting it to 1 causes the property to be used. Setting it to 0 causes it to be ignored.

slew_rise_clock
slew_fall_clock
slew_rise_data
slew_fall_data
slew_rise
slew_fall

Specify the input pin rising and/or falling slew in picoseconds.

NOTE: These are 20-80% slews. But the input waveforms generated in SPICE will be straight lines, not exponentials. It will simply convert your 20-80% slew values into 0-100% straight-line slews by multiplying by 5/3 (which is approximately 1.66667).

slew_rise_clock and slew_fall_clock: These properties let you specify a default value for clock rising and falling input slew. If you don't specify this property, the flow sets it to a default value.

slew_rise_data and slew_fall_data: These properties let you specify a default value for non-clock rising and falling input slew. If you don't specify this property, the flow sets it to a default value.

slew_rise and slew_fall: These properties allow you to set specific rising and falling slew rates for individual pins instead of default values. It overrides the slew_rise_clock, slew_fall_clock, slew_rise_data, and slew_fall_data

properties. You can specify a pin name or a regular expression.

You may be perfectly content just to set default values for input slews. If not, you can use the slew_rise and slew_fall properties for specific input pins.

Here's an example:

```
$rss{$rundir}{"slew_rise_clock"} = 25;
$rss{$rundir}{"slew_fall_clock"} = 25;
$rss{$rundir}{"slew_rise_data"} = 35;
$rss{$rundir}{"slew_fall_data"} = 35;
$rss{$rundir}{"slew_rise"}{"i[0-7]"} = 45;
$rss{$rundir}{"slew_fall"}{"i[0-7]"} = 45;
$rss{$rundir}{"slew_rise"}{"ph1"} = 30;
$rss{$rundir}{"slew_fall"}{"ph1"} = 30;
```

The example above sets the default clock 20-80% slew rate to 25ps and the default data input slew rate to 35ps. In addition to this, there is a setting of 45ps for the slew on signals i0 through i7 (by way of a regular expression). And there's a specific setting of 30ps for the slew on the ph1 signal.

In the example above, if the slew_rise and slew_fall properties weren't applied, then the slew for signals i0 through i7 would be 35ps (assuming they're not clocks). And the slew for signal ph1 would be 35ps (assuming it's a clock).

Here's another example:

```
$rss{$rundir}{"slew_rise_clock"} = 25;
$rss{$rundir}{"slew_fall_clock"} = 25;
$rss{$rundir}{"slew_rise_data"} = 35;
$rss{$rundir}{"slew_fall_data"} = 35;
$rss{$rundir}{"slew_rise"}{"*"} = 50;
$rss{$rundir}{"slew_fall"}{"*"} = 50;
$rss{$rundir}{"slew_rise"}{"ph1"} = 25;
$rss{$rundir}{"slew_fall"}{"ph1"} = 25;
```

This example above sets default clock and data slews, but since the slew_rise and slew_fall properties were specified with "*" as the regular expression, it means that the default slews will never be applied. The "*" regular expression will ensure that all signals have 50ps input slew. And since there's also a setting for "ph1", it overrides the "*" setting, because it's not a regular expression (specific pin names are given higher precedence than regular expressions).

spice_include You can optionally include a SPICE file in the main SPICE file that HSIM runs on. It will be included as a ".include" line. This file can be used to do anything you want. You might add special timing checks or probe voltages or whatever. Example:

```
$rss{$rundir}{"spice_include"} =
```

"$CHIP/random_spice/user/weigand/myinclude.spi";

spice_techfile You need to specify the SPICE technology include file using this property. If you don't give an absolute path to the file, it assumes the file is relative to the following directory:

$CHIP/technology/hspice/

For example:

$rss{$rundir}{"spice_techfile"} = "tt_tt";

sum_cmd This command is used to summarize the simulation results into a report file. You should just use the default command, but you can specify an alternative script if you'd like. The default is set to:

/vlsi/cad2/bin/rss_flow_summarize.pl

Even though the default shown above doesn't show any command line arguments, it doesn't mean that you can't give them. If you want, you can include any number of command line arguments along with your command.

The requirement is that it produces this output file in the working directory:

./${eckt_cellname}.sum.gz

A log of stdout and stderr will be generated to:

./${eckt_cellname}.sum.log.gz

If your script exits with any status code other than 0, it will cause the flow to abort.

supply Use this property to set the voltage level of a power supply during a SPICE simulation. You only need to use this if the default power supply settings are wrong or if the flow doesn't find all of the power ports you need.

For example:

```
$rss{$rundir}{"supply"}{"vdd0"} = "'vcc * 0.7'";
$rss{$rundir}{"supply"}{"vdd1"} = "1.5";
```

The example above will insert the following two lines into the SPICE file:

```
vsupply_vdd0 vdd0 0 dc 'vcc * 0.7'
vsupply_vdd1 vdd1 0 dc 1.5
```

Please note that none of these are global power supplies. They will appear at the top level of your SPICE netlist and are considered local to that level of hierarchy. If you want to make them global, you can simply put a "!"

(exclamation point) character at the end of the power supply name, and the flow will interpret that to mean that you want to make them global power supplies. For example:

```
$rss{$rundir}{"supply"}{"vdd!"} = "vcc";
$rss{$rundir}{"supply"}{"vss!"} = "0";
```

The example above inserts the following into the SPICE file:

```
.global vdd vss
vsupply_vdd vdd 0 dc vcc
vsupply_vss vss 0 dc 0
```

Ordinarily, the flow automatically figures out which supplies are needed and what their voltage levels should be. How it works is as follows: First, it has a list of power supply pins it looks for. This can vary based on the chip, but for CN for example, the pins would be vdd0-9, vss0-9, vbpa-j, and vbna-j. So if it finds any of those pins in the IO interface for the cell in question, it will create voltage sources in the SPICE netlist for them. The flow also has a default voltage setting for each of these. For vdd0-9 and vbna-j, for example, the setting is "vcc" (the default voltage for the corner as specified in your HSPICE technology file). For vss0-9 and vbpa-j, the setting is zero. But for this to happen by default, the IO terminals must be defined somewhere, either in the verilog netlist or the timing model, if given. If they aren't given anywhere, you must define them using this property.

Just to reiterate, if you didn't give it a timing model or a verilog netlist, it won't have a list of IO terminals. And so it might not create all of the supplies you need. So you would use this property to create the ones it is missing. You would also use this property if you didn't like the default voltage settings.

NOTE: No wildcards or regular expressions are allowed here.

Also, regardless of whether or not the flow finds power ports on your IO boundary, it will automatically create some power supplies by default, as a matter of convenience for you so that you won't have to keep defining them. The power supplies it creates by default depends on the chip, but for CN they are: "vdd0", "vss0", "vbna", and "vbpa". Those are all local power supplies. For all chips, "gnd" is defined as a global ground node, because it's used in the XRC extracted SPICE netlists and the HSPICE technology files. For all chips previous to CN, "vdd" is defined as a global power node and "vss" is defined as a global ground node. (And temporarily for CN, global "vdd" and "vss" are defined as global supplies until all eckts are extracted using local power supplies.)

One other thing... If you give it a verilog netlist, it will also use the verilog's "supply0" and "supply1" statements (if present) to assign power supplies if all of the above doesn't identify them as power ports. It will just assume the default voltage for "supply0" is zero volts and the default for "supply1" is "vcc" volts.

Bottom line is that you probably never need to use this property. Even if you don't give it a timing model or verilog netlist so that it can detect the power supply ports, it will still generate the commonly used power supplies by default. You would only need to use this property if you're simulating a cell that has more power supplies than the default ones (and you don't specify a timing model or verilog netlist), or if you'd like to specify an alternative voltage level than the default one. If ever in doubt, use the rss_flow_chkcf.pl script to see what it thinks are the power supplies, so you can

tweak it before you run the flow.

See also supply_remove.

supply_remove This property allows you to remove power supplies that would normally be created by the flow by default. Why use it? You'd use it in case the flow thinks a net or IO boundary pin is a power net/pin, but it really isn't. This may happen since the flow adds power supplies automatically based on name matching. The flow also adds some power supplies by default, regardless of whether or not they are used in the netlist. And so you would use this property to tell the flow not to add the given power supplies. If they were added because of name matching on an IO boundary pin, then the pins in question will be considered signal pins instead of power pins after you use this property.

For example:

```
$rss{$rundir}{"supply_remove"}{"vss"} = 1;
$rss{$rundir}{"supply_remove"}{"vdd"} = 1;
```

The example above causes the flow to remove the "vss" and "vdd" power supplies that the flow would normally create for you.

Let's say you had the following in your control file. What would happen?

```
$rss{$rundir}{"supply_remove"}{"vss"} = 1;
$rss{$rundir}{"supply_remove"}{"vdd"} = 1;
$rss{$rundir}{"supply"}{"vss"} = 0;
$rss{$rundir}{"supply"}{"vdd"} = "vcc";
```

In the example above, even though the "vss" and "vdd" power supplies are defined explicitly, the supply_remove property will remove them. The supply_remove property is always applied last, regardless of the order within your control file. So the result is that no "vss" or "vdd" power supplies will be created.

What happens if you want to remove a global power supply? In this case, just specify the name of the power supply with or without the exclamation point in the name. It works either way. The following two lines each accomplish the exact same thing, so you only have to use one of them:

```
$rss{$rundir}{"supply_remove"}{"vss!"} = 1;
$rss{$rundir}{"supply_remove"}{"vss"} = 1;
```

NOTE: No wildcards or regular expressions are allowed here.

See also supply.

tcheck_abort This property can be used to specify that if you have a tcheck violation during the simulation, then the program should abort immediately rather than wait until the entire simulation has finished. The syntax is:

```
$rss{$rundir}{"tcheck_abort"}{$check_name} = 1;
```

For example:

```
$rss{$rundir}{"insert"} =
   ".tcheck tran mycheck edge ph1 r eph1 r 110e-12 130e-12
                vlth='vcc/2' vhth='vcc/2' trigger=1";
$rss{$rundir}{"tcheck_abort"}{"mycheck"} = 1;
```

The example above creates a new tcheck statement which is inserted into the SPICE file. The tcheck will make sure that the clock delay from eph1 to ph1 will be between 110 and 130 picoseconds. If it turns out that the delay is outside of that limit, then there will be a tcheck violation appearing in the logs. But the example then goes on to use the tcheck_abort property to tell the flow that as soon as it sees any violations for this tcheck in the log file, it should abort the simulation.

You can use this property with any tcheck name, including the ones the flow creates (not just the ones you insert into the SPICE file).

Instead of the literal name of the tcheck, you can also use regular expressions. For example, if you wanted to abort on the first tcheck violation for any given tcheck, you'd do:

```
$rss{$rundir}{"tcheck_abort"}{"*"} = 1;
```

Or maybe you only care about the tchecks you created manually. Suppose you created tchecks beginning with the string, "my_check". Then you'd do:

```
$rss{$rundir}{"tcheck_abort"}{"^my_check"} = 1;
```

tcheck_ignore This property lets you tell the flow to ignore specific tcheck results.

Recall that there are three ways that ".tcheck" commands get into your simulation: automatically from the timing delays (unless delay_check is set to 0), through the use of a custom spice_include file, or by using the insert property.

You may want the flow to ignore the results of one or more of those tcheck commands. This is particularly useful if you're using tcheck command as a way of measuring timing values, for example. The usage of this property is as follows:

```
$rss{$rundir}{"tcheck_ignore"}{$check_name} = 1;
```

Where $check_name is the name of the check you want to ignore. Setting it equal to 1 tells it that you do want to ignore it. Setting it equal to 0 would tell it that you do not want to ignore it (which is the default for everything).

For example:

```
$rss{$rundir}{"tcheck_ignore"}{"checkdelay_ph1_r_q_r"} = 1;
```

The $check_name can not be a regular expression at this time, so you must specify each check you want to ignore individually.

NOTE: Using this property will not actually turn off the tcheck command for the given check. The check will still occur. However, if it fails, it won't be counted by the flow. So you'll still be able to see all of the fail results in the "main.chk.gz" file.

tcheck_morph Normally, the flow will generate a series of ".tcheck" commands if you have delay arcs in your timing model. These commands will cause hsim to check that your delays are within the minimum and maximum range as specified in the timing model. But what if your delays are from the eph1 clock pin to some output pin, yet you want to measure delays instead from the ph1 clock net to that output pin? In this case, you can use the tcheck_morph property.

Here's an example. Suppose the flow originally generates the following .tcheck statement in your main.spi file:

```
.tcheck checkdelay_eph1_r_dout9_f edge dout9 f eph1 r
+ 3.00000e-10 7.62000e-10 vlth=myvlth vhth=myvhth
+ trigger=1
```

Now, suppose that you wanted to change the ".tcheck" statement above to be referenced to the "ph1" clock net instead of "eph1". And you have, say, 80ps of insertion delay from eph1 to ph1. You can achieve this by adding the following into your control file:

```
$rss{$rundir}{"tcheck_morph"}{"eph1 ph1 -80 -80"} = 1;
```

The property above will cause the flow to generate this statement instead:

```
.tcheck checkdelay_ph1_r_dout9_f edge dout9 f ph1 r
+ 2.20000e-10 6.82000e-10 vlth=myvlth vhth=myvhth
+ trigger=1
```

Notice that the "eph1" signal has been changed to "ph1". Also notice that 80ps have been subtracted from the min and max delay numbers.

The syntax of the tcheck_morph property is as follows:

```
$rss{$rundir}{"tcheck_morph"}{"$s1 $s2 $s3 $s4"} = 1;

Where $s1 is the name of the original input net/pin,
      $s2 if the name you want to change it to,
$s3 is the value you want to add to the
```

min delay value,
$s4 is the value you want to add to the
max delay value.

By the way, you can have multiple tcheck_morph properties if you want. But most people will probably only use one to morph eph1 delays into ph1 delays.

You can also give the same net name for both $s1 and $s2. You'd do that if you just wanted to add or subtract some amount from all of the tcheck statements rather than changing the the name of the input pin as well.

tcheck_remove This property deletes specific timing checks. The timing checks will not appear in the main spice file at all.

Usage:

```
$rss{$rundir}{"tcheck_remove"}{$check_name} = 1;
```

Where $check_name is the name of the check you want to ignore. Setting it equal to 1 tells it that you do want to remove it. Setting it equal to 0 would tell it that you do not want to remove it (which is the default for everything).

For example:

```
$rss{$rundir}{"tcheck_remove"}{"checkdelay_ph1_r_q_r"} = 1;
```

The $check_name can not be a regular expression at this time, so you must specify each check you want to ignore individually.

This check differs from tcheck_ignore in that it will actually remove the timing check completely. Whereas, tcheck_ignore will allow the timing check to occur but just ignores its results during post-processing.

But why would you ever want to do this? You may want to remove some timing checks but not others. You could set delay_check to 0 to remove all timing checks, but that's not going to help you if you just want to remove some but not all.

TIP: This property is very useful when using the XA spice simulator. Currently XA has a bug in it which doesn't let you have more than 100 timing check statements. So you can remove all but 100 timing checks and just run the ones you're interested in.

See also: tcheck_ignore, delay_check, delay_check_xa_bug.

tech_updates_file This property lets you specify the location of the technology updates file. The "updates" file is typically maintained by Twila. She changes it to reflect the dates when the SPICE technology has changed in some way. This lets the Random Spice Simulation flow know when it needs to consider all netlists as "modified" for the purpose of prioritizing jobs.

The "updates" file simply contains a list of keywords followed by a datestamp. Each keyword represents some aspect of the technology information, and each datestamp represents when it was updated.

Here's an example "updates" file:

```
* file/dir        mm/dd/yr time      $ comments

SPICE_dir         07/08/01 18:58    $ switched dir v.5 to v0.02
SPICE_model       07/03/01 18:11    $ v0.02 model
SPICE_para        06/14/01 18:33    $
SPICE_case        07/03/01 13:47    $ maxnioff
SPICE_sram        08/16/01 22:43    $ mods to geometries sizes

ARCADIA_dir       06/11/01 09:56    $ switched dir to lowk 9lm
ARCADIA_atf       08/21/01 16:57    $ date of file
ARCADIA_lvs       07/23/01 16:19    $ date of file
ARCADIA_proc      06/09/01 06:15    $ date of file

PEARL_dir         06/11/01 10:10    $ date of directory
PEARL_tech        07/26/01 17:56    $ date of file
```

For our purposes, we typically only care about the "SPICE*" keywords, but you can tell it which ones you care about by using the tech_updates_keywords property.

By default, the flow looks for the "updates" file at:

```
$CHIP/technology/updates
```

By using this property, you can specify your own updates file if you'd like. For example:

```
$rss{$rundir}{"tech_updates_file"} =
   "/home/users3/weigand/c5j.updates";
```

tech_updates_keywords The tech_updates_file contains a list of keywords and datestamps. The datestamps are used to determine when the SPICE technology file has been changed in such a way that would cause the RSS flow to think your netlist has changed since the previous run. Remember that when the flow thinks your netlist has changed, it wipes out all of the previous failing results in the archives directory and starts over from scratch, resetting all the "running total" ("EVER") values to zero in the summary report. And it prioritizes jobs to run sooner if it sees the netlist has changed vs. just a change in test vectors.

By default, the keywords it looks for are: SPICE_dir, SPICE_model, SPICE_para, and SPICE_case. You can override it to look for a different set of keywords if you'd like. For example:

```
@{$rss{$rundir}{"tech_updates_keywords"}} =
  ("SPICE_model","SPICE_para","SPICE_sram");
```

The complete list of keywords is: SPICE_dir, SPICE_model, SPICE_para, SPICE_case, SPICE_sram, ARCADIA_dir, ARCADIA_atf, ARCADIA_lvs, ARCADIA_proc, PEARL_dir, and PEARL_tech. But you probably only want to use the "SPICE*" keywords.

timing_model By default, it will choose between the following timing model files (in this order):

```
$CHIP/lib/dclib/${eckt_cellname}.lib
$CHIP/lib/ntlib/${eckt_cellname}.lib
$CHIP/lib/mlib/${eckt_cellname}.lib
$CHIP/lib/time/${eckt_cellname}.time
```

If a Liberty timing model ("*.lib") is found, it is used. Otherwise it looks for the PTM timing model ("*.time") instead. If more than one Liberty timing model is found (if one was found in both the dclib and mlib directories), it uses the dclib one.

You can, however, specify the timing model file explicitly using this timing_model property. You don't need to choose between those files. You can use any file you want. Give it the full path to the timing model file. The filename must end in ".lib" for Liberty models and ".time" for PTM models.

Example:

```
$rss{$rundir}{"timing_model"} = "$CHIP/lib/mlib2/$cell.lib";
```

The timing_model_type property is used if you don't specify the timing_model property. Otherwise it is ignored.

Note that the timing model's cellname must be the same as what the cell property is set to.

TLF timing models are still supported by this property. The file must end in ".tlf". But nobody should be using this format.

timing_model_type Set to either "PTM", "LIB", "both", or "none". This allows you to search for either a PTM model, a Liberty model, both PTM and Liberty models, or no timing model. By default, if it's blank it gets set to "both". It's used when searching for a default timing model to use.

If it's set to "both" (or you leave it unset), it searches the following directories (in this order):

```
$CHIP/lib/dclib/${eckt_cellname}.lib
$CHIP/lib/ntlib/${eckt_cellname}.lib
$CHIP/lib/mlib/${eckt_cellname}.lib
$CHIP/lib/time/${eckt_cellname}.time
```

If a Liberty timing model ("*.lib") is found, it is used. Otherwise it looks for the PTM timing model

("*.time") instead. If more than one Liberty timing model is found (if one was found in both the dclib and mlib directories), it uses the dclib one.

If it's set to "PTM", it only searches for:

$CHIP/lib/time/${eckt_cellname}.time

If it's set to "LIB", it only searches for:

$CHIP/lib/dclib/${eckt_cellname}.lib
$CHIP/lib/ntlib/${eckt_cellname}.lib
$CHIP/lib/mlib/${eckt_cellname}.lib

If it's set to "none", then no timing model will be used. It won't search at all for any timing models. In this case, you would specify all timing information manually using all of the various RSS flow timing related properties.

Example:

$rss{$rundir}{"timing_model_type"} = "LIB";

If you use the timing_model property, it will use that timing model instead of searching for one. You don't need to set the timing_model_type property if you use the timing_model property.

NOTE: Support for TLF timing models is no longer supported by the timing_model_type property, but you can set your timing_model property to point to a TLF model if you want. Nobody should be using TLF models at this point.

tv_diff_cmd The behavioral model verilog simulation is used to create the expected test vector outputs. But there's also a circuit-level verilog simulation which is done to determine the list of expected outputs. The two runs should produce the same output (except maybe for some X's in the output), but in some cases the two runs may differ. If there are differences, it might indicate that the circuit doesn't match the behavioral level model.

This command is run to compare the two test-vector files. If there are no differences that are cared about, it should exit with status code 0. Otherwise it should exit with a non-zero exit status code to cause the flow to abort.

By default, the flow will run the following command:

/vlsi/cad2/bin/rss_flow_tv_diff.pl

The inputs are hard-coded in the script: "${eckt_cellname}_test.tv" and "${eckt_cellname}_test_test.tv". Command line options can be included.

It will log stdout and stderr to the following file:

./tvdiff.log.gz

tv2sim_cmd This allows you to specify an alternative command for "tv2adm" which is used to convert the "runv" generated test-vectors (from the runv_bhv_cmd command) to HSIM vector format. It combines the test-vector header file along with the verilog test-vector file to produce this HSIM test-vector file. If you don't specify this, it defaults to:

```
/vlsi/cad2/bin/tv2adm \
  -n \
  -u ${eckt_cellname}.vector_header \
  ${eckt_cellname}_test.tv \
  > ${eckt_cellname}.vectors
```

The requirement is that it produces this output file in the working directory:

```
./${eckt_cellname}.vectors
```

The file must end with a line that looks like this:

```
; There were 1600 vectors
```

The number of vectors should be indicated by the line above. The number of vectors actually determines how long the simulation will run, so it has to be accurate. Each vector is usually a half a clock period in duration.

It will log stdout and stderr to the following log file:

```
./${eckt_cellname}.vectors.log.gz
```

If your script exits with any status code other than 0, it will cause the flow to abort.

tv2vrlg_cmd1 This allows you to specify an alternative command for "tv2vrlg" which is used to convert the test vectors (generated from either the random_vec_cmd script or the custom_vec_cmd script) into verilog test-bench code. If you don't specify this, it defaults to:

```
/vlsi/cad2/bin/tv2vrlg \
  -i ${eckt_cellname}.tv \
  -m ${eckt_cellname}
```

The requirement is that it produces these output files in the working directory:

```
./${eckt_cellname}_test.v
./${eckt_cellname}_data
```

All stdout and stderr streams will be sent to the following file:

./${eckt_cellname}_test.v.log.gz

If your script exits with any status code other than 0, it will cause the flow to abort.

tv2vrlg_cmd2 This allows you to specify an alternative command for "tv2vrlg" which is used to convert the test vectors (generated from the runv_bhv_cmd command) into verilog test-bench code. If you don't specify this, it defaults to:

```
/vlsi/cad2/bin/tv2vrlg \
   -c \
   -i ${eckt_cellname}_test.tv \
   -m ${eckt_cellname} \
   -l
```

The requirement is that it produces these output files in the working directory:

./${eckt_cellname}_test_test.v
./${eckt_cellname}_test_data

All stdout and stderr streams will be sent to the following file:

./${eckt_cellname}_test_test.v.log.gz

If your script exits with any status code other than 0, it will cause the flow to abort.

vec_cmd Specify the name of the script that generates the test vectors. This must be at the same level as the ECKT file. So the cell you are targeting is eckt_cellname. This file must be executable and can be in any language you want. For example:

```
$rss{$rundir}{"vec_cmd"} =
   "$CHIP/random_spice/user/weigand/rvec.pl";
```

Your script must produce output to stdout (the screen), but that output (stdout) will be redirected to a file in the current working directory:

./${eckt_cellname}.tv

The stderr output will be redirected to the following file:

./${eckt_cellname}.tv.errors

It can read any number of input files that you like. Most likely you'll want to read a ".tvhead" file and output it to stdout before anything else. Otherwise you can hard-code your ".tvhead" information into the script itself. Just remember to change the script whenever the boundary pins for your cell have been updated in the schematic.

If you want, you can include any number of command line arguments along with your command.

If your script exits with any status code other than 0, it will cause the flow to abort.

vechead_cmd The vector header file is generated by using a vector header template file (vechead_template). The $rssVECHEAD_CMD variable points to the script which does this. The script typically has access to all of the control file variables in addition to the vector header template file. You should never need to use a custom script, but if you do, you can point to it using this variable. By default, it is:

/vlsi/cad2/bin/rss_flow_genvechead.pl

Even though the default shown above doesn't show any command line arguments, it doesn't mean that you can't give them. If you want, you can include any number of command line arguments along with your command.

The requirement is that it produces this output file in the working directory:

./${eckt_cellname}.vector_header

Also, the vector header template file is generally set by default to one of the files in the "templates" directory.

It will send stdout and stderr to the following log file:

./${eckt_cellname}.vector_header.log.gz

If your script exits with any status code other than 0, it will cause the flow to abort.

vechead_template This is the template file which is used by the vechead_cmd script to generate the vector header file. This is file is what's at the top of an HSIM vector file. It specifies the names of the signals, the period, the slew rates, etc.

By default, it is set to:

$CHIP/random_spice/templates/vechead_template

See also: sim_type.

vector_include You can optionally include a list of vector-file lines you want inserted into the HSIM vector file. These lines will be directly inserted (there is no ".include" in the vector file format). It's really rare that you would ever need this, but it's there in case you do.

vih
vil
vhth
vlth

These properties specify how to translate logic levels "0" and "1" into voltages and vice versa. They're

used by the tcheck command for checking delays and by the vector file.

vih: Specifies the input logic "high" voltage level. This is the "level", not the "threshold". So in other words, what voltage level represents logic "1"? Only applies to vector files.

vil: Specifies the input logic "low" voltage level. This is the the "level", not the "threshold". So in other words, what voltage level represents logic "0"? Only applies to vector files.

vhth: Specifies the logic "high" voltage threshold, above which the signal is considered a "1". This applies only to output or bidirectional I/O terminals in the vector file, whereas in the tcheck statements it applies to all terminals.

vlth: Specifies the logic "low" voltage threshold, below which the signal is considered a "0". This applies only to output or bidirectional I/O terminals in the vector file, whereas in the tcheck statements it applies to all terminals.

The units are in volts.

If you don't specify these, they default to "vcc", "0", "vcc/2", and "vcc/2", respectively. The vcc parameter is assumed to be defined in the HSPICE technology file for you.

All of these properties are specified on I/O terminals. Any terminals you don't specify them for will have the default setting. Usage is:

```
$rss{$rundir}{"vih"}{$termname} = $value;
$rss{$rundir}{"vil"}{$termname} = $value;
$rss{$rundir}{"vhth"}{$termname} = $value;
$rss{$rundir}{"vlth"}{$termname} = $value;
```

The $termname can be a terminal name or a regular expression. If you use a regular expression for "vih" or "vil", then it will match only on input or bidirectional terminals. Whereas if you use a regular expression for "vhth" or "vlth", it will match on all terminals, but it will only apply to output or bidirectional terminals in the vector file whereas it will apply to all terminals in the tcheck statements.

For example:

```
# Inputs ab.* have a higher voltage level than normal...
$rss{$rundir}{"vih"}{"ab.*"} = "'1.5*vcc'";
$rss{$rundir}{"vlth"}{"ab.*"} = "'1.5*vcc/2'";
$rss{$rundir}{"vhth"}{"ab.*"} = "'1.5*vcc/2'";

# Output xb0 also has a higher voltage level...
$rss{$rundir}{"vlth"}{"xb0"} = "'1.5*vcc/2'";
$rss{$rundir}{"vhth"}{"xb0"} = "'1.5*vcc/2'";
```

The example above changes the logic "hi" level of all input terminals beginning with "ab" to '1.5*vcc'. When the vectors are applied to these input terminals, they will have a logic-1 at '1.5*vcc'. They will continue to have a logic-0 at 0 volts by default, however. And all other input terminals (if any) will have a logic-1 level at 'vcc' by default... Because we changed the logic level of all the "ab.*" inputs, we also had to adjust their threshold voltages for the tcheck statements (if any). Also, the xb0 terminal will have a logic-1 and logic-0 threshold at '1.5*vcc/2'. But the other terminals (if any) will have a logic-0 and a logic-1 threshold at the default of 'vcc/2'.

NOTE: If you don't set "vlth" when you set "vhth" for a given terminal, it will simply set "vlth" to the value you set for "vhth". And vice versa. So you really only have to set one or the other, but not both.

Also note: If you specified a supply keyword for an input terminal, you still need to use the "vih" property on the terminal. It won't do it for you.

One other thing. If you use the XA SPICE simulator, the vhth and vlth properties must be a function of vcc or just a numeric value. There should be no other parameter names, supply names, function names, or symbols in it. Only "/", "*", "+", "-", and "()" are allowed as math operators. For example, "vcc/2" is good, but "myvhth" is not. And "(vcc+0.5)/2" is good, but "vcc/sqrt(3.14)" is not. An error will result if this is violated. The default values are "vcc/2" for both, which is acceptable. In addition to this, if "vcc" is used, it must be defined in the property xa_vcc, or it must appear somewhere in the SPICE technology file (or any one of the included files or lib files inside of it). If it is defined in the SPICE technology file, it must be defined as a simple parameter name and value, such as:

```
.param vcc = 1.0
```

See also: sim_type, xa_vcc.

vnet You can use this option to specify the name of a verilog netlist file that contains the eckt_cellname cell. It is optional, but if you give it, then it will read it just for the I/O pin definitions. The verilog file can be a "stub" verilog netlist which just contains the I/O interface and nothing else. Or it can contain the guts as well (which will be ignored). If it is a stub verilog, it should have the supply0/supply1 statements present for the IO power ports, but if it doesn't the program may still identify the power ports correctly by regular expression pattern matching.

If both a timing model and this verilog file exist, the verilog I/O pins are compared against the timing model's I/O pins. Any missing or additional pins will be reported and an error message will be generated (and the flow will abort). Any pins that have a different direction will be reported as a warning (and the flow will continue to run). Only the timing model's pin definitions will be used if both a timing model and a verilog netlist file are present.

If you don't have a timing model but you do give it a verilog netlist, then no comparison will take place (obviously). In this case, the verilog I/O pin definitions will be used by the flow since there is no timing model. This allows you to use wildcarding.

The third possibility is to not give the flow a timing model or a verilog netlist. In which case you will not be able to use wildcarding and regular expressions to match on pin names. You would need to specify pin names precisely (without regular expressions). And that can be tedious. So you might want to use this vnet property to help you out. You don't have to.

waive_print_node_err Hsim will report a warning when it encounters a ".print" statement that specifies a node which can't be found:

Warning: .print: cannot find node n265

Whenever this happens, the RSS flow will actually crash. What's happening is that the Hsim simulation completes just fine but with that warning in the log file. And then the RSS flow will scan the log file after Hsim completes, will see the warning, and will escalate it to an error. So the RSS flow specifically looks for these kinds of warning messages and will simply error out, resulting in a crashed run in part 3 of the flow.

The reason why the flow does this is to make sure you're aware that you have a specific node name being probed in Hsim that doesn't exist. This usually doesn't happen on its own. It's usually some kind of a mistake by the user, like a typo or something. And it could be important. So that's why it's escalated to an error in the flow.

If you're okay with the fact that the node doesn't exist, you can waive this error by using this property:

```
$rss{$rundir}{"waive_print_node_err"} = 1;
```

When you use this property, it disables the error message and allows the flow to complete without a crash message or any errors. You'll just see the warning message in the Hsim log file, that's all.

window_start_add
window_start_set
window_start_set!
window_dur_add
window_dur_set!
window_steady_set!
window_period_set!
window_first_set!

The "output check window" (check_window command in the HSIM vector file) is calculated by the flow for each output signal. This is the start time and stop time whereby the output signal must be constant throughout. It is during this time in a simulation when the output signal is compared against the expected value from the test vector file.

The start time of the check window is calculated by finding all delay arcs from any input pins to each output pin. Each delay arc to a given output pin has a max delay value and an input pin arrival time value (see the arrival_set property) associated with it. So for each delay, the input pin arrival time is added to the max delay, and this represents a possible output check window starting time. This calculation occurs for every delay arc to a given output pin, and the maximum one is chosen as the final "calculated" check window start time value for a given output pin. The window duration is calculated arbitrarily to be the amount of time between the start time and the end of the clock period.

For just about every output signal, the output check window can be calculated from the timing model automatically. But in some cases, you might want to alter it by hand either to set values when the flow isn't able to calculate them or to add margin to the analysis. You're free to do that with these properties.

In addition to the output check window start time and duration, there are other parameters that can be controlled, each corresponding to the HSIM check_window function (see the HSIM documentation).

window_start_add: This property adds time to the window's calculated start time in units of picoseconds. Giving it a negative number would subtract time.

window_start_set: This property will set the window's default start time to the value you give it (in picoseconds relative to the start of the clock period). This is only applied if there is no way to calculate the window's start time as would be the case if there were no delay arcs to a given output pin. This seems like an unlikely scenario, but it can happen if you don't use a timing model at all since in that case there will be no delay arcs at all. So consider it a default value. If it is applicable, then it overrides the window_start_add property. Otherwise, the window_start_add will be applied, and the window_start_set property will be ignored.

window_start_set!: This property will set the window's start time to the value you give it (in picoseconds relative to the start of the clock period). Unlike the window_start_set property, this doesn't just set a "default" value. Instead, it sets the value of the window start time regardless of whether or not the flow was able to calculate one on its own. So this property takes precedence over everything else, including the window_start_set and window_start_add properties.

window_dur_add: This property adds time to the window's duration time (the width of the window in units of picoseconds).

window_dur_set!: This property sets the window's duration time. This overrides the calculated window duration time (if any). If it's used, then the window_dur_add property will not be used for the given pin.

NOTE: There is no window_dur_set property, since it's not needed. We have just a window_dur_set! property.

window_steady_set!: This property lets you choose something other than the default "3" for the "steady" parameter in the HSIM check_window function. It must be set to either 2 or 3. Any other setting will result in the RSS flow crashing, and you will be notified of the error. Set this to 3 if you want to make sure that the expected output state remains constant throughout the entire window duration. Set this to 2 if you just want to check that the output state is reached somewhere in the window but not necessarily through the entire window.

window_period_set!: This property lets you choose something other than the default clock period for the "period" parameter in the HSIM check_window function. For example, you may not want to check the output vectors once every clock cycle. Instead, you may want to check it once every two clock cycles. The units are in picoseconds.

window_first_set!: This property lets you choose something other than the default "0" for the "first_time" parameter in the HSIM check_window function. It represents the first time at which to begin checking. For example, you may want to skip the first few vectors if they're just initializing your circuit, and you don't care about the output values during that time. The units are in picoseconds.

There are some issues to keep in mind when using these properties. The order of execution is important. For the window start time, it uses the following order of events to determine its value:

o Is there a window_start_set! property for this output pin? If so, use it. If not, continue...

o Calculate the window start time using the arrival times plus the timing model delay times to this output pin. If it could be calculated, use it and then apply any window_start_add properties. If it could not be calculated, continue...

o Is there a window_start_set property for this output pin? If so, use it. If not, error out.

For the window duration time value, it uses the following order of events to determine its value:

o Is there a window_dur_set! property for this output pin? If so, use it. If not, continue...

o Go through the window start time calculation above. If the start time could be determined, then set the duration time to be the clock period minus the window start time. At this point, apply the window_dur_add property to the output pin. If the flow couldn't calculate the window start time, error out here.

If the resulting window duration value is less than or equal to 0ps, then an error will be generated and the flow will not run for this $rundir. This causes you to have to modify the control file to allow for a positive window duration value.

The syntax of these properties follows that of the arrival_set property's syntax. For example:

```
$rss{$rundir}{"window_start_add"}{"p[0-9]+"} = 20;
$rss{$rundir}{"window_dur_add"}{"p[0-9]+"} = -10;
$rss{$rundir}{"window_start_add"}{"q"} = 10;
$rss{$rundir}{"window_start_set!"}{"qb"} = 250;
$rss{$rundir}{"window_start_set"}{"qab"} = 200;
$rss{$rundir}{"window_dur_set!"}{"qab"} = 200;
$rss{$rundir}{"window_steady_set!"}{"q2"} = 2;
$rss{$rundir}{"window_first_set!"}{"q2"} = 2800;
$rss{$rundir}{"window_period_set!"}{"q2"} = 1400;
```

x_out_vectors

This property allows you to write "x" (don't care) into all output vectors when a given clock signal is 0 or 1. This overrides the test-vectors at the time they're converted into the HSIM file format, but not prior to that. So it will only affect the file:

${eckt_cellname}.vectors

It actually runs after the tv2sim_cmd command is run. So it modifies its output to create the $eckt_cellname.vectors file.

The format of this property is:

```
$rss{$rundir}{"x_out_vectors"}{$clkname} = $hi_or_low;
```

Where $clkname is the name of the clock ("eph1" for example).
Where $hi_or_low is either "^" or "v".

So for example, suppose you want the output vectors to be "x" whenever clock "eph1" is low? Here's what you'd do:

```
$rss{$rundir}{"x_out_vectors"}{"eph1"} = "v";
```

You can only give it the name of a clock signal, and you can't use wildcarding or regular expressions. It must be a specific clock signal. If you have multiple clock signals, it's possible to have multiple x_out_vectors properties (one for each clock signal).

Why would you want to do this? In some cases this is needed because your circuit may have an output transition during the second half the clock cycle, and so it may not be valid all the way through that clock cycle. This way you can have it just compare the output for half of the clock cycle and ignore the other half. And you could also extend the output check window duration using the window_dur_add property to include that part of the second half of the clock cycle when the output signal is still valid.

NOTE: It looks for the following statement in your vectors file to determine where the beginning of the actual vector output is:

```
;       BEGIN
```

So if you've used a custom command file for the tv2sim_cmd, then make sure you output that line prior to writing out the vectors. Also, make sure you don't insert comments with the semicolon anywhere inside of the vector lines, because that triggers it to stop looking for vectors. This is because you typically end your vector lines with something like:

```
; There were 600 vectors
```

WARNING: Use this property with caution. Know what you're doing before using it. You may be masking out real violations.

xa_vcc This property is used only in rare cases when it is running with the XA simulator and tells you that it can't find the vcc voltage value in the SPICE technology file. XA needs this value ahead of time, but it's typically only defined in the SPICE tech file, and XA can't use it in that form. The RSS flow's scripts have to parse the SPICE tech file for it, and it might be undefined or may be in some unusual format that the script doesn't understand. So if you get that error message, you would set this property value yourself. But it's rare that you would need it. It lets you specify the

vcc voltage, in units of volts.

Example:

```
$rss{$rundir}{"xa_vcc"} = 0.9;
```

See also: sim_type