Automated (Sub)Chip Synthesis – Using ACS at the subchip level

Paul Zimmer

Zimmer Design Services
1375 Sun Tree Drive
Roseville, CA 95661

paulzimmer@zimmerdesignservices.com

ABSTRACT

.

ACS is an excellent way to automate the basic synthesis flow and achieve good results. However, doing things from the top of the chip has a number of drawbacks, including long run times, lack of parallelism and longer time-to-update.

Using ACS on the subchip level eliminates many of these drawbacks, but requires special handling of directories and such to avoid conflicts. Also, having many "top" levels makes it worthwhile to introduce more automation and a common set of scripts shared by all subchips.

The paper will show the advantages and requirements of a multi-ACS subchip synthesis methodology as well as detailing some useful scripts and techniques to automate the flow.

Although not intended as an ACS primer, many of the scripts and techniques are also applicable to a top-level ACS run and can provide a starting point for new ACS users.

Please note: the scripts have been updated since this paper was written. The general flow is the same, but I haven't updated all of the example code in the paper to match the latest scripts.

Table of contents

1 Introduction

1.1 What is ACS?

ACS ("Automated Chip Synthesis") is a set of commands built in to Synopsys DesignCompiler ("DC") that help to automate the synthesis process by providing automatic script and constraint file generation based on characterizing the chip logic at various stages of the synthesis process.

ACS produces very good results when compared with other approaches, while requiring much less "grunt" work from the synthesis engineer, as discussed in (5). This paper will focus not on the "why" of using ACS, but on the "how".

1.2 Why run ACS at the subchip level?

ACS was originally designed to allow the user to do a fully automated synthesis from the top level. Why would we want to run it at the subchip level instead?

There are several reasons:

- Runtime. Running at the subchip level completely avoids the top-level characterization step. This step is typically very time-consuming and adds little value on a big chip – DC has no physical information so estimates that it makes on loading and so forth are likely to be way off. The individual module compiles (which can be run in parallel) cannot start until the characterization finishes. Running ACS on the subchip level allows the parallel compilation of the subchips to start much sooner.
- Subchip inputs and outputs are generally registered. This makes it easy to write the constraint files. In fact, we can do a single generic constraint file that all subchips can use.
- Quicker updates. You can re-run only the subchips whose code changed, without touching the other subchips, and still get the value of a full ACS run inside the modified subchip (unlike the ACS update feature).
- Synthesis can make good progress before the chip is stitched together. Since each subchip is its own ACS run, the synthesis engineer can work on the subchips as they become available without waiting for a stitched top.
- The chip may be too big (require too much memory) to run ACS from the top.

2   A quick ACS primer

2.1 The standard ACS flow

The typical ACS flow looks something like this:

1. ACS reads in all the hdl and the top level constraints.
2. ACS breaks the design into compile "partitions" (which could include all the modules in the design), characterizes these partitions and generates script and constraint files in a new subdirectory (usually called "pass0").
3. ACS creates a makefile for the design, also in the new subdirectory.
4. Make is run (either from within DC or from the outside) and the partitions are compiled using the script and constraint files generated earlier.
5. If desired, ACS can then do a "recompile". To do this, it reads in the output db files from pass0, and uses these to better characterize the design. The new script and constraint files are put in a new subdirectory (usually called pass1). These scripts read in the ORIGINAL hdl source (actually, the db files from the original source, but the effect is the same) as a starting point, not the outputted db files from pass0.
6. Make is run (either from within DC or from the outside) and the partitions are compiled using the script and constraint files generated earlier (constraints based on pass0 gate results but starting from the uncompiled hdl).
7. If desired, ACS can then do a "refine". To do this, it reads in the output db files from pass1, and uses these to better characterize the design. The new script and constraint files are put in a new subdirectory (usually called pass2). These scripts read in the outputted db files from pass1 as a starting point, rather than the hdl source.
8. Make is run (either from within DC or from the outside) and the partitions are compiled using the script and constraint files generated earlier (constraints based on pass1 and starting with the pass1 gate structure rather than hdl).

Steps 7 and 8 can be repeated indefinitely if desired, using the previous pass gate structure as a starting point.

There are many controls and options along the way. But this is the basic flow.

2.2 ACS directories and file naming

The default ACS directory structure looks something like this:

```
                                    synth root
        ┌──────────┬────────────────────────┬─────────────────┐
        │          │                         │                 │
      elab        pass                     script          constraint
              ┌─────┬──────┬─────┐      ┌──────┬──────┐    ┌──────┬──────┐
         constraints script  ...   d   pass  pass   ...  pass  pass   ...
                           ┌──────────┐
                      pre_compile  post_compile
```
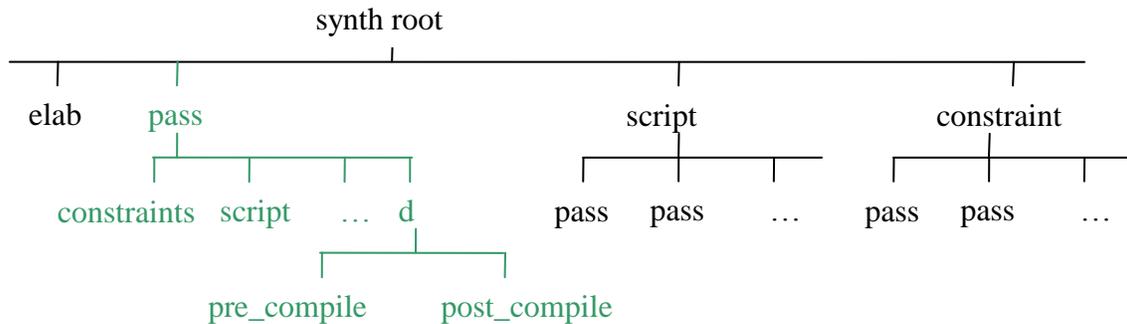
Notice that the "pass0" directory (and other pass directories) occur in multiple places, as do the "scripts" and "constraints" directories. This can be a bit confusing – what's the difference between pass0/scripts and scripts/pass0? The root level pass0 directory (in green) is where ACS puts the script, constraint, and makefiles that it generates, and where the compile puts the results. This is a transient directory and gets re-created each time ACS is run. The pass0 directories under synth_root/scripts and synth_root/constraints are where ACS looks for specially-named script and constraint files to override the default operations (more on this later). These are permanent directories created by the user that don't change for each ACS run.

Also, ACS uses all sorts of suffixes for files that have nothing whatsoever to do with the *language type* of the file. For example, ".autoscr", ".scr", ".default", ".compile", ".report" (and others) all refer to tcl files. This is a real hassle for anyone using a color syntax highlighting editor, because you want the editor to figure out the language based on the suffix.

Fortunately, ACS gives you a way to change this stuff. You have to redefine some magic variables ***in the .synopsys_dc.setup file***. You *cannot* do this after the .synopsys_dc.setup file has been read!

For example, I use the suffix ".dctl" to denote all DC tcl files (I wrote my own color syntaxing code for the editor "vim" – it knows about DC-specific commands as well as standard tcl. You can get it at my web site – www.zimmerdesignservices.com, at deepchip (www.deepchip.com) or at the vimonline site – vim.sourceforge.net). So, I have the following in my .synopsys_dc.setup file:

```
# Change suffixes so vim can syntax them.
set acs_override_script_suffix "dctl"
set acs_constraint_file_suffix "const.dctl"
set acs_budgeted_ctsr_suffix "const.dctl"
set acs_compile_script_suffix "autoscr.dctl"
set acs_log_file_suffix "synlog"
# older dc versions (2002.5 and below) have a bug that makes these not work:
set acs_global_user_compile_strategy_script "compile"
set acs_user_compile_strategy_script_suffix "strategy.dctl"
```

3   My ACS flow

3.1 Hierarchical or flat?

ACS clearly needs a certain amount of hierarchy. At a minimum, the pass-related files need to be separated from one another. Beyond this, however, I prefer to keep my synthesis directory as flat as possible. I find this makes grepping, perling, RCS updating, and sharing common scripts much easier. It has the downside, of course, of creating potentially large "ls" output, but I don't usually just "ls" the whole directory anyway.   So, I normally keep all the hdl source files, dc-tcl scripts, etc in the synthesis root directory.

Since the synthesis engineer frequently works on a stable snapshot of the verilog code, it is usually pretty easy to keep synthesis flat, even if the overall project directory structure is not flat.

This raises a new issue when running ACS at the subchip level. Should I create a different "root" directory for each subchip, and a master top directory above them all for the final read and stitch operations, or should I keep things flat, and differentiate the various pass directories some other way?

The first time through, I tried it with different synthesis root directories per subchip. But I found I ran into the same grep, perl, RCS, and sharing problems I have found with hierarchical synthesis directory structures in general. So, I flattened things in the end.

This paper will, therefore, be based on a single flattened synthesis root directory, with hierarchy only as required by ACS. Since the names must be unique, I will have to use ACS's directory control features to use non-default names for these subdirectories.

Much of what I will discuss, however, is independent of whether the synthesis environment is flattened or not. It is fairly easy to apply these techniques in whatever directory structure the reader prefers. I see advantages and disadvantages in each approach and the final choice really depends on the personal preferences of the synthesis engineer.

3.2 My directory structure

ACS's default directory structure is basically flat, which is what I want. The only real change that I had to make to accommodate multiple subchips running ACS independently was to change all of the "passN" directories to something like "passN_<subchipname>". Also, I keep those special override scripts that go in scripts/passN and constraints/passN in the root directory, and link or copy them as the scripts run, as we shall see.

Let's introduce my example chip. I call it "dogbert" (because dogbert is the supreme ruler). It has two subchips – evilceo and catberthr. Catberthr has no hierarchy under it (Catbert doesn't need his own employees – he has the whole company to torture!). Evilceo has 2 submodules – phb (pointy-haired boss) and bungieboss.

Phb has 3 submodules – dilbert, wally, and asok. Bungieboss has 2 submodules – alice and asok. Note that poor asok is multiply-instantiated – he works for everyone.

```
                        dogbert
          ┌────────────────┴────────────────┐
      catberthr                          evilceo
                              ┌─────────────┴─────────────┐
                            phb                       bungieboss
                    ┌────────┼────────┐            ┌──────┴──────┐
                 dilbert  wally     asok         alice         asok
```
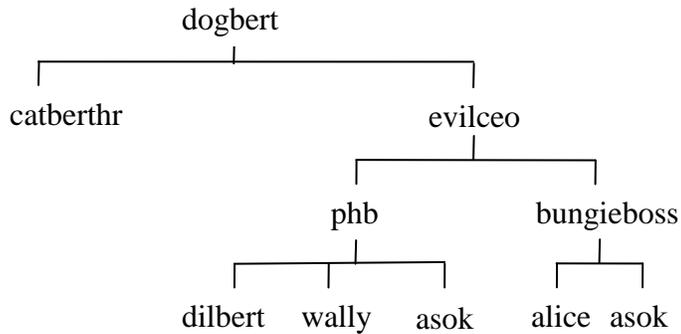
So, I end up with a directory structure that looks like this (showing only the evilceo dirs):

```
                              synth root
     ┌──────────┬──────────────────────────────┬───────────────────────┬───────────────────┐
   elab    pass0_evilceo                                    scripts                  constraint
          ┌───────┬───────┐                          ┌────────┬────────┐       ┌────────┬────────┐
   constraints  script  …  d                       pass0    pass1   …      pass0    pass1   …
                   ┌──────┴──────┐                 _evilceo _evilceo       _evilceo _evilceo
              pre_compile   post_compile
```
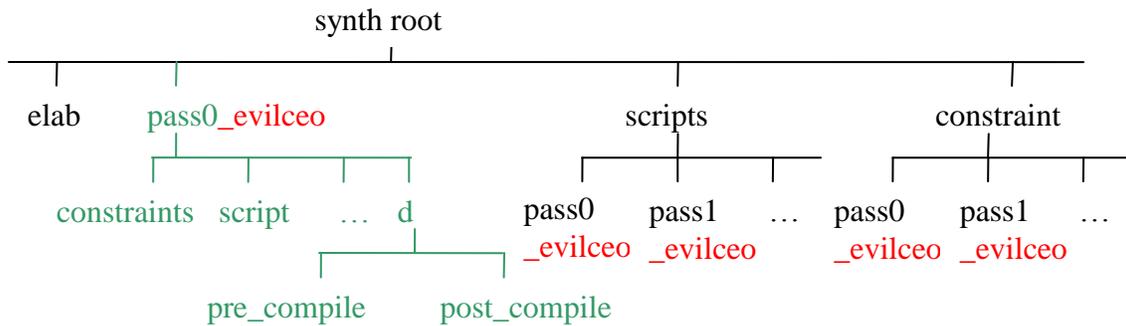
There are, of course, similar directories for catberthr.

3.3 Shared scripts

One of the advantages of running ACS from the top is that you only need to create one script and one constraint file. Running ACS at the subchip level introduces the need for multiple script and constraint files, but we can offset this by collecting most of the actual code in a few shared "common" files, and calling these with variables set to identify subchip-specific information. .

The common scripts are:
- run_acs.pl – main control script
- acs_common.dctl – has the basic acs flow. Looks at the pass number and does the appropriate acs commands.
- compile.strategy.dctl – shared compile script. See hooks below.
- common.const.dctl – shared constraint code
- common_procs.dctl – shared tcl procs (subroutines)
- top.info.pl – any chip-specific info that run_acs.pl needs. This includes script and constraint files that need to be copied, as well as any special compile commands for any of the partitions anywhere in the chip. If desired, virtually everything can be controlled from this script.

3.4 Hooks

ACS generates all the script and constraint files itself. This is handy for getting started, but I quickly found that there are things that I want to do in some or all of the scripts that ACS doesn't normally do. Fortunately, ACS provides a number of features which allow the user to override the default behavior. I use two of these features to provide a pretty flexible flow without losing the automation that ACS offers.

First and foremost, I use the "user compile strategy" feature in ACS to allow me to gain control of the compile script before any compiling or writing has actually been done. Using this feature, ACS will call *my* special script instead of doing the compile. This allows me to jump in and fix anything I want, do the compile the way I want, do any ungrouping, recompiling, etc, and finally do any writes and reports before returning to the ACS-generated compile script.

The feature is simple to invoke – you just make sure that your target compile script is in the magic directory "scripts/<passdirname>" with the magic name "default.compile". In theory, you can change the magic name, but this was broken up through version 2002.05. That's ok, I can still name the file "compile.strategy.dctl", and I just copy or link it to "default.compile" from the control script (coming soon).

There's another feature I want to work around. ACS will special-case the top and do a "compile –top" even if the default.compile script is in the magic place. You can only override this with a "*subchip*.compile" file. I want to override it, so the control script copies compile.strategy.dctl to that as well.

For example, the dogbert chip with 2 subchips – evilceo and catberhr. Let's say that catberthr does passes 0 and 1, and evilceo does passes 0 and 2. So, the control script will copy or link compile.strategy.dctl to:

```
scripts/pass0_catberthr/default.compile
scripts/pass0_catberthr/catberthr.compile

scripts/pass1_catberthr/default.compile
scripts/pass1_catberthr/catberthr.compile

scripts/pass0_evilceo/default.compile
scripts/pass0_evilceo/evilceo.compile

scripts/pass2_evilceo/default.compile
scripts/pass2_evilceo/evilceo.compile
```

I also use the user override constraint file feature. This gives me the ability to mess with the constraints after the ACS-generated constraint file has been read in. I used this a lot early on, but it has fallen out of favor – I have either found better ways of accomplishing the same thing, or simply put the feature into the compile strategy script. Still, the hook is there.

One final "hook" I use involves the control script (coming *real* soon). By doing the make from the system level instead of having ACS do it automatically, I can get in and do arbitrary changes to the script, constraint, and makefiles before anything is executed. Also, it allows me to modify the make command itself to give big modules compile priority, as we shall see.

I find that these 3 hooks give me nearly complete control over the ACS run, while retaining all the automation inherent in ACS.

4   The control script

The basic ACS flow usually looks something like this:

For each pass:

1. Create (or clean out) directories, copy (or link) override files and generally set the stage.
2. Run DC on the "top" level ACS script (which calls acs_common.dctl to do all the heavy lifting).
3. Fix anything that needs fixing before make is run.
4. Run make.

Starting out with this, most users go through the familiar process. You begin running these things from the command line. Then you tire of this and stick them in a shell script. Then the shell script gets messy, needs options, etc and you realize you need something more sophisticated.

At this point, you have a couple of choices. ACS will allow you to do the entire operation from within DC. This is handy, as it keeps everything in one place and in one language. It has a couple of disadvantages, however. First, it consumes a DC license the whole time, which then isn't available to do real compiles. Second, it requires you to write the whole thing in tcl, which isn't the handiest language for file system and file manipulation.

Instead, I chose to write the control script in perl.

4.1 Issue – passing information

When you start trying to automate ACS runs, you immediately hit a snag. You have all this information in the control script, but when the make runs DC to do the compiles, the information isn't there. You need some way to pass this information.

The mechanism I came up with is shell environment variables. The control script sets the shell environment variables (using the prefix "_acs_") before calling the scripts that run ACS and before calling make, like this:

$ENV{"_acs_top"} = $_acs_top;

Since these calls create child processes in unix, the shell environment variables are passed to the scripts. They're accessible through tcl directly using the "getenv" command, but this is clumsy. Instead, I wrote a generic piece of tcl code that imports these variables using their simple names. I call this script "getvars.dctl" and it is sourced in .synopsys_dc.setup:

```
echo "Start of getvars"
suppress_message CMD-041

foreach _var [array names env] {
  if {[string match _acs* [string tolower $_var]]} {
    set _tcl_var [string tolower $_var]
    eval "set $_tcl_var \{[getenv $_var]\}"
    echo "  getvars setting $_tcl_var to \{[eval format %s \$$_tcl_var]\}"
  }
}
unsuppress_message CMD-041
echo "End of getvars"
```

For example, if the control script creates a shell environment variable "_acs_top" with a value of "evilceo", this chunk of tcl code will create a variable "$_acs_top" with a value of "evilceo" within tcl for any child process spawned by the control script (which includes all the scripts run by gmake, because it is run by the control script).

By the way, all the case manipulaton stuff (the "string tolower…") isn't necessary on unix. I was developing the scripts on a windows machine, which is "case challenged" and had this stuff to make sure everything works. It's harmless on unix, so I left it in.

While this mechanism is very handy, it has one drawback. It means that the make command *cannot be run directly from the shell*. If it is, the shell environment variables won't be set, and the scripts will bomb out on unknown variable names.

This is easily fixed by adding a "-makeonly" option to the control script.

4.2 The control script in detail

The control script is called run_acs.pl. The full script is available (along with all the other scripts) on my website (www.zimmerdesignservices.com). I'll leave out non-essential details here to focus on the important parts.

After doing standard option parsing using Getopt, the chip info file is read. The name of the info file is specified using the "-f" option. Note that this is real perl code being sourced, and that multiple –f options are allowed. So, you could put all sorts of things into the script. But the main use is to load the top.info.pl file.

```
# Load up the chip data (typical usage is "-f <chip>.info.pl")
foreach $file (@morefiles) {
  require $file;
}
```

In addition to allowing you to set any run_acs.pl variable (in particular, the _acs_ variables), the info file has a special feature that allows you to specify command line options for a subchip in the %clargs array and have them behave as if they had been on the command line. This trick makes it easier to keep all the important information (like run_acs.pl options) in the single info file. The only arguments required on the call to run_acs.pl are –f (the info file) and –t (the name of the top module - the subchip name).

To implement this feature, the script first looks to see if there is an entry in %clargs for this subchip. If so, it sticks the entry in @ARGV and calls &process_options (which calls Getopt) again:

```
# If there is an entry for module-specific options, parse them
if (exists $clargs{$top}) {
  print "Parsing clargs from info file $clargs{$top}\n";
  @ARGV = split(/\s+/,$clargs{$top});
  &process_options;
}
```

With the command line parsed, the info file read, and any clargs-specified options parsed, the script can now set defaults for variables not set, like this:

```
# Default options if not set by process_options or -f call
$synlog = "synlog" unless ($synlog);
$_acs_dctl = "dctl" unless ($_acs_dctl);
```

and so forth . . .

The script next sets up the various shell environment variables that are pass-independent, such as the top module name, the home and work directories, etc. Remember that these will end up in the tcl scripts as tcl variables.

```
# Pass globals via environment variables.  These can be accessed in all dctl
# code.
$ENV{"_acs_top"} = $_acs_top;
$ENV{"_acs_globals_files"} = join(" ",@_acs_globals_files);
$ENV{"_acs_libdirs"} = join(" ",@_acs_libdirs);
$ENV{"_acs_home"} = $ENV{"PWD"};
$ENV{"_acs_workdir"} = "$_acs_workdir";
```

and so forth . . .

The info file will also specify any special compile commands. These are converted to shell environment variables (see the section on the compile.strategy.dctl script) by the following code:

```
# Convert special_compile_cmds hash to separate env variables
&hash2env("_acs_special_compile_cmds__",%special_compile_cmds);
```

Next, the script parses the makefile, if one was specified. The results are passed in the shell environment variable $_acs_verilog_src_files. For more details on the makefile option, see the section on makefiles.

```perl
# Parse makefile if specified.
if ($makefile) {
  require "parsemake.pm";
  $pmdb = &parsemake::parsemake($makefile);

  # Build a hash of the globals files so we can leave these out of the list
  foreach $globalsfile (@_acs_globals_files) {
    $globalsfiles{$globalsfile} = 1;
  }
  foreach $flatdep (sort
(&parsemake::get_target_deps_flat($pmdb,"${_acs_top}.db"))) {
    if (($flatdep =~ /\.v/) && !($globalsfiles{$flatdep})) {
      push(@verilog_src_files,$flatdep);
    }
  }
  $ENV{"_acs_verilog_src_files"} = join(" ",@verilog_src_files);
}
```

Note that the actual parsing of the makefile is left to a perl module called "parsemake.pm". Again, see the makefile section for details.

To allow run_acs.pl to be used to build the top, a new "noacs" option was added after this paper was originally published. When invoked, this will skip the main loop (by setting passes to an empty array) and just run DC using whatever scriptfile name was specified (which should be top.dctl, *not* acs_common.dctl!).

```perl
if ($noacs) {  # run_acs.pl is also abused to run the top
  @_acs_passes = ();  # block normal code by making no passes
  # Run the top-level script
  $dcsh= "dc_shell-t" unless $dcsh;
  $dcshcmd = "$dcsh -f $scriptfile" unless $dcshcmd;
  $logfile = "${_acs_top}.${synlog}";
  $bsublogfile = "${_acs_top}.${bsublog}";
  &rundc($dcshcmd,$logfile,$bsublogfile,$lsf);
}
```

The &rundc subroutine runs DC (via lsf if the lsf option is invoked) and will be explained below.

Now the main loop.

```perl
foreach $_acs_pass (@_acs_passes) {
  print "Doing pass $_acs_pass of \{@_acs_passes\}\n";
  $_acs_passdir = "pass${_acs_pass}_${_acs_top}";
  # Set up globals
  $ENV{"_acs_src"} = "${_acs_passdir}/db/pre_compile";
  $ENV{"_acs_dest"} = "${_acs_passdir}/db/post_compile";
  $ENV{"_acs_pass"} = $_acs_pass;
  $ENV{"_acs_prevpass"} = $_acs_prevpass;
  $ENV{"_acs_passdir"} = $_acs_passdir;

  unless ($makeonly) {
```

(Notice the rest is qualified by the makeonly flag)

Create directories and copy or link files:

```perl
    # Set up directories
    &makedir($_acs_passdir);

    # Copy scripts and constraints down to ./scripts/<passdir> and
    # ./constraints/<passdir>

    # Now do the copies
    &makedir("./scripts/$_acs_passdir");
    &copy_files(".","./scripts/$_acs_passdir",@scripts2copy);
    &makedir("./constraints/$_acs_passdir");
    &copy_files(".","./constraints/$_acs_passdir",@constraints2copy);
    # ACS now does a special-case on top modules - if left
    # to its own devices it will do a compile -top.  More importantly, it
    # IGNORES default.compile.  So, copy default.compile to <subchip>.compile
    # to fool it.
    system "cp -pf ./compile.strategy.${_acs_dctl}
./scripts/$_acs_passdir/$_acs_top.compile";
    # Workaround for bug in older versions where the renaming of the
    # default strategy script has no effect.  Copy them as well.
    if (join(" ",(@scripts2copy,@constraints2copy)) =~
/compile\.strategy\.${_acs_dctl}/) {
      print "Copying compile.strategy.${_acs_dctl} to default.compile\n";
      system "cp -pf ./compile.strategy.${_acs_dctl}
./scripts/$_acs_passdir/default.compile";
    }
```

If this is pass0, the acs_read_hdl command will execute, so we need to set up the working directory:

```perl
    if ($_acs_pass eq "0") {
      # for pass 0, make the work and elab directory structure.
acs_common.dctl
      # uses these to force junk files out of the root file system
      &makedir($ENV{"_acs_workdir"});
    }
```

Now we can actually run ACS. This is again done by &rundc.

```
    # Run the top-level script
    $dcsh= "dc_shell-t" unless $dcsh;
    $dcshcmd = "$dcsh -f $scriptfile" unless $dcshcmd;
    $logfile = "${_acs_passdir}.${synlog}";
    $bsublogfile = "${_acs_passdir}.${bsublog}";
    &rundc($dcshcmd,$logfile,$bsublogfile,$lsf);
```

Let's take a quick look at &rundc.

```perl
# rundc - run dc_shell
sub rundc {
  my($dcshcmd,$logfile,$bsublogfile,$lsf) = @_;
  if ($lsf) {
    # For lsf, create the command with the redirect to the logfile as part of
    # the command that bsub executes.  -o on bsub is used to log the lsf
    # output messages.
    # Unfortunately, there is no way to get bsub to send the output to stdout.
    # Therefore, we cannot use open(PASSLOG.. as below to watch the output.
    # It is possible to watch the output using tail, but you need a special
    # version of tail that always starts at the beginning of the file.
    # Since this gets into OS and system specific features, I won't do this.
    $top_dcshcmd = "bsub -o ${bsublogfile} -K $bsubargs_all
$partition_bsubargs{${_acs_top}} \'$dcshcmd > ${logfile}\'";
    print "\nRunning scriptfile \"$scriptfile\" using command \"$top_dcshcmd
\"\n";
    system $top_dcshcmd;
  } else {
    # Without lsf, we can just execute the command and parse for error
    # on the fly.
    open(PASSLOG,"$dcshcmd |tee ${logfile} |") || die "Couldn't open files to
run $_acs_pass";
    while (<PASSLOG>) {
      if (/Error:/) {
        print STDERR;
      }
        #if ($verbose) {print};
        print;
    }
    close(PASSLOG);
  }
}
```

If lsf is enabled, we create the bsub command, putting the lsf output into the bsublogfile via the –o switch. The –K switch keeps bsub from returning before the job is complete. The actual dcshcmd is put into quotes along with the redirect to the synlog file. Putting all of this in quotes causes bsub to treat the redirect as part of the command. This gets the output into the synlog file without involving bsub's own redirection stuff (which is a mess).

Notice that there are a couple of ways to add bsub options. You can add generic bsub options via the bsubargs_all variable (which you can set in top.info.pl). These will effect all bsub calls (including the –noacs top-level build). Partition-specific options can be specified in the %partition_bsubargs hash (again in top.info.pl). An example of using %partition_bsubargs to

add the "-N" switch (causing email to be sent) for the top level is in the sample top.info.pl script in the distribution.

If lsf is not enabled, we just run the dcshcmd (which will invoke dc_shell-t). Since the output comes to stdout, we can easily peek at it and watch for errors (which we can't do with lsf).

That's all for the "unless ($makeonly)" code.

Now, on to the make. Before we actually run the make, we have an opportunity to mess around with any files that ACS created, if necessary. The example here removes the line that sources ACS's environment file from all of the ACS-generated scripts. It also changes the "quit" to use my &exit proc (more on this later).

```perl
# modify $_acs_passdir/Makefile, acs-generated scripts, etc here...
# autoscr mods
@autoscrfiles = <${_acs_passdir}/scripts/*.${_acs_dctl}>;
foreach $autoscrfile (@autoscrfiles) {
  chomp($autoscrfile);
  ($origfile = $autoscrfile) =~ s/\.${_acs_dctl}/_orig.${_acs_dctl}/;
  system "mv $autoscrfile $origfile";
  open(ORIG,"< $origfile") || die "Could not open ${autoscrfile}_orig\n";
  open(NEW,"> $autoscrfile") || die "Could not open ${autoscrfile}\n";
  while (<ORIG>) {
    # comment out source of env file
    s/^(\s*source.*env)/# \1/;
    # change exit to use my &exit proc
    s/^\s*(exit|quit)/&exit/;
    print NEW;
  }
  close ORIG;
  close NEW;
}
```

Why would you want to get rid of the lines that source the env file? Well, the ACS-generated environment file *attempts* to reproduce the environment – meaning library search paths and such. Many of us set this stuff up in the .synopsys_dc.setup file, meaning it will be picked up anyway when the compile starts. In my environment, it will either be in the .synopsys_dc.setup file, or in my compile_env.dctl file. In either case, it will not be missed.

All of this would simply be redundant if the ACS-generated environment file accurately reproduced the environment, but it *doesn't*. One day I was adjusting some attributes on the vendor libraries in the .synopsys_dc.setup file and wondering why they were having no effect. I finally traced it to this ACS-generated environment file. This file was re-reading the library *after* I had already read it in and set the attributes, but failing to recreate the attributes. The net result was to leave the attributes unset.

After this, I added the code above to prevent this from ever happening again…

Run_acs.pl also has code to modify the ACS-generated Makefile:

```perl
  # makefile mods
  # Modifying the makefile allows us to use partition-specific dc_shell
  # commands (via %partition_dcshcmds), and support lsf with partition-
specific
  # bsub args (via %partition_bsubargs).
  $origmkfile = "${_acs_passdir}/Makefile_orig";
  $newmkfile = "${_acs_passdir}/Makefile";
  system "mv $newmkfile $origmkfile";
  open(ORIG,"< $origmkfile") || die "Could not open ${newmkfile}_orig\n";
  open(NEW,"> $newmkfile") || die "Could not open ${newmkfile}\n";
  while (<ORIG>) {
    # Modify lines like "$(DC_SHELL) -tcl_mode -f ...."
    if (($makefile_dcshcmd) = /^\s*\$\(DC_SHELL\)(.*)/) {
      # makefile_dcshcmd now has command less the DC_SHELL stuff at the
      # beginning.
      # Extract partition name
      ($partition) = /\/(\w+).autoscr.dctl/;
      # Figure out what dc_shell command to use
      if (exists $partition_dcshcmds{$partition}) {
        # If there is a special one, use it
        $partition_dcshcmd = $partition_dcshcmds{$partition}
      } else {
        # Otherwise, default
        $partition_dcshcmd = "dc_shell-t";
      }
      # Build the new command
      $new_dcshcmd = "$partition_dcshcmd $makefile_dcshcmd";
      if ($lsf) {
        $_ = "\tbsub -o \$(LOG_PATH)/${partition}.${bsublog} -K $bsubargs_all
$partition_bsubargs{${partition}} \'$new_dcshcmd\'";
      } else {
        $_ = "\t$new_dcshcmd";
      }
    }
    print NEW;
  }
  close ORIG;
  close NEW;
```

The primary objective here is to change the $(DC_SHELL) command itself. Doing this allows run_acs.pl to use the %partition_dcshcmds hash to specify different revisions of DC for different partitions. It also allows run_acs.pl to convert the Makefile to use lsf.

OK. Now we're ready to do the make. We'll build the make command first.

```perl
# Build make command
$makecmd = "gmake" .
 " -j $joblimit" .
 " -f ${passdir}/Makefile";
# Handle hotmods.  Hotmods are modules that should be given priority in
# compiles.  In make, this is accomplished by listing them explicitly on
# the command line, followed by "all".
foreach $hotmod (@hotmods) {
  $makecmd .= " $ENV{_acs_dest}/${hotmod}.db";
}
$makecmd .= " all";  # build everything else
```

This hotmod stuff requires some additional explanation. In large chips, it is not uncommon to have certain leaf modules dominate the compile time. If there are a lot of other leaf modules to be compiled, and multiple machines or cpu's are available, much wall clock time can be gained by starting these compiles first. I call these big leaf modules (or even non-leaf modules) "hotmods".

Unfortunately, ACS has no native feature to do this compile prioritization. Having moved to gmake from my own hand-written make script (in perl, of course), I had a heck of a time figuring out how to do this. I tried using perl to re-order the makefile to put the hotmods first (which is how it worked in my script – it just went through the list in order on every pass), but this didn't do it. I bought the O'Reilly "make" book and scoured it for an answer. Nothing. Finally, I broke down and called Steve Golson. Steve explained that it was all very simple. You just put the target files of the modules (<modulename>.db) *on the command line* ahead of the "all" call and gmake will prioritize them. And it works. Thanks (again) Steve.

So, that's what this code does. It takes the list of hotmods (passed on the command line using multiple –hotmod option calls or via the %clargs hash in the info file, or even set directly in the info file) and puts them at the head of the gmake command.

With the make command built, we can now execute it. If we see an error, we'll kick that out to STDERR:

```perl
# Do the make
print "\n\nRunning make for pass $_acs_pass using command:\n\"$makecmd\"\n";
unless ($nomake) {
  open(MAKE,"$makecmd |");
  while(<MAKE>) {
    if (/Error:/) {
      print STDERR;
    }
    print;
  }
  close GMAKE;
  print "Done with make for pass $_acs_pass\n";
}
```

Finally, we keep track of the previous pass, and the loop over the passes is complete:

```perl
  $_acs_prevpass = $_acs_pass
}
```

5   The acs_common.dctl script

The acs_common.dctl script is where the ACS and dc commands are actually executed to run ACS. It is called directly by run_acs.pl, or can be sourced by the subchip's ACS script file (typically <subchip_name>_acs.dctl), if the subchip has one. There are a number of variables that control the operation of acs_common.dctl:

- $_acs_top_envfile – specifies the environment file name for the ACS run. This defaults to "top_env.dctl". This file sets up any chip specific constraints.

- $_acs_top_const_file – specifies the name of the top level constraint file. run_acs.pl defaults to ${_acs_top}.const.dctl.

- $_acs_src_const_cmd – Override the source of the constraint file

- _acs_readhdl_cmd – Override for acs_read_hdl command

- _acs_ungroup_cmd – Ungroup command

- _acs_setpartitions_cmd – "set_compile_partitions" command

- _acs_recompile_cmd – Override "acs_recompile_design" command

- _acs_refine_cmd – Override for "acs_refine_design" command

There are a couple of ways that you can set these variables. Since they are _acs_ variables that will be passed by run_acs.pl, you can set them directly in the info file. This requires that you escape special characters to hide them from perl. Alternatively, you can create a subchip-specific scriptfile (typically <subchip_name>_acs.dctl) and set these variables before calling acs_common.dctl. This avoids the need to escape special characters. It may be preferable if you have to insert sophisticated code snippets into the variables (just because it is called *_cmd doesn't mean you're limited to a single command – you can list many commands separated by ";").

If you use the scriptfile approach, you have multiple ways to specify the scriptfile. You can set it in the info file (based on detecting the $_acs_top is the target subchip), you can specify it on the command line using the –scriptfile option, or you can put the –scriptfile option in the %clargs hash ("hash" is perl for associate array). As they say in perl, TMTOWTDI (There's More Than One Way To Do It).

Examples of some of this will be shown later.

5.1 The initialization code

Most of acs_common.dctl is concerned with what to do on any given pass, but there is some initial housekeeping to be done first. This initialization code looks like this:

```
# Force DC to dump all the junk files in workdir rather than in the root dir
define_design_lib DEFAULT -path $_acs_workdir
```

This command tells ACS/DC not to dump its junk all over the root directory. Reading certain files, like parameterized designs, will have (permanent!) side-effects in the file system. Without this code, these side-effect files will be put into the root directory, where they can affect later runs (besides looking messy). This command puts them in a separate subdirectory, which can then be removed on each pass by run_acs.pl. The name of the directory is set by run_acs.pl, which also creates the directory. The directory will be "work_<subchip>". Using separate work directories for each subchip is important – otherwise parallel compiles of different subchips might interfere with one another.

5.2 The pass0 code

The first thing that the pass0 code does is execute the acs_read_hdl command. This command can be overridden by setting $_acsreadhdl_cmd.

```
if {$_acs_pass == "0"} {
  &msg "Starting pass 0"

  # Do the acs read command
  # Note that the user can pre-set this, including to a no-op command (like
  # echo "Skipping acsreadhdl because it has already been done".  This allows
  # higher level scripts to do the read themselves and mess with grouping and
  # so forth while still allowing simpler subchips to use this code.
  # If you do the read in a higher level script, don't forget to do the
  # define_design_lib command above first, or DC will defecate all over the
  # root directory.
  if {[info exists _acs_readhdl_cmd]} {
    # if pre-set, use variable
    &cmd eval $_acs_readhdl_cmd
  } else {
    # otherwise default
    # if _acs_verilog_src_files is set, use it (comes from run_acs.pl parsing
makefile)
    if {[info exists _acs_verilog_src_files]} {
      set _hdl_source "$_acs_globals_files $_acs_verilog_src_files
$_acs_libdirs" \
    } else {
      set _hdl_source "$_acs_globals_files . $_acs_libdirs" \
    }
    &cmd acs_read_hdl \
      -format verilog \
      -hdl_source "$_hdl_source" \
      $_acs_top
  }
  current_design $_acs_top
  link
```

The "&cmd" proc is one of my handy procs described in the "My Favorite DC/PT Shell Tricks" paper (3). It echoes the exact command before executing it, making it much easier to follow program flow. I use it for most DC and ACS commands.

As you can see, the default value of the –hdl_source argument to acs_read_hdl (effectively, the search path) is "$_acs_globals_files", followed by "." (the current directory – remember that my directory structure is flat and the hdl source files are in the synthesis root directory), followed by "$_acs_libdirs".

What *are* all these things? The variable "$_acs_globals_file" is passed by run_acs.pl and is intended to point to a file that contains "global" settings. These are typically 'define values that are used throughout the chip. For DC to recognize and apply them correctly, they must be included *first* in the read list (in standard DC this is accomplished by doing "read –format verilog {$_globals_files <foo.v>}").

In my opinion, this is the correct way to include global defines - synthesizable code (or any code) should avoid the use of `include to fetch globals!

The variable "$_acs_libdirs" passed by run_acs.pl and is intended to point to a directory (or file) where any shared library elements can be found.

If your directory structure is hierarchical, you can just pre-set $_hdl_verilog_src_files to point to a list of files and/or directories to find the source (don't let the name fool you – it can include directories).

The next step is to source the environment file:

```
# Now that a design is loaded, source the environment file
&cmd source $_acs_top_envfile
```

The environment file is sourced *after* the hdl read operation because it may contain commands that must be applied to a *design*. And it is sourced *before* the top level constraints because it might set global values, such as clock frequencies, used by the constraint file (this provides a handy way to have a single "knob" to adjust the clock frequencies throughout the chip without editing every constraint file).

The next step is to read in the constraint file. This can be overridden by setting $_acs_src_const_cmd.

```
# Source the top level constraint file with mapped flag false (this allows
# constraint file to behave differently with mapped or unmapped logic).
# This may include forced mapping of config registers and clock flops in the
# unmapped case (to get timing constraints to propagate).
if {[info exists _acs_src_const_cmd]} {
  &cmd eval $_acs_src_const_cmd
} else {
  set _mapped 0 ; &cmd source $_acs_top_const_file
}
```

If the default read is used, a special variable "$_mapped" is set to zero first. This allows the constraint file to behave differently when it is called on a mapped design versus an unmapped design. And right after the acs_read_hdl, the design is definitely unmapped.

Now it is time to do any ungrouping required. Since this will be module-specific, it is passed in a variable to be executed (typically set by the <subchip>_acs.dctl script or in the info file). If the variable is not set, skip it.

```
# Call ungroup command if set
if {[info exists _acs_ungroup_cmd]} {
  &cmd eval $_acs_ungroup_cmd
}
current_design $_acs_top ;# just in case _acs_ungroup_cmd forgot...
```

One last thing before compiling – do the set_compile_partitions command. This tells ACS how to break the chip up for compiling. See the ACS manual for more details.

The default is to compile every module (or every module that remains after ungrouping), using the "-force –all" switches.

```
# Set partitions
if {[info exists _acs_setpartitions_cmd]} {
  # if pre-set, use variable
  &cmd eval $_acs_setpartitions_cmd
} else {
  # otherwise default to force, all
  &cmd eval set_compile_partitions -force -all
}

report_partitions
```

Finally we actually do the compile command.

```
  # Do the compile
  if {[info exists _acs_compile_cmd]} {
    # if pre-set, use variable
    &cmd eval $_acs_compile_cmd
  } else {
    &cmd acs_compile_design -destination $_acs_passdir -prepare_only -force
$_acs_top
  }

  &msg "End of pass 0"

  ;# end of pass 0 code
```

5.3 The pass1 code

The pass1 code is really quite simple. All it does is the acs_recompile_design command (or $_acs_recompile_cmd if set).

```
} elseif {$_acs_pass == "1"} {
  &msg "Starting pass 1"

  if {[info exists _acs_recompile_cmd]} {
    # if pre-set, use variable
    &cmd eval $_acs_recompile_cmd
  } else {
    &cmd acs_recompile_design -force \
      -prepare_only \
      -budget_source "pass${_acs_prevpass}_${_acs_top}" \
      -source "pass${_acs_prevpass}_${_acs_top}" \
      -destination "pass${_acs_pass}_${_acs_top}" \
      $_acs_top
  }

  &msg "End of pass 1"
  ;# end of pass 1 code
```

5.4 The pass2 (or greater) code

For pass2 (or passN where N is >=2), all we need to do is the acs_refine_design command (or $_acs_refine_cmd if set).

```
} else  {
  # pass > 1
  &msg "Starting pass $_acs_pass"

  if {[info exists _acs_refine_cmd]} {
    # if pre-set, use variable
    &cmd eval $_acs_refine_cmd
  } else {
  &cmd acs_refine_design -force \
    -prepare_only \
    -source "pass${_acs_prevpass}_${_acs_top}" \
    -destination "pass${_acs_pass}_${_acs_top}" \
    $_acs_top
  }

  &msg "End of pass $_acs_pass"
} ;# end of pass 2 code
```

6   The compile.strategy.dctl script

6.1 The special_compile_cmds hook

Recall that in my environment, I always force an override of the compile command during the DC compiles of the partitions initiated by the ACS make. That is, the script files generated by ACS will source my compile.strategy.dctl file instead of issuing the compile command.

This "hook" allows me to add any special writes, reports, checks, etc that I want to do on all partitions, but I still want to somehow leave the door open for special processing on a partition-specific basis. One way to accomplish this would be to just use ACS's own hook – put a script named <subchip.compile.dctl> in the correct scripts/pass*_<subchip> directory. And that does work in my environment (you'll need to add it to the @scripts2copy array in top.info.pl to make sure it gets used). But that would bypass my code in compile.strategy.dctl completely, and it would require me to maintain another script even if all I wanted to do was change the compile command arguments or skip the compile completely.

Instead, I provide this feature by having an array in the topinfo.pl file. This is a perl "hash" (associative array) that maps module names to special compile commands. It is called the %special_compile_cmds hash and looks like this:

```perl
%special_compile_cmds = (
  "evilceo" => "echo \"Skipping compile\"",
);
```

Note that in the real code this entry (for evilceo) is commented out, as evilceo has synthesizable code. Normally, the top level of each subchip will not have synthesizable code, and something like the above can by used.

The first time I implemented this, I used a tcl associative array, which meant I had to have two "info" files – one for perl (run_acs.pl) and one for tcl (anything that runs in DC). I wanted to have only a single file for all this information, so I had to find away to transform the perl hash into information that could be used in tcl.

My passing mechanism, as explained earlier, uses shell environment variables. But shell environment variables generally don't include associative arrays, so I "flattened" the array into distinct variables. This is done in run_acs.pl by a perl subroutine called "&hash2env":

```perl
sub hash2env {
  my ($prefix,%hash) = @_;

  foreach $key (keys %hash) {
    #print "key is $key\n";
    $ENV{"${prefix}${key}"} = $hash{$key};
  }

}
```

In the example show above, this would create an environment variable called "_acs_special_compile_cmds__evilceo" with a value of "`echo \"Skipping compile\"`". &getvars.dctl then transforms this into a tcl variable of the same name with the same value. There is also a provision for pass-specific special compile commands. More on this later.

6.2 Initialization code

The script starts by sourcing the compile-time environment file (which itself probably just sources the top-level environment file).

```
&msg "Starting the standard custom compile strategy script"

# Source the compile environment file
&default_var _acs_compile_envfile "compile_env.${_acs_dctl}"
if {[file exists $_acs_compile_envfile]} {
  source $_acs_compile_envfile
}

set _module $current_design
```

Then it does a check_design:

```
# Do lint
check_design
```

6.3 Special compile command or standard compile

Now its time to look for the special compile command and execute it. If there is none, then do the standard compile.

```
# Look up module in _special_compile_cmds array.
if {[info exists _acs_special_compile_cmds_pass${_acs_pass}__${_module}]} {
  # Pass-specific command
  #   Get the command and execute it
  eval "set _cmd \$_acs_special_compile_cmds_pass${_acs_pass}__${_module}"
  &cmd eval $_cmd
} elseif {[info exists _acs_special_compile_cmds__${_module}]} {
  # All-pass special command
  #   Get the command and execute it
  eval "set _cmd \$_acs_special_compile_cmds__${_module}"
  &cmd eval $_cmd
} else {
  # Not found, do standard compile
  &default_var _compile_cmd "compile -map_effort medium"
  &msg "No entry found for $_module in _special_compile_cmds - using default
compile process with command \"$_compile_cmd\""

  # Suppress output of check_design -summary that is run automatically by
  # compile. (Full check_design is done above).
  suppress_message LINT-30
  &cmd eval $_compile_cmd
  unsuppress_message LINT-30

  # Do any pass-specific processing
  if {$_acs_pass == "0"} {

  } elseif {$_acs_pass == "1"} {

  } elseif {$_acs_pass == "2"} {

  }
}
```

Hooks are provided for doing pass-specific processing. The code first checks for a variable called _acs_special_compile_cmds_pass<pass>__<module>. This would come from the info file hash "%special_compile_cmds_pass1", for example. If no such variable exists, it looks for a non-pass-specific special compile command in $_acs_special_compile_cmds__<module>. If this also doesn't exist, it does a standard compile.

6.4 After the compile

With the compiling complete, now's our chance to do some common processing.

First, I would like to uniquify at the top so that things will link cleanly when I put the whole chip together. But I don't want to lose ACS's feature of allowing compiles without uniquification, so I won't do this until the last pass:

```
# On final pass, uniquify from top.
```

```
if {$_acs_pass == $_acs_final_pass && $_module == $_acs_top} {
  set uniquify_naming_style $_acs_uniquify_naming_style
  &cmd uniquify -force ;# force is probably unnecessary
}
```

The next step is naming rules. I have found through painful experience that applying the naming rules anywhere other than at the top and on the last past will cause grief for ACS. Things get renamed and ACS will get lost. So:

```
# On final pass, do rename from top.  Don't do it sooner, or name mismatches
# will result.
if {$_acs_pass == $_acs_final_pass && $_module == $_acs_top} {
  # source naming_rules.dctl
}
```

The original motivation for doing rename as I went along was performance. In older versions of DC, a rename from the top could take a very long time. This improved *dramatically* in version 2003.06. If you're having trouble with long rename times and you're using 2002.05 or older, try using 2003.06.

Similar code would appear here for atpg, if your flow does test insertion during synthesis.

If this is the top, I want to write out hierarchical verilog gate netlist and db files, and do reports. If this is the top and the final pass, I want to write out the hierarchical db file as *the* db file in the synthesis root directory for later linking at the top:

```
# If this is the top, write out the verilog netlist and the hier'l db file.
if {$_module == $_acs_top} {
  write -format verilog -hier -output "${_acs_dest}/${_module}.h.psv" $_module
  write -format db -hier -output "${_acs_dest}/${_module}.h.db" $_module
}

# If this is the top AND the final pass, write out the hier'l db file as a
# ".db" file in the synth root directory to simplify top level linking.
if {$_acs_pass == $_acs_final_pass && $_module == $_acs_top} {
  write -format db -hier -output "${_acs_home}/${_module}.db" $_module
}
```

I also do reports here. The standard script generated by ACS will also do reports, but it puts the top-level reports in a different directory. To make automated analysis of the results simpler, I do the reports the same way for all partitions.

```
# If this is the top, do reports.  ACS script will also do reports, but it
will
# do them to the /reports directory, which makes writing a perl script to scan
# all the block compile results messy.  Putting them here as well keeps
# everything regular.
if {$_module == $_acs_top} {
  report_constraint
  report_area
  report_timing -nosplit -max_paths 4 -path full -flat
}

&msg "End of standard custom compile strategy script"
```

7   The common.const.dctl script

The common.const.dctl script is called by the subchip constraint files. It applies constraints to all inputs and all outputs based on pre-defined budgets. It must do this without knowing which inputs or outputs are related to which clocks. This is the old problem of "generic" constraints.

One common technique is described in (5). It involves using 3 clocks for each clock in the design – the real clock and two virtual clocks.

I described an alternative technique in ESNUG several years ago (**ESNUG 307 Item 9),** based on a suggestion made by Juergen Stallmann of Siemans. It creates only the real clock(s). The basic premise is to set input and output delays relative to each clock according to the budgets. This is farily straightforward using the "-clock" argument on set_input_delay and set_output_delay. The trick is in handling combinational paths. The set_input_delay and set_output_delay commands will result in combinational paths that are badly overconstrained, since both delays are added into the path. Juergen discovered that this can be fixed by using set_max_delay, with a value of (comb_budget + largest_set_input_delay_value + largest_set_output_delay_value).

Both techniques work. Which one is preferable? The "1-clock" technique has the advantage of making the timing reports easier to read, since the only path groups are the real clocks. The "3-clock" technique involves some rather complicated cross-clock false paths when there are multiple clocks. The 1-clock technique only requires doing false paths directly between all clocks. This makes the 1-clock technique easier to code in a multi-clock environment.

For these reasons, and because I have a lot more experience personally with the 1-clock technique, it is the 1-clock technique I have used in common.const.dctl.

For those who are interested, I have included a comparison of the two techniques in the appendix.

7.1 What happens *before* the call to common.const.dctl

Recall that common.const.dctl is sourced from the subchip's constraint file, or from the $_acs_src_const_cmd variable set in the info file. The only thing that is required before the call to common.const.dctl is creation of the clocks. I create the clocks using a proc called &create_clock. This proc creates the clock and does all the other things I usually want to do at the same time – set_clock_skew, set_drive 0, set_ideal_net, etc.

One final thing that &create_clock does is to add the clock port to a global collection called "_all_clk_ports".

7.2 Finding all_inputs_no_clks

In order to do the set_input_delay, we need to create a collection of all the inputs, but *without* the clock ports. I'd like to be able to do this using attributes, but I have not yet found a way. Instead, I rely on the global collection "_all_clk_ports" created by &create_clock:

```
# Find the non-clock ports
# &create_clock keeps a list of ports that are clocks

set _all_inputs_no_clks [remove_from_collection [all_inputs] \
  $_all_clk_ports]
```

7.3 Constraining the inputs and outputs relative to all clocks

This code is pretty straightforward. We just loop through all the clocks and apply the appropriate input and output delays (based on that clock's period and the pre-defined budgets) using the "–clock" switch on set_input_delay and set_output_delay.

We also keep track of the max input and output delay values for use in the combinational path constraints below.

```
# Do the set_input_delay and set_output_delay for each clock.  Keep track
# of longest period for use in comb path processing.
set _max_input_delay "0"
set _max_output_delay "0"
foreach_in_collection _clk [get_clocks *] {
  set _clkname [get_object_name $_clk]
  set _period [get_attribute $_clk period]
  set _input_delay [expr (1 - $_input_budget) * $_period]
  set _max_input_delay [&max $_max_input_delay $_input_delay]

  if {[sizeof_collection $_all_inputs_no_clks] != 0} {
    &cmd set_input_delay -add_delay \
      -clock $_clkname \
      $_input_delay \
      $_all_inputs_no_clks
  }

  set _output_delay [expr (1 - $_output_budget) * $_period]
  set _max_output_delay [&max $_max_output_delay $_output_delay]

  &cmd set_output_delay -add_delay \
    -clock $_clkname \
    $_output_delay \
    [all_outputs]

}
```

7.4 Constraining combinational paths

Now the trick for constraining combinational paths. Note that there ***shouldn't be any*** combinational paths through a block at this level. The combinational path code is included here so that the script can be used successfully in other DC environments and to document the basic technique.

```
# Handle comb paths
if {[sizeof_collection $_all_inputs_no_clks] != 0} {
  &cmd set_max_delay \
    [expr $_max_input_delay + $_max_output_delay + $_comb_budget] \
    -from $_all_inputs_no_clks \
    -to [all_outputs]
}
```

For completeness, I should note that this approach is not really 100% correct. In theory, one could have the max_input_delay and max_output_delay set by different clocks, in which case the above calculation would not be correct. To fix this, one would have to keep track of each max value for each clock and look for the worst pair. Since this environment will set the delays using the same set of budgets for all clocks, the code above will work and implementing the more precise code just makes things messier. So, I used the simple code above.

7.5 Setting false paths between clocks

Now we need to set false paths from every clock to/from every other clock. Although this is not appropriate for related clocks in STA analysis (see 1), I believe it *is* the correct technique for synthesis. If there is a setup or hold problem between related clock domains, it should be fixed by adjusting the clock phases in layout, not by DC.

Again, this code is pretty straightforward. The key is knowing that it needs to be done and doing it.

```
# Set false paths btw clocks
foreach_in_collection _clk [all_clocks] {
  foreach_in_collection _other_clk [remove_from_collection [all_clocks] $_clk]
{
    &cmd set_false_path -from $_clk -to $_other_clk
  }
}
```

7.6 Subchips with no clocks

It occasionally happens that you get little pieces of combinational logic that end up as tiny little subchips at core (top). In this case, there will be no clock to declare. Common.const.dctl will produce errors if there is no clock (this is deliberate – most subchips have clocks and I want to force special handling if no clocks are found) The solution is not to use common.const.dctl at all. Instead, set explicit constraints like this:

```
set_max_delay "1.0" -from [all_inputs] -to [all_outputs]
```

This can be put either in the $_acs_src_const_cmd variable via the info file, or in a <subchip>.const.dctl file.

.

8   Makefile generation and module-specific makefiles

While ACS will automatically generate a makefile for each subchip compile, there are still a couple of reasons why you might want to create your own makefiles. Before going into this in detail, I'd like to touch on the subject of automatic makefile generation in general.

8.1 Automatic makefile generation

Although makefile creation was generally a requirement before ACS, I've never found any particular tool that is in common use. Makefile generators were included in a couple of SNUG papers over the years (4)(6), but none of these seem to have caught on, and they tend to be tied up in some larger system. I suspect many people just write them by hand. I'm not one for doing things by hand.

Writing a makefile generator that works for all coding styles isn't a trivial exercise, because you have to parse Verilog in the general case. Fortunately, there is already a publicly available general-purpose Verilog parser. It is called "rvp" (for Rough Verilog Parser) and was created by Costas Calamvokis as the core engine of his "v2html" tool, which is also free. V2html, by the way, creates fantastic frames-based web pages for exploring your chip code – give it a try. I have used rvp for a number of things over the years and have found it to be robust and dependable. The groups I have worked in have had a very wide assortment of coding styles, and rvp handled them all.

Previously, I had used rvp to generate input for my own hand-written make script (which was in perl, of course). When I decided to switch to standard make (a decision I'm still wondering about), I needed to write something to output standard makefiles. I looked at the tools available on the web, and didn't like any of them. So, I decided to roll my own, using rvp.

 It turned out that Costas (the author of rvp and v2html) had already done the guts of the script as an rvp example! So, I added a bunch of switches to customize the output, and created "synth_make.pl". This is in the script distribution, along with the current version of rvp.

Synth_make.pl creates a very hard-coded makefile, designed for post-processing by other scripts, not reading by humans. It avoids abbreviations and shortcuts for this reason.

Explaining synth_make.pl in detail is beyond the scope of what I'm trying to convey in this paper, but here are the options.

```
bash-2.05b$ synth_make.pl -help
Syntax:
  synth_make.pl [options] *.v

  options:
  -help                 Show this help
  -verbose              Be Chatty
  -debug                Turn on debug messages
  -incdir "incdir"    Add "incdir" to the list of include directories
                         (can be used multiple times)
  -libext "libext"    Add "libext" to the list of library extensions
                         recognized by the parser.
                         (can be used multiple times)
  -scr "suffix"       Use "suffix" as script suffix
                         (defaults to "scr")
  -synlog "synlog"    Use "synlog" as synthesis log file suffix
                         (defaults to "synlog")
  -dcsh "command"     Use "command" as the command to build a module.
                         This will be eval'ed by perl.  Var names available
                         include:

                         ${mod}          module being built
                         ${scr}          script suffix (from -scr or default)

                         defaults to "dc_shell -f ${mod}.${scr} >
${mod}.${synlog}"
  -default_deps "dep"  Use "dep" as a default dependency for all modules.
                         This would be used for the .synopsys_dc.setup file, as
                         well as any other script files shared by all modules.
                         This will be eval'ed by perl.  Var names available
                         are the same as for -dcsh option.

  Example:
  synth_make.pl \
  -scr dctl \
  -synlog synlog \
  -dcsh '\@echo Compiling design : ${mod}\n\ttouch ${mod}.start ; dc_shell -f
${mod} > ${mod}.synlog ; touch ${mod}.done' \
  -default_dep '.synopsys_dc.setup' \
  -default_dep 'environment.dctl' \
  -default_dep 'common.dctl' \
  -default_dep 'common.const.dctl' \
  -default_dep 'do_${mod}' \
  *.v

  NOTE: This script requires rvp - the rough verilog parser.  It is available
at http://www.burbleland.com/v2html/rvp.html
```

Notice the options for redefining the dcsh command, the suffix (-scr, -synlog), and the default dependencies. Default dependencies are all those shared scripts – the "common" files, "env" files, and so forth.

Running synth_make.pl on my dogbert chip, I might use something like this:

```
synth_make.pl \
  -scr dctl \
  -synlog synlog \
  -dcsh '\@echo Compiling design : ${mod}\tdc_shell -f ${mod}.${scr} >
${mod}.${synlog}' \
  -default_dep '.synopsys_dc.setup' \
  -default_dep 'top_env.dctl' \
  -default_dep 'compile_env.dctl' \
  -default_dep 'common_procs.dctl' \
  -default_dep 'acs_common.dctl' \
  -default_dep 'getvars.dctl' \
  -default_dep 'compile.strategy.dctl' \
  -default_dep 'common.const.dctl' \
  $* \
  globals.v *.v > top.make
```

Which would produce the following output (in top.make):

```
all: dogbert.db

dogbert.db: \
        dogbert.v \
        catberthr.db \
        evilceo.db \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : dogbert       dc_shell -f dogbert.dctl >
dogbert.synlog

catberthr.db: \
        catberthr.v \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : catberthr     dc_shell -f catberthr.dctl >
catberthr.synlog

evilceo.db: \
        evilceo.v \
        phb.db \
        bungeeboss.db \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
```

```
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : evilceo        dc_shell -f evilceo.dctl >
evilceo.synlog

phb.db: \
        phb.v \
        wally.db \
        asok.db \
        dilbert.db \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : phb    dc_shell -f phb.dctl > phb.synlog

wally.db: \
        wally.v \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : wally  dc_shell -f wally.dctl > wally.synlog

asok.db: \
        asok.v \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : asok   dc_shell -f asok.dctl > asok.synlog

dilbert.db: \
        dilbert.v \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \
```

```
        @echo Compiling design : dilbert        dc_shell -f dilbert.dctl >
dilbert.synlog

bungeeboss.db: \
        bungeeboss.v \
        asok.db \
        alice.db \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : bungeeboss     dc_shell -f bungeeboss.dctl >
bungeeboss.synlog

alice.db: \
        alice.v \
        .synopsys_dc.setup \
        top_env.dctl \
        compile_env.dctl \
        common_procs.dctl \
        acs_common.dctl \
        getvars.dctl \
        compile.strategy.dctl \
        common.const.dctl \

        @echo Compiling design : alice  dc_shell -f alice.dctl > alice.synlog

DB_FILES=dogbert.db catberthr.db evilceo.db phb.db wally.db asok.db dilbert.db
bungeeboss.db alice.db

clean:
        rm -f ${DB_FILES} ${DB_FILES:db=synlog}


# These lines attempt to force gmake to not mess about with RCS...
%.dctl: ;
%.v: ;
```

A note on those last few lines. Since moving to gmake from my perl-based make script, I keep bumping into gmake's many hidden "features". One of these is to look for an RCS directory, and automatically do an update. This may be great for software builds, but it is a disaster for synthesis. Synthesis is usually done on a "snapshot" of the verilog code to disconnect evolving RTL code from synthesis effects. The directory is manually updated to match current RCS via a link to the verilog RCS directory. The last thing I want is for gmake to automatically do an update! I have asked many people and scoured the O'Reilly make book, and I can't find a way to turn this "feature" off. The code above is the closest I've gotten, but gmake *occasionally* still messes around with RCS. Email me if you know how to really, truly turn this RCS "feature" off (paulzimmer@zimmerdesignservices.com).

Anyway, this makefile is fine for standard DC, but not what we need for ACS. It needs some post-processing.

8.2 Creating a top level makefile

As soon as you get more than a few subchips, you may find you need a top level makefile as well.

But this is a special sort of a makefile, because it is "flat". You only want one level, the subchips, but you want this level to include *all* the dependencies below it. Synth_make.pl doesn't do this, but I wrote a script that processes the output of synth_make.pl to do this. It is called flatten_make. Note that it is written to read the output of synth_make.pl only, and does not support the full makefile syntax.

flatten_make.pl uses a perl module called "parsemake.pm" that I wrote for parsing (my) makefiles. This perl module is used in several places, including run_acs.pl.

flatten_make.pl has options like this:

```
bash-2.05b$ flatten_make.pl -help
Syntax:
  flatten_make.pl [options] *.v

  options:
  -help                  Show this help
  -verbose               Be Chatty
  -m "module"            Module to be flattened
                          If none specified, flatten_make.pl will default to
first level
  -nodb                  Don't include db files in the -m dependencies
  -top "top"             Top - this is built from the -m modules only
  -topcmd "topcmd"       Command to build top (syntax similar to -dcsh)
  -scr "suffix"          Use "suffix" as script suffix
                           (defaults to "scr")
  -synlog "synlog"       Use "synlog" as synthesis log file suffix
                           (defaults to "synlog")
  -default_deps "dep"   Use "dep" as a default dependency for all modules.
  -dcsh "command"        Use "command" as the command to build a module.
                           This will be eval'ed by perl.  Var names available
                           include:

                           ${mod}            module being built
                           ${scr}            script suffix (from -scr or default)

                           defaults to "dc_shell -f ${mod}.${scr} >
${mod}.${synlog}"
```

To create my top level makefile for dogbert, I need a command like:

```
flatten_make.pl \
  -scr dctl \
  -synlog synlog \
  -dcsh '\@echo Compiling design : ${mod} ;  run_acs.pl -top ${mod} -f
top.info.pl > ${mod}.${synlog}' \
  -topcmd '\@echo Compiling design : ${top} ;  run_acs.pl -top \${top} -f
top.info.pl -scriptfile top.dctl -noacs > top.${synlog}' \
  -nodb \
  $* \
  < top.make  > top.flat.make
```

Notice that I have defined the –dcsh option to run run_acs.pl with the appropriate value for –top, which for the subchips is "${mod}" (the module, or sub-chip, name). Also notice that I did not have to specify the –top and –m options. If the code is clean (there is only one "top"), synth_make.pl and flatten_make.pl will infer the correct top and all of the options.

The –topcmd option specifies what the command should be to build the real top. More on this later.

Because I want to add options, I actually embed this in a perl script called flatten_top.pl.

flatten_top.pl produces:

```
bash-2.05b$ flatten_top.pl
No -top specified - setting top to "dogbert"
No -mods specified - setting mods to "catberthr evilceo"
bash-2.05b$
```

This is the top and module inference.

If we peek at the top.flat.make file, it looks like this:

```
all: \
        dogbert.db \
        catberthr.db \
        evilceo.db \


dogbert.db: \
        catberthr.db \
        evilceo.db \

        @echo Compiling design : dogbert ;  run_acs.pl -top dogbert -f
top.info.pl -scriptfile top.dctl -noacs -globalsfile globals.v >
dogbert_top.synlog

catberthr.db: \
        ./catberthr.v \
        .synopsys_dc.setup \
        acs_common.dctl \
        common.const.dctl \
        common_procs.dctl \
```

```
        compile.strategy.dctl \
        compile_env.dctl \
        getvars.dctl \
        globals.v \
        top_env.dctl \

        @echo Compiling design : catberthr ;  run_acs.pl -top catberthr -f
top.info.pl -globalsfile globals.v > catberthr_top.synlog

evilceo.db: \
        ./alice.v \
        ./asok.v \
        ./bungeeboss.v \
        ./dilbert.v \
        ./evilceo.v \
        ./phb.v \
        ./wally.v \
        .synopsys_dc.setup \
        acs_common.dctl \
        common.const.dctl \
        common_procs.dctl \
        compile.strategy.dctl \
        compile_env.dctl \
        getvars.dctl \
        globals.v \
        top_env.dctl \

        @echo Compiling design : evilceo ;  run_acs.pl -top evilceo -f
top.info.pl -globalsfile globals.v > evilceo_top.synlog


# These lines attempt to force gmake to not mess about with RCS...
%.dctl: ;
%.v: ;
```

This is the makefile I need to drive ACS. It contains a flattened list of all dependencies of each subchip, and invokes run_acs.pl for each.

Notice that this makefile also has instructions for building the top:

```
dogbert.db: \
        catberthr.db \
        evilceo.db \

        @echo Compiling design : dogbert ;  run_acs.pl -top dogbert -f
top.info.pl -scriptfile top.dctl -noacs -globalsfile globals.v >
dogbert_top.synlog
```

The top is built by invoking run_acs.pl with the (nonsensical-sounding) –noacs switch, and setting the –scriptfile option to top.dctl. The top.dctl script will be covered later.

8.3 Using makefiles to control acs_read_hdl

There is another reason why you might want to create makefiles in a multiple subchip environment. By default, the ACS command "acs_read_hdl" reads in *all* the hdl in the specified path. For each subchip, this may involve lots of code that is unused by that subchip.

While acs_read_hdl is smart enough to only *elaborate* the required modules, it still *analyzes* (reads) them all. The extra time is usually small, but it introduces one nasty potential problem – no subchip can be compiled unless *all* the code is clean.

In a well-controlled environment, the introduction of chip code that doesn't syntax cleanly in DC should be a rare event. Most organizations have processes in place to prevent this. Most – but not all.

I was recently in the unfortunate position of doing synthesis in such an environment. Code was continually being checked into RCS that didn't syntax cleanly. Not just occasionally, but all the time. Every time this happened, *all* my subchip compiles would fail – even if the bad code wasn't in *any* of them.

To get around this, I added the –makefile option to run_acs.pl. It uses the parsemake.pm perl module used by flatten_make.pl to extract the flattened verilog source files required, then passes them to acs_common.dctl using the $_acs_verilog_src_files shell environment variable. acs_common.dctl will then use this list of files in place of "." in the acs_read_hdl command.

In addition to insulating each subchip from coding problems elsewhere, this has the nice side-effect of speeding up the acs_read_hdl command by eliminating the unnecessary analysis of unrelated code. And it's very easy to use, since it parses the top-level makefile, which has to be created for the top-level make anyway.

9 Building dogbert

Now let's get back to the example chip and see how all of this goes together. .

9.1 File mods

To build dogbert, we need to pass the clock information to acs. This can be done in several ways, but the simplest way is to edit the top.info.pl script like this:

```
if ($_acs_top eq "catberthr") {
  $_acs_src_const_cmd = " \
    &create_clock -period 8.0 -port coffeebreak ;\
    source common.const.dctl \
  ";
} elsif ($_acs_top eq "evilceo") {
  $_acs_src_const_cmd = " \
    &create_clock -period 8.0 -port coffeebreak ;\
    &create_clock -period 10.0 -port impossible_deadline ;\
    source common.const.dctl \
  ";
}
```

9.2 Running the make

Before we run the make, we need to generate the make files. This is done by the script "mk_makefiles.pl". mk_makefiles.pl runs synth_make.pl and flatten_top.pl. Because dogbert has a globals file, we need to use the –globalsfile option. Although mk_makefiles.pl will infer that dogbert is the top, it is safest to specify this using the –top switch:

```
bash-2.05b$ mk_makefiles.pl -globalsfile globals.v  -top dogbert
No -mods specified - setting mods to "catberthr evilceo"
bash-2.05b$
```

The output ("No -mods specified - setting mods to "catberthr evilceo") is the script telling us that we didn't tell it what the subchips were - so it inferred them. Notice that the output about setting top (seen in the direct invocation of flatten_top.pl earlier) is now gone, since we set the –top option explicitly.

We can now run the make:

```
gmake --jobs --makefile top.flat.make
```

```
bash-2.05b$ gmake --jobs --makefile top.flat.make
Compiling design : catberthr
Compiling design : evilceo
Compiling design : dogbert
bash-2.05b$
```

That's it!

10 Analyzing the results

Using ACS on a large chip, you really need some way to see the compile results in an organized, compact form. I have written a script for that, called analyze_acs.pl.

Through experience, I have determined that the key numbers I want to know are usually :

1. slack
2. area
3. memory usage
4. runtime

And I want to know this information for *every pass* and for *every partition* in *every subchip*.

Here's what the output of analyze_acs.pl looks like. I have limited it to pass0 and pass1 to make it fit on the page.

```
bash-2.05b$ analyze_acs.pl -pass pass0 -pass pass1
                   || pass0                       || pass1
Partition          || hours  Gbytes Kgate  slack || hours  Gbytes Kgate  slack
================================================================================
catberthr (acs)    || 0.0    0.0    N/A    N/A   || 0.0    0.0    N/A    N/A
  /catberthr       || 0.0    0.1     0     0.00  || 0.0    0.1     0     0.00
  (wall clock)     || 0.0
--------------------------------------------------------------------------------
evilceo (acs)      || 0.0    0.1    N/A    N/A   || 0.0    0.1    N/A    N/A
  /alice           || 0.0    0.1     0     N/A   || 0.0    0.1     0     N/A
  /asok            || 0.0    0.1     1     0.00  || 0.0    0.1     1     0.00
  /dilbert         || 0.0    0.1     0     4.00  || 0.0    0.1     0     4.00
  /bungeeboss      || 0.0    0.1     2     0.00  || 0.0    0.1     2     0.00
  /phb             || 0.0    0.1     2     0.00  || 0.0    0.1     2     0.00
  /wally           || 0.0    0.1     0     7.00  || 0.0    0.1     0     7.00
  /evilceo         || 0.0    0.1    16     1.00  || 0.0    0.1    16     1.00
  (wall clock)     || 0.3
```

This shows all the relevant information about all the partition compiles, as well as the runtime and memory usage for the subchip acs runs (the acs runs don't compile anything themselves, which is why the "(acs)" lines show "N/A" for Kgate and slack).

As an interesting aside, you might wonder why alice has an "N/A" in the slack column. This is because alice clocks data from one domain into another domain – and nothing else. Since the cross-clock paths are false_path'ed, this results in no constraints on alice and thus no slack report.

An earlier version of this script used files to time the synthesis runs (touching a ".start" and a ".done" file). This latest incarnation uses the unix "date" function with the "+%s" option to give an absolute timestamp. In order to make this work, it is necessary to output this timestamp when the script starts running, and when it ends. The start is easy – I put a "&msg" command in the .synoposys_dc.setup file (my &msg command includes the date output). To get the end, it is

necessary to force all the scripts to exit via my "&exit" proc, which also uses &msg. This is why run_acs.pl modifies all the autoscr.dctl files to use &exit.

By the way, I discovered a command "cputime" in dc_shell-t, but it doesn't respond to "man" or "help", and I haven't found any documentation on it. So, I continue to use the date feature of &msg/&exit.

11 Optimizing dogbert

The dogbert scripts in the release collection have some more optimizations added – primarily to illustrate how to do various things. These are all hooked in via the top.info.pl file (the simple version used earlier is in the release collection as well – it's called simple_top.info.pl).

Let's take a look at some of these scripts in more detail.

11.1 The fancy top.info.pl file

Recall that I try to put as much of the chip-specific information as possible into one file – the "info" file. In fact, it is possible to describe everything in the info file.

Let's take a look at dogbert's top.info.pl file in the release set.

The first part provides defaults for the makefile, but the default only takes effect if the file exists:

```
# Set basic stuff
$makefile = "top.flat.make" if (-f "top.flat.make");
$tclsuffix = "dctl";
```

The next two parts are alternative ways of setting subchip-specific variables.

First, you can set them by checking $_acs_top against the desired subchip (perl's creator doesn't like case statements…):

```
# Sub-chip specific var settings (this is an alternative to <subchip>_acs.dctl
# scriptfile).
if ($_acs_top eq "catberthr") {
  #   $_acs_ungroup_cmd = " ungroup -all -flatten; ";
  $_acs_src_const_cmd = " \
    &create_clock -period \$_period_125mhz -port coffeebreak ;\
    source common.const.dctl \
  ";
} elsif ($_acs_top eq "evilceo") {
  #   $_acs_ungroup_cmd = " \
  #     current_design phb ; ungroup -all -flatten ; \
  #     current_design evilceo ; \
  #   "
  @_acs_passes = (0, 2);
}
```

I call this the "pseudo-case" code. In this example, I have avoided creating a catberthr.const.dctl file by burying the clock creation and the source of common.const.dctl in $_acs_src_const_cmd, as was done in the simple example earlier. Commented out are lines that would do ungroup's on catberthr and evilceo. Evilceo is set to do only passes 0 and 2 by setting the @_acs_passes array.

The next section of top.info.pl sets the %clargs hash. Recall that %clargs allows you to apply command-line options from the info file (this is an alternative mechanism to actually using them on the command line):

```
# clargs has any subchip-specific command line arguments.  This is an
# alternative to issuing them on the command line.
%clargs = (
  catberthr => "-pass 0 -pass 1",
  # evilceo uses a special script file to set the _ungroup_cmd before
  # sourcing acs_common.dctl.  It also has one hotmod.
  evilceo => "-hotmod alice -scriptfile evilceo_acs.dctl",
);
```

Notice that I have set catberthr to only run passes 0 and 1. I could just have well have done this by setting @_acs_passes in the "pseudo-case" statement earlier (like I did for evilceo), but I wanted to illustrate that it can be done using %clargs as well. I have made partition "alice" a hotmod in evilceo. I have also used the –scriptfile option to override the call to acs_common.dctl – instead run_acs.pl will call "evilceo_acs.dctl", which will set a value for _ungroup_cmd before calling acs_common.dctl (see below). This does the same thing that the commented-out code above would have done.

Note that either of these could have been done in the pseudo-case statement earlier – the -hotmod option by setting the @hotmods array, and the –scriptfile option by setting $scriptfile. TMTOWTDI.

The next bit sets the @scripts2copy and @constraints2copy arrays:

```
@scripts2copy = (
  "compile.strategy.dctl",    # always include this
);

# Any partition override constraint files here
@constraints2copy = (
#  "dilbert.const.dctl",
);
```

The @scripts2copy array lists the scripts that need to be copied to pass<N>_<subchip>/scripts. This is where ACS looks for override scripts. In my flow, this will always include the compile.strategy.dctl script. Due to a bug in 2002.05 and earlier ACS, it will be copied as "default.strategy" as described earlier.

Similarly, @constraints2copy lists the override constraint files to be copied to pass<N>_<subchip>/constraints. This is used for any partition-specific constraints.

Next we set up any special compile commands:

```
%special_compile_cmds = (
#   "evilceo" => "echo \"Skipping compile\"",
);

%special_compile_cmds_pass1 = (
  # Re-read constraints after pass 1 compile with mapped flag true.
  # Cannot do this on pass 0 because db will not be used for pass 1 compile.
  "catberthr" => "compile -map_effort medium ; echo \"Re-reading constraints
with mapped flag true\" ; set _mapped 1 ; source catberthr.const.dctl",
);
```

%special_compile_cmds is the hash (associative array) that indicates special processing for a partition. For example, most subchips will have no synthesizable logic in them. I don't want DC to do the buffering (the physical tool will do this), so I may want to skip the compile. (In the case of dogbert, both of the subchips have synthesizable logic in them, so this is commented out to show how it would be done).

%special_compile_cmds_pass1 is similar – it indicates pass-specific special processing for a partition (It has priority over any entry in %special_compile_cmds). In this case, I want to re-read the constraint file on pass1 for catberthr after the compile, with the _mapped flag set true. This is handy when you have constraints that can only really be applied to mapped logic.

Since this file is being sourced with the "require" command, we need to return 1 to make perl happy:

```
# when doing require or use we must return 1
1
```

11.2 The env files

There are two "environment" files in my flow. top_env.dctl has the environment information for top level characterization, while compile_env.dctl has environment information for when the make-initiated partition compiles are running. Often, compile_env.dctl just sources top_env.dctl, but they are separated to provide better control over the flow (you *might* want to do something different).

```
# top_env.dctl
# This environment file is used by the subchips in their acs runs.
# A different environment file (compile_env.dctl) is used by the acs
# modules as they compile.
#
# Put various chip-specific info here, as well as any vendor-specific things
# that require a design to be loaded.
#
echo "Start of environment file"

# Create clock periods (this allows them to be adjusted everywhere by editing
# this file.

set _period_100mhz [expr 1000.0 / 100]
set _period_125mhz [expr 1000.0 / 125]

set _input_budget 0.1
set _output_budget 0.1
set _comb_budget 0.0  ;# shouldn't be ANY comb paths at this level!

echo "End of environment file
```

Top_env.dctl for this simple chip just sets the budgets. In more complex chips, you might have code that sets attributes on the design or library, for example.

compile_env.dctl just sources top_env.dctl:

```
# This is the compile-time environment file.  It is sourced by the
# compile.strategy.dctl funnel script, which is sourced in place of the
# compile command by acs-generated <module>.scr scripts.

# Source top-level env file
source top_env.dctl
```

11.3 The evilceo_acs.dctl file

As mentioned above, I have used the %clargs hash in top.info.pl to invoke the –scriptfile option on subchip evilceo, specifying that the script evilceo_acs.dctl should be sourced instead of acs_common.dctl.

Here's what evilceo_acs.dctl looks like:

```
set _acs_ungroup_cmd " \
  current_design phb ; \
  ungroup -all -flatten ; \
  current_design evilceo ; \
  ";

source acs_common.dctl
```

I have set the _ungroup_cmd variable in order to flatten phb before the acs_compile_design. Then I source acs_common.dctl as usual.

11.4 The evilceo.const.dctl file

Recall that I did the clock creation and sourced common.const.dctl for catberthr by setting the $_acs_src_const_cmd variable earlier. For evilceo I have created an evilceo.const.dctl script that looks like this:

```
&create_clock -period $_period_100mhz -port impossible_deadline
&create_clock -period $_period_125mhz -port coffeebreak

source common.const.dctl
```

All it does is create the clocks and call common.const.dctl to set up all the default constraints. The budgets for common.const.dctl are in top_env.dctl. You can certainly add more specific constraints if necessary, but the default ones are usually enough at this level.

Since top_env.dctl was sourced by acs_common.dctl *before* this file, you can also override the budget values before calling common.const.dctl. Or you can override specific paths by adding code *after* calling common.const.dctl.

Note that the code uses *my* proc "&create_clock", *not* the standard create_clock command. My proc adds such things as ideal_net, drive 0, etc when creating the clock. It also keeps a list of all clock ports for use by common.const.dctl. See common_procs.dctl for details.

Note that evilceo.constl.dctl will be sourced by acs_common.dctl because I *didn't* override the $_acs_src_const_cmd in top.info.pl (look at the "pseudo-case statement" section again – I set $_acs_src_const_cmd for catberthr, but I didn't set it for evilceo. It will default to "source ${_acs_top}.const.dctl").

12 Building the top - revisited

12.1 The top.dctl script

In the makefile section, I explained that the top (dogbert) will be built by this entry in the top.flat.make file:

```
dogbert.db: \
        catberthr.db \
        evilceo.db \

        @echo Compiling design : dogbert ;  run_acs.pl -top dogbert -f
top.info.pl -scriptfile top.dctl -noacs -globalsfile globals.v >
dogbert_top.synlog
```

The –noacs option will cause run_acs.pl to just run DC. We have set the scriptfile to top.dctl. Let's take a look at top.dctl:

```
read_verilog {$_acs_globals_files $_acs_verilog_src_files}
link ;# should pick up .db files in the synth root directory

if {[info exists _acs_change_names_cmd]} {
  &cmd eval $_acs_change_names_cmd
} else {
  &cmd change_names -rules verilog -hierarchy
}

write -format verilog -hier -output "${_acs_top}.h.psv" $_acs_top
write -format db -hier -output "${_acs_top}.h.db" $_acs_top

&exit
```

Note how simple this script is. Because you're working at the top of the chip, I think it's a good idea to avoid doing anything fancy – you might run out of memory! The script just reads the verilog (the list is passed by run_acs.pl - $_acs_verilog_src_files will just have one file – the name of the top level verilog), and links. Because the subchip runs have written their db files to the root directory, the link should proceed cleanly.

Next the script does change_names. If the variable $_acs_change_names_cmd is set, it uses this. Otherwise, it uses the default verilog rules.

Finally, the script writes out the results. Done.

12.2 build_top.pl – the final automation

To actually start the whole process off, we would do something like:

```
gmake --jobs --makefile top.flat.make
```

HOWEVER, this doesn't ensure that the top level makefiles themselves are up-to-date. So, I wrote *one more* little perl script called "build_top.pl":

```perl
#!/usr/local/bin/perl

$| = 1;  # force immediate output

# Make sure calling location is in INC path
use File::Basename;
$my_dir = &dirname($0);
$me = &basename($0);
unshift(@INC,"$my_dir");

&process_options;

#$newverilog = `find $vlog -name '*.v' -newer $topflatmake`;
if ((-f $topmake) && (-f $topflatmake)) {
  foreach $globalsfile (@globalsfiles) {
    $newverilog .= `find $globalsfile -newer $topflatmake`;
  }
  foreach $vlog (@vlogs) {
    $newverilog .= `find $vlog -name '*.v' -newer $topflatmake`;
  }
  if ($newverilog) {
    print "Running $mkmakefiles because these files are newer than
$topflatmake:\n $newverilog\n";
    &run_mkmakefiles;
  }
} else {
  print "Running $mkmakefiles because $topmake or $topflatmake doesn't
exist\n" ;
  &run_mkmakefiles;
}

print "\n\nRunning top-level make\n";
system "gmake --jobs --makefile $topflatmake";

sub run_mkmakefiles {

  $cmd = "$mkmakefiles";
  $cmd .= " -top $top" if ($top); # Add in top
  foreach $globalsfile (@globalsfiles) {
    $cmd .= "  -globalsfile $globalsfile";
  }
  foreach $vlog (@vlogs) {
    $cmd .= "  -vlog $vlog";
  }
  foreach $mod (@mods) {
    $cmd .= "  -mod $mod";
  }
  print "cmd is:\n$cmd\n" if ($verbose);
  system "$cmd";

}
```

(rest deleted)

All this does is check to see if there are any verilog files that are newer than the makefile, and run mk_makefiles.pl (which runs synth_make.pl and flatten_top.pl) if there are. It provides hooks for specifying the globalsfiles and alternate paths to the vlog source (see Appendix 2). Then it invokes gmake (at last…).

```
bash-2.05b$ build_top.pl -top dogbert -globalsfile globals.v
Running mk_makefiles.pl because top.make or top.flat.make doesn't exist
No -mods specified - setting mods to "catberthr evilceo"


Running top-level make
Compiling design : catberthr
Compiling design : evilceo
Compiling design : dogbert
bash-2.05b$
```

13 Conclusion

ACS is an excellent tool for automating the synthesis flow. Using it at the subchip level provides a nice balance of performance and control versus automation, especially for large chips. Using the environment and scripts I have provided allows the synthesis engineer to focus on the best optimization strategies – leaving the data management to the tools.

14 Acknowledgements

The author would like to acknowledge the following people for their assistance and review:

Steve Brown (Intel)

15 References

(1) Complex Clocking Situations Using PrimeTime
   Paul Zimmer
   Synopsys Users Group 2000 San Jose
   (available at www.zimmerdesignservices.com)

(2) Working with DDR's in PrimeTime
   Paul Zimmer, Andrew Cheng
   Synopsys Users Group 2001 San Jose
   (available at www.zimmerdesignservices.com)

(3) My Favorite DC/PT Shell Tricks
   Paul Zimmer
   Synopsys Users Group 2002 San Jose
   (available at www.zimmerdesignservices.com)

(4) Push-button Synthesis or, using dc_perl to do_the_right_thing
   Kurt Baty, Steve Golson
   Synopsys Users Group 1998 San Jose
   (available from SOLV-IT, or from www.trilobyte.com)

(5) A comparison of Hierarchical Compile Strategies
   Steve Golson
   Synopsys Users Group 2001 San Jose
   (available from SOLV-IT, or from www.trilobyte.com)

(6) ASIC Flow Engine for Timing Closure and a Makefile Generator to Automate Design Budgeter Methodology
   Tom Tessier, Marvin Anderson
   Synopsys Users Group 2000 San Jose
   (available from SOLV-IT or www.hdl-design.com)

16 Appendix

16.1 The Virtues of Real Clocks – A comparison of the 1-clock and 3-clock technique

16.1.1 Case 1 – timing the simple circuit

Consider the following simple circuit:
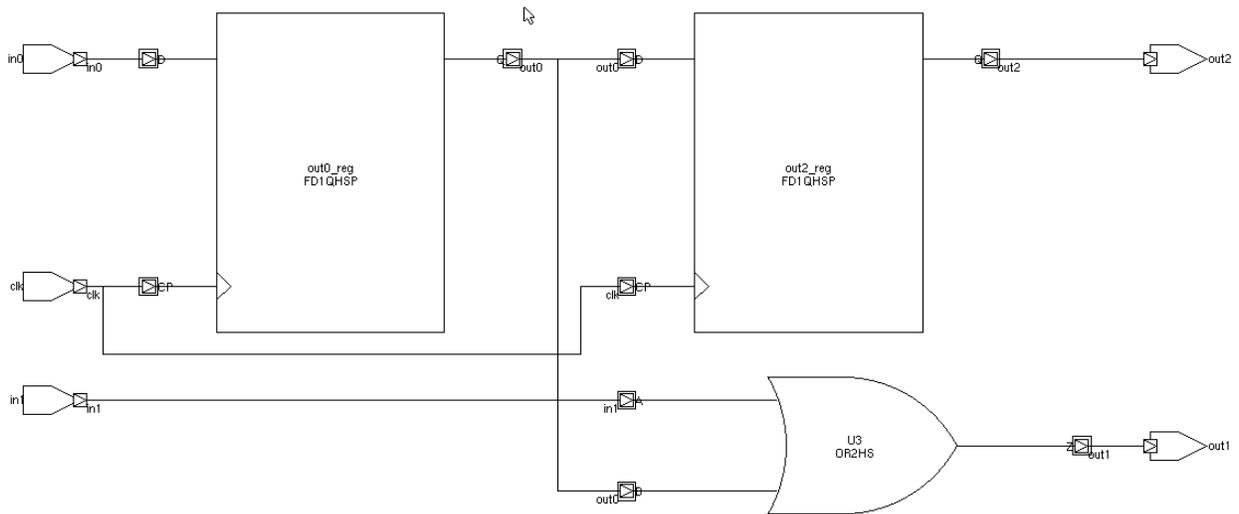
This circuit has all four path types – input-to-clock, clock-to-clock, clock-to-output, and input-to-output (combinational).

Using a clock period of 10 ns, input budget of 3.75 ns, output budget of 1.25 ns, and a combinational delay budget of 5.0 ns, here's what the constraints look like using the 3-clock technique:

```
set _all_inputs_no_clks [remove_from_collection [all_inputs] [get_ports clk]]

# constrain comb paths
create_clock -period 10.0 -name comb_virtual_clk
set_input_delay "1.25" -clock comb_virtual_clk $_all_inputs_no_clks
set_output_delay "3.75" -clock comb_virtual_clk [all_outputs]

# constrain flop -> flop path
create_clock -period 10.0 -name real_clk clk

# constrain input -> flop paths
create_clock -period 10.0 -name io_virtual_clk
set_input_delay "6.25" -clock io_virtual_clk -add_delay $_all_inputs_no_clks

# constrain flop -> output paths
set_output_delay "8.75" -clock io_virtual_clk -add_delay [all_outputs]

set_false_path -from [get_clocks io_virtual_clk] -to [get_clocks io_virtual_clk]
set_false_path -from [get_clocks comb_virtual_clk] -to [get_clocks real_clk]
set_false_path -from [get_clocks real_clk] -to [get_clocks comb_virtual_clk]
set_false_path -from [get_clocks io_virtual_clk] -to [get_clocks comb_virtual_clk]
set_false_path -from [get_clocks comb_virtual_clk] -to [get_clocks io_virtual_clk]
```

I believe the input/output delays in the comb path constraint (1.25/3.75) can be any numbers that add up to 5.0 (the combinational delay budget).

Here's what the constraints look like with the 1-clock techinique:

```
set _all_inputs_no_clks [remove_from_collection [all_inputs] [get_ports clk]]
create_clock -period "10.0" clk

set_input_delay "6.25" -clock clk  $_all_inputs_no_clks
set_output_delay "8.75" -clock clk [all_outputs]

# Handle comb paths
set_max_delay \
  [expr "6.25" + "8.75" + "5.0"] \
  -from $_all_inputs_no_clks -to [all_outputs]
```

Applying these constraints to the circuit, here's what the timing reports look like for the input-to-clock path:

`report_timing` `-path` short `-from` in0

3-clock:

```
   Point                                   Incr         Path
   -------------------------------------------------------------
   clock io_virtual_clk (rise edge)        0.00         0.00
   clock network delay (ideal)             0.00         0.00
   input external delay                    6.25         6.25 f
   in0 (in)                                0.00         6.25 f
   out0_reg/D (fdf1a3)                     0.00         6.25 f
   data arrival time                                    6.25

   clock real_clk (rise edge)             10.00        10.00
   clock network delay (ideal)             0.00        10.00
   out0_reg/CLK (fdf1a3)                   0.00        10.00 r
   library setup time                     -0.16         9.84
   data required time                                   9.84
   -------------------------------------------------------------
   data required time                                   9.84
   data arrival time                                   -6.25
   -------------------------------------------------------------
   slack (MET)                                          3.59
```

1-clock:

```
   Point                                   Incr         Path
   -------------------------------------------------------------
   clock clk (rise edge)                   0.00         0.00
   clock network delay (ideal)             0.00         0.00
   input external delay                    6.25         6.25 f
   in0 (in)                                0.00         6.25 f
   out0_reg/D (fdf1a3)                     0.00         6.25 f
   data arrival time                                    6.25

   clock clk (rise edge)                  10.00        10.00
   clock network delay (ideal)             0.00        10.00
   out0_reg/CLK (fdf1a3)                   0.00        10.00 r
   library setup time                     -0.16         9.84
   data required time                                   9.84
   -------------------------------------------------------------
   data required time                                   9.84
   data arrival time                                   -6.25
   -------------------------------------------------------------
   slack (MET)                                          3.59
```

For the clock-to-clock paths:

```
report_timing -path short -from [get_pins out0_reg/Q] -to [get_pins
out2_reg/D]
```

3-clock:

```
  Point                                 Incr        Path
  ----------------------------------------------------------
  clock real_clk (rise edge)            0.00        0.00
  clock network delay (ideal)           0.00        0.00
  out0_reg/CLK (fdf1a3)                  0.00        0.00 r
  out0_reg/Q (fdf1a3) <-                0.53        0.53 f
  out2_reg/D (fdf1a3)                    0.00        0.53 f
  data arrival time                                 0.53

  clock real_clk (rise edge)           10.00       10.00
  clock network delay (ideal)           0.00       10.00
  out2_reg/CLK (fdf1a3)                  0.00       10.00 r
  library setup time                   -0.19        9.81
  data required time                                9.81
  ----------------------------------------------------------
  data required time                                9.81
  data arrival time                                -0.53
  ----------------------------------------------------------
  slack (MET)                                       9.28
```

1-clock:

```
  Point                                 Incr        Path
  ----------------------------------------------------------
  clock clk (rise edge)                 0.00        0.00
  clock network delay (ideal)           0.00        0.00
  out0_reg/CLK (fdf1a3)                  0.00        0.00 r
  out0_reg/Q (fdf1a3) <-                0.53        0.53 f
  out2_reg/D (fdf1a3)                    0.00        0.53 f
  data arrival time                                 0.53

  clock clk (rise edge)                10.00       10.00
  clock network delay (ideal)           0.00       10.00
  out2_reg/CLK (fdf1a3)                  0.00       10.00 r
  library setup time                   -0.19        9.81
  data required time                                9.81
  ----------------------------------------------------------
  data required time                                9.81
  data arrival time                                -0.53
  ----------------------------------------------------------
  slack (MET)                                       9.28
```

For the clock-to-out path:
```
report_timing -path short -to out1 -from [get_pins out0_reg/Q]
```

3-clock:

```
  Point                                   Incr         Path
  ------------------------------------------------------------
  clock real_clk (rise edge)              0.00         0.00
  clock network delay (ideal)             0.00         0.00
  out0_reg/CLK (fdf1a3)                   0.00         0.00 r
  out0_reg/Q (fdf1a3) <-                  0.53         0.53 f
  ...
  out1 (out)                              0.24         0.77 f
  data arrival time                                    0.77

  clock io_virtual_clk (rise edge)       10.00        10.00
  clock network delay (ideal)             0.00        10.00
  output external delay                  -8.75         1.25
  data required time                                   1.25
  ------------------------------------------------------------
  data required time                                   1.25
  data arrival time                                   -0.77
  ------------------------------------------------------------
  slack (MET)                                          0.48
```

1-clock:

```
  Point                                   Incr         Path
  ------------------------------------------------------------
  clock clk (rise edge)                   0.00         0.00
  clock network delay (ideal)             0.00         0.00
  out0_reg/CLK (fdf1a3)                   0.00         0.00 r
  out0_reg/Q (fdf1a3) <-                  0.53         0.53 f
  ...
  out1 (out)                              0.24         0.77 f
  data arrival time                                    0.77

  clock clk (rise edge)                  10.00        10.00
  clock network delay (ideal)             0.00        10.00
  output external delay                  -8.75         1.25
  data required time                                   1.25
  ------------------------------------------------------------
  data required time                                   1.25
  data arrival time                                   -0.77
  ------------------------------------------------------------
  slack (MET)                                          0.48
```

And for the combinational paths:

```
report_timing -path short -from in1 -to out1
```

3-clock:

```
  Point                                Incr         Path
  --------------------------------------------------------
  clock comb_virtual_clk (rise edge)   0.00         0.00
  clock network delay (ideal)          0.00         0.00
  input external delay                 1.25         1.25 f
  in1 (in)                             0.00         1.25 f
  ...
  out1 (out)                           0.21         1.46 f
  data arrival time                                 1.46

  clock comb_virtual_clk (rise edge)  10.00        10.00
  clock network delay (ideal)          0.00        10.00
  output external delay               -3.75         6.25
  data required time                                6.25
  --------------------------------------------------------
  data required time                                6.25
  data arrival time                                -1.46
  --------------------------------------------------------
  slack (MET)                                       4.79
```

1-clock:

```
  Point                                Incr         Path
  --------------------------------------------------------
  input external delay                 6.25         6.25 f
  in1 (in)                             0.00         6.25 f
  ...
  out1 (out)                           0.21         6.46 f
  data arrival time                                 6.46

  max_delay                           20.00        20.00
  output external delay               -8.75        11.25
  data required time                               11.25
  --------------------------------------------------------
  data required time                               11.25
  data arrival time                                -6.46
  --------------------------------------------------------
  slack (MET)                                       4.79
```

As you can see, although the details of the calculation my differ slightly, the results are the same.

16.1.2 Case 2 – Compiling a complex circuit.

OK, so both technques time the circuit correctly. But they might have different effects on synthesis. The 1-clock technique introduces a special timing exception (the set_max_delay). The 3-clock technique creates path groups. What effect do these things have on synthesis?

To answer this, I compiled a reasonably large, reasonably complex block with fairly tight timing constraints. The block is "rcu" from the picojava design discussed in (1). It has all four path types. I set the period to 2.0ns.

I compiled it with both the standard 1-clock and standard 3-clock techniques. The results look like this:

Compile Technique	Runtime	Area	In2clk	Clk2clk	Clk2out	comb
Standard 1-clock	53	328	-0.72	-0.09	-0.72	-0.72
Standard 3-clock	56	333	-0.72	-0.04	-0.65	-0.47

Runtime is similar. Area is similar. Timing is slightly "better" for the 3-clock technique, but probably only because the group paths told DC that we cared about these paths separately.

Now let's try the 1-clock technique with group_path commands that match the 3-clock technique:

```
group_path -default
group_path -name "real_clk" -from [get_clocks *] -to [get_clocks *]
group_path -name "real_clk" -from $_all_inputs_no_clks -to [get_clocks *]
group_path -name "io_virtual_clk" -from [get_clocks *] -to [all_outputs]
group_path -name "comb_virtual_clk" -from $_all_inputs_no_clks -to
[all_outputs]
```

Compile Technique	Runtime	Area	In2clk	Clk2clk	Clk2out	comb
Standard 1-clock	53	328	-0.72	-0.09	-0.72	-0.72
Standard 3-clock	56	333	-0.72	-0.04	-0.65	-0.47
1-clock with 3-clock group paths	53	330	-0.86	-0.11	-0.58	-0.41

I was hoping for identical results. No such luck. Better on some paths, worse on others.

OK, so now I'll try the 3-clock technique with a set_max_delay command to match the 1-clock technique:

`set_max_delay` 2.0 -from $_all_inputs_no_clks -to [`all_outputs`]

Compile Technique	Runtime	Area	In2clk	Clk2clk	Clk2out	comb
Standard 1-clock	53	328	-0.72	-0.09	-0.72	-0.72
Standard 3-clock	56	333	-0.72	-0.04	-0.65	-0.47
1-clock with 3-clock group paths	53	330	-0.86	-0.11	-0.58	-0.41
3-clock with set_max_delay	54	334	-0.74	-0.09	-0.66	-0.46

Comparing this new result with the standard 3-clock compile seems to indicate that the presence of the set_max_delay has a slight negative effect on results – even though the other constriants are the same.

Let's try it with "optimal" path groups (separate all 4 path types) using both techniques.

Compile Technique	Runtime	Area	In2clk	Clk2clk	Clk2out	comb
Standard 1-clock	53	328	-0.72	-0.09	-0.72	-0.72
Standard 3-clock	56	333	-0.72	-0.04	-0.65	-0.47
1-clock with 3-clock group paths	53	330	-0.86	-0.11	-0.58	-0.41
3-clock with set_max_delay	54	334	-0.74	-0.09	-0.66	-0.46
1-clock with 4 group paths	68	335	-0.85	-0.07	-0.58	-0.36
3-clock with 4 group paths	68	332	-0.66	+0.06	-0.61	-0.44

Still not an exact match. That "+0.06" in the clk2clk group looks really encouraging, though. I ran report_constraints to verify that nothing is weighted in the old path groups (the real_clk, io_virtual_clk, and comb_clk). Is this just a random variation?

16.1.3 Conclusions

Both techniques apply the same constraints, but they may have slightly different results in synthesis. The data doesn't lend itself to sweeping generalizations. There seems to be a chance that the 3-clock technique has some slight advantage, independent of path groups. But the differences are very slight.

In the case of setting top-level (subchip-level) constraints for ACS, none of this is likely to matter. There won't be any comb paths, and the input-to-clock and clock-to-output paths won't have much if any optimizable logic. Since the 1-clock technique is shorter, simpler to write, and makes the reports easier to read, I favor the 1-clock approach.

16.2 Using hierarchical directories

As mentioned earlier, I prefer using a flat synthesis root directory containing all scripts, db's, and chip code in one place. The scripts do have the ability to deal with hierarchy now – I added this after this paper was originally published.

Using hierarchy is fairly simple. You'll need to run build_top.pl from whatever directory you want the pass* directories to go in. You can use the following switches on build_top.pl to specify where the verilog files, script files, and log files should go:

```
  -vlog dir              Look in "dir" for verilog files
                         (can be used multiple times, or used with -recurse)
  -recurse               Look for verilog files recursively in the vlog dir
  -scriptdir dir         Specify directory for scripts (will be passed to
mk_makefiles.pl)
  -logdir dir            Specify directory for synlog files
```

So, if you were building in the current directory (say "build"), and you keep your scripts in "scripts", your logs in "logs", and your verilog in "src" with subdirectories for each level of hierarchy, the command would look like:

```
$> cd build
$> ../scripts/build_top.pl -top dogbert -globalsfile ../src/globals.v -vlog
../src -recurse -scriptdir ../scripts -logdir ./logs
```

Note that the log files generated by ACS will still be in pass*/logs.

When using analyze_acs.pl, be sure to use the –logdir option if you used the –logdir option on build_top.pl.

But I *still* say this structure is a mess…