My Favorite `dc_shell` Tricks

Steve Golson
Trilobyte Systems, 33 Sunset Road, Carlisle MA 01741
Phone: 508/369-9669
Fax: 508/371-9964
Email: sgolson@trilobyte.com

Abstract[†]**: You can make `dc_shell` do amazing and wonderful things.**

1.0 Lists

`dc_shell` supports lists using list expressions delineated by curly braces:

```
dc_shell> list1 = {a, b, c d e}
{"a", "b", "c", "d", "e"}
```

You can separate list elements with either commas or spaces, but `dc_shell` always uses commas when displaying a list.

Lists can be concatenated using the + operator:

```
dc_shell> list2 = {f g h}
{"f", "g", "h"}
dc_shell> list3 = list1 + list2
{"a", "b", "c", "d", "e", "f", "g", "h"}
```

You can add elements to a list in a similar way:

```
dc_shell> list2 = list1 + f + g
{"a", "b", "c", "d", "e", "f", "g"}
```

but watch what happens if you put the new elements at the beginning:

```
dc_shell> list2 = f + g + list1
{"fg", "a", "b", "c", "d", "e"}
```

The first + concatenates the two strings `"f"` and `"g"` into a single string `"fg"`, and then the new string is added to the beginning of the list. Instead try

```
dc_shell> list2 = {f g} + list1
{"f", "g", "a", "b", "c", "d", "e"}
```

You can also subtract elements from a list:

```
dc_shell> list2 = {f g a b c d e}
{"f", "g", "a", "b", "c", "d", "e"}
dc_shell> list3 = list2 - a
{"f", "g", "b", "c", "d", "e"}
```

This can be very useful for reordering the elements of a list. We can move element `"a"` to the end by subtracting it and adding it in a single command[1]:

```
dc_shell> list2 = {f g a b c d e}
{"f", "g", "a", "b", "c", "d", "e"}
dc_shell> list3 = list2 - a + a
{"f", "g", "b", "c", "d", "e", "a"}
```

---

†. An earlier version of this paper was presented at the 1995 Synopsys User's Group Conference.

1. I keep my module names in an alphabetical list, and use this trick to reorder the list so that certain modules are at the end (e.g. if I want to compile all modules using a single `foreach` loop, but hierarchical modules need to compile after all the leaf modules).

If you have duplicate elements they will all be removed with a single subtraction:

```
dc_shell> list2 = {a b a b c d a}
{"a", "b", "a", "b", "c", "d", "a"}
dc_shell> list3 = list2 - a
{"b", "b", "c", "d"}
```

Remember that objects can be typed, and that the typing is invisible. The subtraction only removes elements with the proper type:

```
dc_shell> list2 = find(net,clk) + find(port,clk) + find(clock,clk)
{"clk", "clk", "clk"}
dc_shell> list3 = list2 - find(net,clk)
{"clk", "clk"}
```

but subtracting a string constant will remove all objects with that name, regardless of type:

```
dc_shell> list2 = find(net,clk) + find(port,clk) + find(clock,clk)
{"clk", "clk", "clk"}
dc_shell> list3 = list2 - "clk"
{}
```

Here is how to use list subtraction to ensure that the current directory is at the beginning of your search path:

```
dc_shell> search_path = search_path - "."
{"/usr/synopsys/libraries/syn"}
dc_shell> search_path = "." + search_path
{".", "/usr/synopsys/libraries/syn"}
```

You can have lists of lists:

```
dc_shell> list2 = {a b c {d e f}}
{"a", "b", "c", {"d", "e", "f"}}
dc_shell> foreach (element, list2) {
          list element
          }
element = "a"
element = "b"
element = "c"
element = {"d", "e", "f"}
```

Here is how to extract the first element of a list[2]:

```
dc_shell> foreach (first_element, list2) {
          break
          }
1
dc_shell> list first_element
first_element = "a"
```

Exercise for the reader: build a function that returns everything *except* the first element of a list[3]. You must handle duplicates and lists of lists correctly.

2. car, for all you LISP hackers.
3. cdr, for all you LISP hackers.

2.0 Wildcards

dc_shell supports wildcards using the * character. Thus if you want to find all the designs loaded into memory you can say

```
all_the_designs = find(design,"*")
```

Actually, if you give find no argument at all, the "*" is implied. Thus we can be a bit more succinct:

```
all_the_designs = find(design)
```

If we want to find all designs except those created by DesignWare (adders, comparators, etc.) we can say:

```
all_the_designs = find(design) - find(design,"*DW*")
```

3.0 Variable names

Be careful using names like design. It is a bad idea to do things like

```
design = current_design
```

because further in your script you will no doubt try

```
find(design,foobar)
```

which will die mysteriously. Always use names like the_design or my_design; they are much safer.

Another problem occurs due to variable typing. Consider the variable assignment

```
dc_shell> foo = lkj
"lkj"
```

But we might just as easily get

```
dc_shell> foo = lkj
{"lkj"}
```

In the first case, foo was initialized as a string variable:

```
dc_shell> remove_variable foo
Removing variable 'foo'.
1
dc_shell> foo = ""
Warning: Defining new variable 'foo'. (EQN-10)
""
dc_shell> foo = lkj
"lkj"
```

and in the second case as a list variable:

```
dc_shell> remove_variable foo
Removing variable 'foo'.
1
dc_shell> foo = {}
Warning: Defining new variable 'foo'. (EQN-10)
{}
dc_shell> foo = lkj
{"lkj"}
```

This can cause many mysterious errors, particularly in foreach constructs. Use remove_variable if you aren’t sure.

4.0 Quoting

dc_shell has a few special characters. The double quote mark " is used to delineate a string expression. The semicolon ; is a terminate statement and can be used to separate commands on a single line:

```
dc_shell> current_design ; pwd
Current design is 'foo'.
{"foo"}
"/home/sgolson/play"
```

But what if we want to use these characters in an alias definition, or pass them to sh[4]? They must be escaped with a backslash. Let's say we want to use the shell echo command to echo a single semicolon. From a regular shell prompt we can say

```
$ echo ";"
;
```

In order to pass this to sh from inside dc_shell, the quotes must be escaped:

```
dc_shell> sh "echo \";\""
;
```

Rather than quoting, we can escape the semicolon in the shell:

```
$ echo \;
;
```

The corresponding dc_shell version requires both the backslash and the semicolon to be escaped:

```
dc_shell> sh "echo \\\;"
;
```

Now if we want to create an alias that does this, we have yet another level of quoting:

```
dc_shell> alias semicolon "sh \"echo \\\"\;\\\"\""
1
dc_shell> alias semicolon
sh "echo \";\""
1
dc_shell> semicolon
;
```

and the second version looks like

```
dc_shell> alias semicolon "sh \"echo \\\\\\\;\""
1
dc_shell> alias semicolon
sh "echo \\\;"
1
dc_shell> semicolon
;
```

5.0 Fun aliases

Put this alias in your .synopsys_dc.setup file

```
alias sys_stat "sh \"(date ; ps xv | egrep 'dc_shell|PID')\""
```

4. Remember that sh invokes the Bourne shell, *not* csh.

Then at interesting points in your script you can get a record of wall clock time, CPU time, and process statistics:

```
dc_shell> sys_stat
Thu Mar  2 17:28:14 EST 1995
 PID TT STAT  TIME SL RE PAGEIN SIZE  RSS   LIM %CPU %MEM COMMAND
6805 p4 S     0:20  0  3    242 4676  660    xx  0.0  2.2 dc_shel
```

Here are some more fun ones:

```
alias tokyo_time  "sh \"env TZ=Japan date\""
alias newfie_time "sh \"env TZ=Canada/Newfoundland date\""
```

6.0 Things to do at startup

If you have many servers that you run dc_shell from, you might want to put the following in your .synopsys_dc.setup file:

```
sh "echo Running on `hostname`"
sh cat /etc/motd
```

This leaves a nice record in your log file.

Sometimes you can have mysterious initialization problems. It may be helpful to put messages like

```
echo End of ~sgolson/.synopsys_dc.setup
```

at the end of each .synopsys file.

7.0 How to get the cell name when you have the pin name

Given a list named thepins, this dc_shell alias creates a list named thecells which has the last hierarchical element (i.e. the pin name) stripped off, thus leaving only the cell names.

```
alias get_thecells " \
  sh \" \
    (echo -n \\\"thecells = \\\" \\\; \
     echo \"thepins\") | \
    sed -e 's?/[^,}/][^,}/]*[,}]?,?g' -e 's/,$/}/' > tmp \" \; \
  include tmp \; \
  sh /bin/rm tmp "
```

Caveats: thepins must be a list, i.e. it must have {} around it. There might be only one element in the list, however. If thecells already exists, it had better be a list as well. Also, sed may die if you pass it a list that is too many characters long.

Here is an example. Assume we want to find all the cells connected to port a. First use all_connected to find the pins that port a is wired up to:

```
dc_shell> thepins = all_connected(find(port,a))
{"n123"}
```

Not quite right. What all_connected finds is net n123. It appears that what we really want is to find out what is connected to net n123. We can do this by passing the output of the first all_connected to another all_connected:

```
dc_shell> thepins = all_connected(all_connected(find(port,a)))
{"U4/d", "U5/g", "U6/i", "U7/U8/a", "a"}
```


Almost. Now we have port a itself showing up. So subtract that off, and we'll have only the pins connected to port a, which is what we want:

```
dc_shell> thepins = all_connected(all_connected(find(port,a))) \
                    - find(port,a)
{"U4/d", "U5/g", "U6/i", "U7/U8/a"}
```

Now invoke get_thecells:

```
dc_shell> get_thecells
1
list thecells
thecells = {"U4", "U5", "U6", "U7/U8"}
```

8.0 Returning a result from sh into dc_shell

In the previous example we passed the result of a shell command back into dc_shell by writing to a temp file that later gets included in dc_shell. This is necessary because there is no way to pass an arbitrary output from a shell command back into dc_shell (i.e. there is nothing analogous to using backquote in the shell for command substitution)[5].

You might think that you could set an environment variable in sh, and then use get_unix_variable to read it in. This doesn't work, because sh can only affect the environment of the subshell it executes in, and not the parent shell:

```
dc_shell> sh "(FOO=help \; echo $FOO)"
help
1
dc_shell> get_unix_variable("FOO")
""
```

Normally sh returns a status back to dc_shell: 0 if no arguments are given, and 1 otherwise. This is not very useful. We can instead make sh give us the 8-bit status returned by the shell command itself, by setting this variable:

```
sh_returns_process_status = "true"
```

Now we can do fun things like use the UNIX find command to compare the date stamps on two files. Let's say we are reading in our Verilog source files, but this takes a long time. We might want to save each design as an intermediate db file, and only read in the source if it is newer than the corresponding db file.

The find command in UNIX would look something like this:

```
find db_file -newer verilog_file -print
```

To generate the process status we use test and command substitution:

```
test `find db_file -newer verilog_file -print`
```

From within dc_shell this looks like

```
sh( test `find db_file -newer verilog_file -print` )
```

so by passing the status of sh to if we can determine which file is newer:

```
if (! sh( test `find db_file -newer verilog_file -print` )) {
        /* db_file is newer */
   } else {
        /* verilog_file is newer */
   }
```

5. There is now! Newer versions of Design Compiler have the execute -s command.

Notice the ! that sneaked in. This is because sh defines a “true” result as 0, and “false” otherwise. dc_shell is the opposite! So we need to perform a boolean inversion.

What happens if db_file doesn’t exist? Then the find command will fail, and thus the verilog_file is newer. This is why we don’t say

```
find verilog_file -newer db_file -print
```

because if db_file doesn’t exist, the find will fail, but we interpret that to mean db_file is newer, when actually it doesn’t exist!

So we can conditionally read in all our modules as follows:

```
foreach (module, the_modules) {
    verilog_file = sources_dir + "/" + module + ".v"
    db_file      = db_dir + "/" + module + ".db"

    echo ##### looking for db_file newer than verilog_file
    if (! sh( test `find db_file -newer verilog_file -print` )) {
        echo ##### found it -- reading db_file
        read db_file
    } else {
        echo ##### not found -- reading verilog_file
        read -format verilog verilog_file
        write -out db_file module
        }
    }
```

9.0 RTFM

Sure, the on-line manuals are nifty. Even so, you can still access the old reference manual pages using help[6] in dc_shell. Furthermore, you can display these manual pages from your UNIX prompt just like all the other UNIX man pages! Put the following in your .cshrc (or whatever) file:

```
alias synman "/usr/ucb/man -M $SYNOPSYS/doc/syn/man"
```

Now synman *command_name* from your UNIX prompt will call up the appropriate Synopsys manual pages.

This is preferable to adding the Synopsys directory to your normal MANPATH variable, because there are too many dc_shell commands that have the same name as UNIX commands (e.g. find).

10.0 Thermonuclear ungroup

If you wish to completely remove all hierarchy in a design, usually all you need to do is

```
ungroup -flatten -all
```

but a dont_touch attribute on a design, cell, or reference will prevent the ungroup from proceeding.

Here is a script that will completely ungroup a design, no matter what. First it removes any dont_touch attribute on designs in the hierarchy. Then it does the usual ungroup -flatten -all.

Now, as long as there is any hierarchy remaining (i.e. find(-hierarchy design) is true) the while loop will be active and the dont_touch attribute is removed from the cells and references that still have it. Then ungroup again, and repeat until no more hierarchy remains.

6. Or man.

The redirection to /dev/null prevents the (sometimes long!) lists of cells and references from being echoed to the log file.

```
/* remove dont_touch from designs in hierarchy */
remove_attribute -quiet find(-hierarchy design) dont_touch
/* and ungroup */
ungroup -flatten -all
/* now ungroup any remaining hierarchical cells and references */
while (find(-hierarchy design)) {
  echo "#### find hierarchical cells and references on this level ####"
  hier_cells = filter(find(cell), \
                      "@is_hierarchical==true")   > /dev/null
  hier_refs = filter(find(reference), \
                     "@is_hierarchical==true") > /dev/null
  echo "#### remove dont_touch from cells ####"
  remove_attribute -quiet find(cell,hier_cells) dont_touch
  remove_attribute -quiet find(reference,hier_refs) dont_touch
  echo "#### and ungroup ####"
  ungroup -flatten -all
  }
```

11.0 Converting a list to a string

Some dc_shell commands always return a list, but you may want a string. Let's say you want to take the output of get_attribute, and use it as an argument to another dc_shell command:

```
/* find the direction of the pin */
direction = get_attribute(pin_name, pin_direction)

/* create a port in the current design */
create_port port_name -dir direction
```

This won't work, because variable direction is a list, and when you pass it to create_port bad things will happen.

To get around this, if you know what values are expected, you can build an if-else construct that tries out all the possible values in turn, and creates a new string variable:

```
/* find the direction of the pin */
dirlist = get_attribute(pin_name, pin_direction)

/* convert the direction from a list to a value */
if (dirlist == {in}) {
    direction = "in"
} else if (dirlist == {out}) {
    direction = "out"
} else {
    direction = "inout"
    }

/* create a port in the current design */
create_port port_name -dir direction
```

Another way is to use sh to run a sed script on the list variable that strips off the braces.

12.0 allcells.ss script

Here is a script that uses some of the tricks outlined here.

Given a technology library, it creates a design that contains one cell of each type in the library. This is useful when you are creating a new library.

```
/* @(#)allcells.ss  1.1 11/21/94 19:27:15 */

/*
** allcells.ss
**
** This script creates a new design containing one cell of each type
** in a given library. This is useful for testing new libraries.
*/

/* change this to whatever library you want */
thelib = "vlib"

read thelib + ".db"

/* create a new design */
remove_design allcells
create_design allcells

current_design = allcells

/* initialize the port names and instance names */
portnum = 0
unum = 0

/* find all the cells in the library */
foreach(cell, find(lib_cell, thelib + "/*")) {

    /* instance name */
    instance_name = U + unum
    unum = unum + 1

    /* create a cell in the current design */
    create_cell instance_name cell

    /* find all the pins on the library cell */
    foreach(thepin, find(lib_pin, cell + "/*")) {
        port_name = "p_" + portnum
        net_name = "net" + portnum
        portnum = portnum + 1

        pin_name = cell + "/" + thepin

        /* find the direction of the pin */
        dirlist = get_attribute(pin_name, pin_direction)

        /* convert the direction from a list to a value */
        if (dirlist == {in}) {
            direction = "in"
```

```
        } else if (dirlist == {out}) {
            direction = "out"
        } else {
            direction = "inout"
            }

        instance_pin = instance_name + "/" + thepin

        /* create a port in the current design */
        create_port port_name -dir direction

        /* and a net */
        create_net net_name

        /* now wire up the port and the pin */
        connect_net net_name { instance_pin port_name }
        }
    }

write
/* end of script allcells.ss */

exit
```