Push-button Synthesis

or,

Using `dc_perl` to `do_the_right_thing`

Kurt Baty

WSFDB Consulting
26 Hill Street
Medway MA 02053
Phone: +1.508.429.4198
Email: kurt@wsfdb.com

Steve Golson

Trilobyte Systems
33 Sunset Road
Carlisle MA 01741
Phone: +1.978.369.9669
Fax: +1.978.371.9964
Email: sgolson@trilobyte.com

ABSTRACT

We have developed a methodology to automatically synthesize large hierarchical designs. This methodology combines the advantages of bottom-up compilation with top-down rebudgeting.

Starting with only the Verilog source code, all required makefiles, synthesis scripts, and constraint files are automatically derived. An overconstraining leaf module time-budgeting method is used for initial synthesis.

For subsequent synthesis runs, a top-level constraint file (perhaps manually generated) is used to automatically create constraints for leaf modules. Timing information is extracted from top-level timing reports. True timing budgets can be generated while avoiding the limitations of `characterize`.

1.0 Introduction

There are several basic techniques for compilation of large hierarchical designs [1]. All have their advantages and disadvantages.

1.1 Top-down hierarchical compile

This methodology uses a top-level constraint file and a simple hierarchical `compile` to generate the entire design at once. Reference [12] describes this methodology.

+ Good quality of results due to optimization over the entire design
+ Only needs top-level constraints
- Long compile times
- Incremental changes require complete recompilation
- Requires `uniquify` for multiply-instantiated designs
- Limitations on design style (single clock, etc.)

1.2 Bottom-up compile

This methodology compiles leaf modules using individual scripts and constraint files. References [7] and [10] describes this methodology.

+ Clarity and simplicity of individual scripts
+ Good results for many designs
+ Incremental changes can be compiled quickly
- Top-level critical paths may not be leaf module critical paths
- Manual updating of scripts and constraints is tedious
- Limitations on design style (restrictions on combinational Mealy paths, etc.)

The use of default scripts and constraints with the UNIX `make` command was first presented in [2]. Using `make` with standard suffix rules was described in [3] and [4]. This simplified the compile process, and because `make` handles all the design hierarchical dependencies, modules that have not been modified are not recompiled, thus saving time.

1.3 Bottom-up compile with top-down constraint propagation

Recently many techniques have been described for using a bottom-up compilation strategy while automating the updating of scripts and constraints.

References [6], [8], [11] discuss this strategy. However all use `characterize` to propagate timing constraints down the hierarchy, thus they are subject to the classic “ping-pong” effect.

2.0 What is the right thing to do?

What we would like is a technique that combines the best of these methodologies!

+ A static predictable re-synthesis state
+ Clarity and simplicity of individual scripts
+ Automatic generation of scripts and constraints
+ Automatic updating of constraints using true time budgeting
+ Good results for all types of designs (many clocks, multicycle paths, multiple Mealy paths, etc.)

We call this methodology `do_the_right_thing`.

The `do_the_right_thing` command is an extension of `dc_perl` which is described in [9].

2.1 Automatic generation of default scripts and constraints

Given only Verilog source code, `do_the_right_thing` will automatically perform initial compilation using an overconstraining leaf module bottom-up compile strategy [10]. Minimal input from the user is required (top-level module name, target library, operating conditions, clock speed).

The design hierarchy is extracted automatically. Script and constraint files are created for each module in the hierarchy. A `Makefile` is generated which has all the proper dependencies.

The result is an initial full-chip synthesis requiring very little input from the user. Many designs will have successful results from this methodology [10].

2.2 Automatic time budgeting

A top-level constraint file specifies "hard" timing requirements at the inputs and outputs of your chip. Wireload models, operating conditions, clocks, and perhaps point-to-point timing exceptions (multicycle paths and false paths) are also given. By running `report_timing` with these top-level constraints applied, the true critical paths of the chip are reported.

If your design is not meeting its timing constraints, the leaf module constraint files must be updated. Previous efforts to propagate top-level constraints downward to the leaf modules have all used the `characterize` command. We chose to use `report_timing` which allows true time budgeting to be accomplished.

Every pin in your design (except the pins of leaf cells) represents the port of a subdesign. By executing `report_timing -through` *`thepin`* you get a timing report which describes the critical path through that pin (which represents the port of a subdesign).

The total delay of the path may have positive or negative slack. Each subdesign that the path traverses provides some percentage of the total delay. A *time budget* for this path is created by normalizing the total delay to the desired clock period, with each subdesign allocated a percentage of the budget according to the percentage of the actual delay used by that subdesign.

Now the input delay or output delay for the subdesign port corresponding to *`thepin`* may be set. If a subdesign is multiply instantiated, use the worst constraint of the path through each instance.

For top-level critical paths this will tighten constraints in leaf modules. For paths that are already meeting timing (positive slack), the leaf module constraints will be relaxed, allowing area reduction.

`do_the_right_thing` will take a top-level constraint file along with a gate-level netlist, and automatically generate new leaf module constraint files using this time budgeting methodology.

+ Top-level critical paths are also the leaf module critical paths
+ Works for designs with any number of subdesigns in the critical path (multiple combinational or Mealy modules)
+ Converges on a timing budget with no “ping-pong” effect
+ Maintains all the advantages of `make`-driven bottom-up compile
+ Input drive strengths and output loads can also be derived from `report_timing`

3.0 Gory details

3.1 `dc_perl`

`dc_perl` uses the Perl interpreter as a “wrapper” around `dc_shell`, thus allowing powerful extensions to `dc_shell` to be created. `dc_shell` commands can be generated by Perl, and the results analyzed by Perl in real time (not post-processed). Further `dc_shell` commands can be algorithmically generated by Perl based on the given results.

Since its first description last year [9] a number of bug fixes and enhancements have been made to `dc_perl`. In particular support for Solaris and HP-UX has been added, and the user interface is much more robust.

3.2 Creation of `environment` file

The `environment` file specifies chip-wide defaults such as operating conditions, clock period, nominal drive strength of module inputs, and nominal loading on module inputs and outputs [10].

`do_the_right_thing` tests to see if an `environment` file exists in the current directory. If not, one is automatically created by suggesting default values and asking the user for confirmation:

```
well there’s no environment so I guess we have to make one!
Loading db file ‘/RAID/tools/asic_libs/vlsi/vsc883.db’
{}
Operating Conditions:

    Name       Library       Process     Temp    Volt    Interconnect Model
    -----------------------------------------------------------------------
    TYP        vsc883           1.00    25.00    3.30    balanced_tree
    TYP_3V     vsc883           1.00    25.00    3.00    balanced_tree
    MAX_3V     vsc883           1.25   150.00    2.70    balanced_tree
    MIN_3V     vsc883           0.76   -50.00    3.30    best_case_tree
    WCMAX      vsc883           1.25   150.00    3.00    balanced_tree
    BCMIN      vsc883           0.76   -50.00    3.60    best_case_tree

The operating conditions you want set are [MAX_3V] : WCMAX
The default clock period you want is [10] :
A good drive nand gate in your library is [nd02d2] :
The cells-to-gates ratio for your library has a value of [1] :
```

A typical `environment` file looks like:

```
/*########################################################################*/
/*#                                                                      #*/
/*#    This environment file was generated by dc_perl for the design     #*/
/*#                          your_design_name                            #*/
/*#                                                                      #*/
/*########################################################################*/

set_operating_conditions WCMAX
set_wire_load -mode segmented
suppress_errors = suppress_errors + {EQN-10 UID-401}
default_clock_period = 10  * (1 - .20) /* 20% timing margin */
set_driving_cell -cell nd02d2 all_inputs()
set_load 4 * load_of(vsc883/nd02d2/a1) all_inputs()
set_load 20 * load_of(vsc883/nd02d2/a1) all_outputs()
max_transition 6.0
cells_to_gates_ratio = 1
```

3.3 Creation of `Makefile`

An implicit rule and assumption in this design methodology is a strict suffix rule oriented file naming convention. That is, each `module_name.v` file contains only one Verilog module declaration as `module_name`. This allows the `Makefile` to use simple suffix rules [3] [4]. All of the suffixes are defined by `do_the_right_thing` and may be changed at the user's option:

```
$hld_langage = "verilog";
$hdl_suffix = ".v";
$constraint_file_suffix = ".const";
$script_file_suffix = ".script";
$gate_level_file_suffix = ".psv";
$synthesis_log_file_suffix = ".synlog";
$design_ware_or_template_suffix = ".dwt";
```

The `make` suffix rules:

```
.SUFFIXES: .v .const .script .db .dwt .psv .synlog

.v.const:
    touch $*.const

.const.script:
    touch $*.script

.script.db:
    dc_shell -f $*.script > $*.synlog


.v.dwt:
    dc_shell -f $*.script > $*.synlog
```

The determination of design dependencies is done recursively from each point in the design hierarchy.

```
building make dependences for design fifo
 the design "fifo" has a sub-design "fifo_ctrl"
 the design   "fifo_ctrl" has a sub-design "fifo_ctrl_decoder"
 the design      "fifo_ctrl_decoder" has no fifo_ctrl_decoder.const file creating one
 the design      "fifo_ctrl_decoder" has no fifo_ctrl_decoder.script file creating one
 the design   "fifo_ctrl" has no fifo_ctrl.const file creating one
 the design   "fifo_ctrl" has no fifo_ctrl.script file creating one
 the design "fifo" has a sub-design "fifo_mux"
 the design   "fifo_mux" has no fifo_mux.const file creating one
 the design   "fifo_mux" has no fifo_mux.script file creating one
 the design "fifo" has a sub-design "fifo_row"
 the design   "fifo_row" has no fifo_row.const file creating one
 the design   "fifo_row" has no fifo_row.script file creating one
 the design "fifo" has no fifo.const file creating one
 the design "fifo" has no fifo.script file creating one
```

At each point in the hierarchy all sub-designs must be found. The Perl subroutine `&get_sub_design_list()` returns a list of the sub-designs of the current design. This shows the `dc_shell` command invoked by Perl:

```
<current_design>_sub_designs = {};

foreach( each_ref_name, filter(find(reference)  \
     @is_black_box == true && @is_unmapped == true \
     && @is_synlib_module == false  && @is_synlib_operator == false  \
     && @is_combinational == false))  {
         if((get_attribute(-quiet each_ref_name, is_a_generic_tristate) != true) \
              && (get_attribute(-quiet each_ref_name, is_a_generic_seq) != true) \
              && (get_attribute(-quiet each_ref_name, hdl_template) == {})) {
     <current_design>_sub_designs =  <current_design>_sub_designs  \
     - each_ref_name + each_ref_name;
         }
     }
```

Any dependencies on local DesignWare or template parts must be determined. The Perl subroutine `&get_local_dw_list()` invokes the following `dc_shell` command to find such parts:

```
<current_design>_local_dw = {};

foreach( each_ref_name, filter(find(reference)  \
     @is_black_box == true && @is_unmapped == true \
     && @is_synlib_module == false  && @is_synlib_operator == false  \
     && @is_combinational == false))  {
         if((get_attribute(-quiet each_ref_name, is_a_generic_tristate) != true) \
              && (get_attribute(-quiet each_ref_name, is_a_generic_seq) != true) \
              && (get_attribute(-quiet each_ref_name, hdl_template) != {})) {
     <current_design>_sub_designs =  <current_design>_sub_designs  \
     - get_attribute(each_ref_name,hdl_template) \
     + get_attribute(each_ref_name,hdl_template);
         }
     }
```

Dependencies on preprocessor `'include` files also must be found. The Perl subroutine `&get_include_files_list()` recursively finds these files:

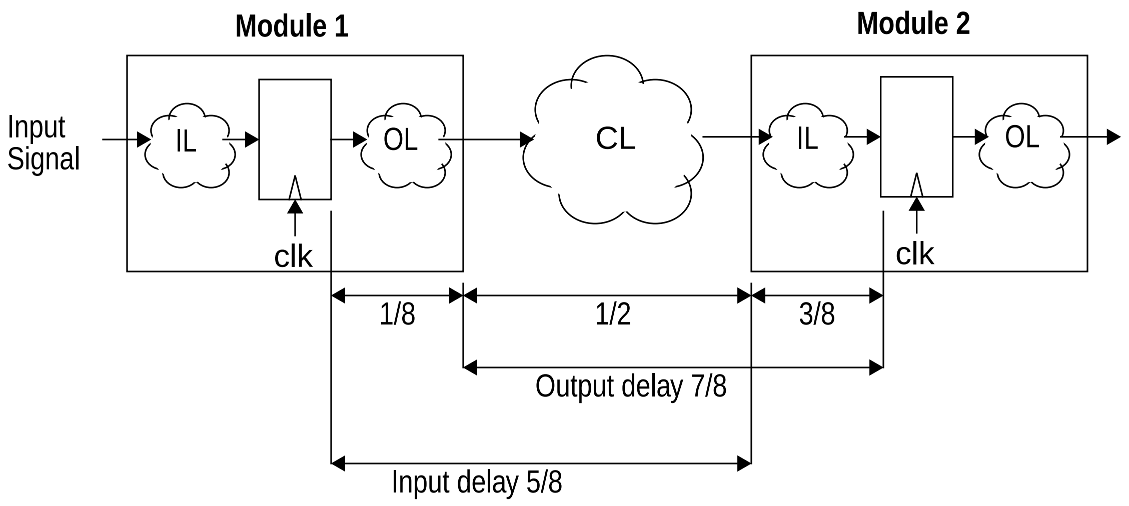
```
########################################
## @include_files = &get_include_files_list();
##
## this subroutine returns a list of the include files
## of the current design
##
##

sub get_include_files_list {
     my (@include_files);
     $_ = &get_dc_shell_variable("current_design");
     s/.*: *//;
     @include_files = &_process_include_file("$_$hdl_suffix",1000);
     @include_files = sort(@include_files);
     return @include_files;
}

########################################
## @include_files = &_process_include_file($a_file_name,$next_filehandle);
##
## Recursively find `include files.
##

sub _process_include_file {
     my (@include_filename_list);
     my ($filename, $filehandle) = @_;
     $filehandle++;
     unless (open($filehandle, $filename)) {
         print STDERR "Error: Can't open $filename: $!\n";
         return;
     }
     while (<$filehandle>) {
         chop;
         if (/^\s*`include(\s+)"(\S+)"/) {
     @include_filename_list = (@include_filename_list, $2);
     @include_filename_list = (@include_filename_list,
       &_process_include_file($2, $filehandle));
     next;
         }
     }
     close($filehandle);
     return @include_filename_list;
}
```

3.4 Creation of default constraint file `module_name.const`

The default module constraint file supports an overconstraining leaf module bottom-up compile strategy [10]. Four types of paths must be properly constrained:

- Flop to flop (clock period)
- Input to flop (`set_input_delay` and clock period)
- Flop to output (`set_output_delay` and clock period)
- Input to output combinatorial or Mealy path (`set_input_delay` and `set_output_delay` using virtual input and output clocks)

The timing budget assumes a single Mealy module in between the flops:



OL: Ouput forming Logic
IL: Input forming Logic
CL: Combinatorial Logic

Here is a typical constraint file:

```
/*########################################################################*/
/*#                                                                      #*/
/*#     This constraint file was generated by dc_perl for the design     #*/
/*#                              fifo_ctrl                               #*/
/*#                                                                      #*/
/*########################################################################*/
/* dc_perl's breadcrumb_trail: {"0", "{"clk"}", "0", "0"} */

reset_design
include environment
clock_period = default_clock_period /* this is a guess */
create_clock -period clock_period -name inputs_virtual_clk
create_clock -period clock_period -name outputs_virtual_clk
create_clock -period clock_period clk
set_drive 0 clk
set_input_delay clock_period * 5 / 8 all_inputs()
set_input_delay clock_period * 1 / 8 all_inputs() -clock inputs_virtual_clk
set_output_delay clock_period * 7 / 8 all_outputs()
set_output_delay clock_period * 3 / 8 all_outputs() -clock outputs_virtual_clk
max_area 1000 * cells_to_gates_ratio
set_dont_touch find(reference,fifo_ctrl_decoder)
```

Notice that submodules have `dont_touch` attributes set on their reference to avoid `uniquify` problems.

3.5 Creation of default script file `module_name.script`

A medium effort compile is used, followed by ungrouping of any DesignWare subdesigns. The `set_dont_touch` in the constraint file will prevent ungrouping of user-created hierarchy. A final `compile -incremental` is done, and the design is written out in `.db` and netlist formats.

```
/*########################################################################*/
/*#                                                                      #*/
/*#        This script file was generated by dc_perl for the design      #*/
/*#                             fifo_ctrl                                #*/
/*#                                                                      #*/
/*########################################################################*/
/* dc_perl’s breadcrumb_trail: {“1”, “0”, ““} */

read -format verilog fifo_ctrl.v
link
verbose_messages = “false”
include fifo_ctrl.const
verbose_messages = “true”
compile
ungroup -all -flatten
compile -incremental
check_design -one_level
create_schematic
write
write -f verilog -output fifo_ctrl.psv
report_area
report_constraint -verbose
report_timing -path full -max_paths 4
exit
```

The `create_schematic` is done here to save time when opening the `.db` file in Design Analyzer.

Some interesting reports are run with the outputs kept in the synthesis log file.

3.6 Invoking `make`

After building the `Makefile` and all the module script and constraint files, `do_the_right_thing` invokes `make` to compile the complete design.

From this point onward `make` can be invoked directly without using `do_the_right_thing`.

4.0 References

[1] "High-Level Design Methodology Overview", Ken Nelsen. Synopsys Online Documentation: Methodology Notes

[2] "Script Automation For Efficient Compilation", Glenn Dukes. SNUG 1993

[3] "Using `make` and `sccs`", Kurt Baty. SNUG 1993

[4] "System Design and Validation", Kurt Baty. SNUG 1994

[5] "My favorite `dc_shell` Tricks", Steve Golson, SNUG 1995

[6] "Auto-Synthesis", Leonard J. LaPadula. SNUG 1996

[7] "Synthesis of a Million Gates", Don Mills. SNUG 1997

[8] "The Boa Methodology", Wilson Snyder. SNUG 1997

[9] "`dc_perl`: Enhancing `dc_shell` Using A Perl Wrapper", Steve Golson. SNUG 1997

[10] "Evolvable Makefiles and Scripts for Synthesis", Anna Ekstrandh, Wayne Bell. SNUG 1997

[11] "Synopsys Makefile for Automation of Required Timing: SMART", Rodney Ramsay. SNUG 1997

[12] "Synthesis Methodology for Large Designs, Design Compiler 1997.01 Release", Don Chan, Susan Runowicz-Smith. Synopsys, Inc., June 1997

5.0 Availability

All `dc_perl` and `do_the_right_thing` scripts can be retrieved via anonymous ftp from

```
ftp://ftp.ultranet.com/pub/sgolson/dc_perl
```

Please contact the authors if you have any comments or suggestions regarding `dc_perl` or `do_the_right_thing`.